Compositional Synthesis of Reactive Systems

A dissertation submitted towards the degree
Doctor of Natural Sciences (Dr. rer. nat.)
of the Faculty of Mathematics and Computer Science
of Saarland University

Noemi Estrid Passing

Saarbrücken, 2023

Abstract

Synthesis is the task of automatically deriving correct-by-construction implementations from formal specifications. While it is a promising path toward developing verified programs, it is infamous for being hard to solve. Compositionality is recognized as a key technique for reducing the complexity of synthesis. So far, compositional approaches require extensive manual effort. In this thesis, we introduce algorithms that automate these steps.

In the first part, we develop compositional synthesis techniques for distributed systems. Providing assumptions on other processes' behavior is fundamental in this setting due to inter-process dependencies. We establish delay-dominance, a new requirement for implementations that allows for implicitly assuming that other processes will not maliciously violate the shared goal. Furthermore, we present an algorithm that computes explicit assumptions on process behavior to address more complex dependencies.

In the second part, we transfer the concept of compositionality from distributed to single-process systems. We present a preprocessing technique for synthesis that identifies independently synthesizable system components. We extend this approach to an incremental synthesis algorithm, resulting in more fine-grained decompositions. Our experimental evaluation shows that our techniques automate the required manual efforts, resulting in fully automated compositional synthesis algorithms for both distributed and single-process systems.

Zusammenfassung

Synthese ist die Aufgabe korrekte Implementierungen aus formalen Spezifikation abzuleiten. Sie ist zwar ein vielversprechender Weg für die Entwicklung verifizierter Programme, aber auch dafür bekannt schwer zu lösen zu sein. Kompositionalität gilt als eine Schlüsseltechnik zur Verringerung der Komplexität der Synthese. Bislang erfordern kompositionale Ansätze einen hohen manuellen Aufwand. In dieser Dissertation stellen wir Algorithmen vor, die diese Schritte automatisieren.

Im ersten Teil entwickeln wir kompositionale Synthesetechniken für verteilte Systeme. Aufgrund der Abhängigkeiten zwischen den Prozessen ist es in diesem Kontext von grundlegender Bedeutung, Annahmen über das Verhalten der anderen Prozesse zu treffen. Wir etablieren *Delay-Dominance*, eine neue Anforderung für Implementierungen, die es ermöglicht, implizit anzunehmen, dass andere Prozesse das gemeinsame Ziel nicht böswillig verletzen. Darüber hinaus stellen wir einen Algorithmus vor, der explizite Annahmen über das Verhalten anderer Prozesse ableitet, um komplexere Abhängigkeiten zu berücksichtigen.

Im zweiten Teil übertragen wir das Konzept der Kompositionalität von verteilten auf Einzelprozesssysteme. Wir präsentieren eine Vorverarbeitungmethode für die Synthese, die unabhängig synthetisierbare Systemkomponenten identifiziert. Wir erweitern diesen Ansatz zu einem inkrementellen Synthesealgorithmus, der zu feineren Dekompositionen führt. Unsere experimentelle Auswertung zeigt, dass unsere Techniken den erforderlichen manuellen Aufwand automatisieren und so zu vollautomatischen Algorithmen für die kompositionale Synthese sowohl für verteilte als auch für Einzelprozesssysteme führen.

Acknowledgements

I am very grateful to my advisor Bernd Finkbeiner for introducing me to the fascinating topic of compositional reactive synthesis. I deeply appreciate his guidance and valuable advice during the last few years as well as his insightful views on my research. The countless hours of thriving discussions did not only shape this thesis but also let me grow personally.

I want to thank Bernd Finkbeiner, Gideon Geier, Philippe Heim, Jana Hofmann, Florian Kohn, Kaushik Mallik, Stefan Oswald, Malte Schledjewski, Anne-Kathrin Schmuck, and Maximilian Schwenger for the fruitful collaboration during our joint research endeavors. Many thanks go to my colleagues Jan Baumeister, Raven Beutner, Norine Coenen, Arthur Correnson, Matthias Cosler, Peter Faymonville, Hadar Frenkel, Michael Gerke, Christopher Hahn, Jesko Hecking-Harbusch, Jana Hofmann, Swen Jacobs, Felix Klein, Florian Kohn, Sabine Nermerich, Niklas Metzger, Mouhammad Sakr, Christa Schäfer, Malte Schledjewski, Frederik Schmitt, Maximilian Schwenger, Julian Siber, Leander Tentrup, Hazem Torfah, Alexander Weinert, and Martin Zimmermann from the *Reactive Systems Group* for all the discussions, coffee breaks, lunch breaks, and delicious cakes. I am particularly thankful to my office mates Maximilian and Matthias for all the fun and the shared love for office plants. I want to thank Werner Damm and Anne-Kathrin Schmuck for their time and effort in reviewing this thesis and I sincerely appreciate their constructive feedback.

I am grateful to Alexandra, Azin, Carolyn, Clara, Jana, Kathrin, Nathalie, Norine, and Sebastian for making my time in Saarbrücken such a wonderful experience. Thank you for the countless *Tatort* nights, girl's nights, theatre visits, iCoffee breaks, knitting afternoons, walks at Staden, dinners at St. Johanner Markt, and all the joint cooking events. Thank you for always having my back and cheering me up when needed. Furthermore, I want to thank Anja, Jana, Kathrin, and Susanne for almost twenty years of friendship and for never getting tired of asking what my Ph.D. is about in almost every call.

I want to express my gratitude to my family for supporting me in every possible way and for always believing in me. Finally, I owe special thanks to Jesko. Thank you for your unconditional love and support through all these years. Thank you for your patience with me working late, for listening to me when I desperately needed to talk about crazy cycle structures in alternating co-Büchi automata on a Sunday evening, and for being the best rubber duck I can imagine. Your never-ending encouragement was crucial for bringing this thesis into existence.

Contents

Chapter 1

Introduction

Over the last decades, computer systems have evolved into a part of the fabric of life. Nowadays, we are surrounded by digital systems and interact with them on numerous occasions every day. We greatly benefit from the tremendous technological advances as they allow us, for instance, to place our lives in the hand of medical systems such as heart-lung machines or cardiac pacemaker devices, to fly with airplanes that allow the pilot to rely on the support of an extensive autopilot system, and to even let the vision of the far-reaching use of self-driving vehicles seem realistic. As most of our critical infrastructure depends on computer systems today, the dependability and robustness of such safety-critical systems are indispensable.

More and more of these systems are of a reactive nature. Instead of receiving a single input and computing an output based on it, they continually interact with their environment and run for an indefinite time, resulting in an infinite input-output-behavior. Typical examples for such *reactive systems* [HP84] are hardware circuits, embedded devices, and communication protocols. The infinite behavior and, in particular, the possible need for both repeating tasks and attending to tasks only triggered by inputs renders the development of correct reactive systems particularly challenging [HP84].

Formal methods are a branch of computer science that addresses the *provable* correctness of systems, including reactive systems. It aims at developing automated techniques for proving that a system satisfies specific properties – called *verifying* a system – as well as for constructing systems that inherently satisfy specific properties – called *synthesizing* a system. Synthesis is an up-and-coming technique for developing formally verified programs as it eliminates the need for manual, and thus error-prone, implementation tasks. It allows a developer to focus on *what* a system should do instead of *how* it should be done. Furthermore, synthesis is able to detect contradictory system specifications, which prevent the existence of an implementation that realizes them, early in the design process. In such situations, synthesis provides a counterexample, which can guide the developer in refining the specification.

Naturally, synthesis is thus one of the grand visions of computer science. It was first formulated by Alonzo Church in the late 1950s [Chu57] and has been an intriguing challenge ever since. In the last decade, there have been breakthroughs in terms of practical applications of synthesis, such as the synthesis of a controller for the AMBA AHB bus protocol [BGJ+07, Job07, GCH13, BJP+12], an industrial standard for the on-chip communication of functional blocks in system-on-a-chip devices. Moreover, several tools (see, e.g., [FFRT17, MSL18, MC18, RSDP22, Kha21]) that automatically construct correct implementations from formal specifications have been developed. Since 2014, these synthesis tools compete in the annual reactive synthesis competition *SyntComp* [BEJ14] on, today, roughly 1000 standard benchmarks. Until today, the synthesis competition heavily facilitates the development of synthesis tools. Despite these recent advances, the vision of synthesis is far from becoming a reality: the success stories are limited to relatively small systems, and currently available synthesis tools do not scale to complex system designs as commonly used in today's practice.

Verification, in contrast, scales to much larger and more complex systems. Consequently, it has already proven its applicability in industry (see, e.g., [HSLL97, BDG$^+$04, CGP02, JGK$^+$15]). *Compositionality* has long been recognized as the key technique that makes a "significant difference" [dRLP98] for the scalability of verification algorithms. Compositional approaches break down the analysis of a complex system into several smaller tasks over individual components, which can then be solved independently. Afterward, the individual analysis results can be recomposed into a solution for the entire system. Naturally, applying compositional techniques to synthesis is thus a promising path to pursue.

In synthesis, however, developing successful compositional algorithms has proven much more challenging. In a nutshell, synthesis seeks an implementation that satisfies the given specification for all behaviors of the system's environment. In compositional synthesis, we seek an implementation of an individual system component that satisfies the specification for all behaviors of the *component's environment*. The component's environment also includes the remaining components of the system. Thus, we consider the other components to be adversarial. However, a component implementation that satisfies the specification for all behaviors of the component's environment rarely exists: such an implementation needs to guarantee the satisfaction of *all* system requirements – even of those that specify the behavior of other parts of the system – irrespective of whether or not the other components cooperate in the goal of satisfying the system specification. In practice, the satisfaction of the requirements for the whole system usually cannot be guaranteed by one component alone but requires collaboration between several components. For successful compositional synthesis, it is, therefore, crucial to identify the connections between system components and their dependencies induced by the system requirements. The critical question is what a component needs to know about other components and their behavior in order to be able to satisfy the specification for all behaviors of its environment. So far, identifying this knowledge and including it in the synthesis tasks for the individual components has been a manual task. We develop techniques that automate the extensive manual effort and address the question from two angles.

In the first part of this thesis, we focus on *distributed systems*, i.e., systems that inherently consist of several components, so-called processes. The distributed synthesis problem seeks implementations for all processes such that their composition satisfies the given system specification. Due to the challenges outlined above, synthesizing implementations for the individual processes separately often does not succeed. Therefore, classical distributed synthesis algorithms rely on directly synthesizing an implementation for the whole system, thus considering the composition of the processes. We introduce two approaches that, in contrast, enable compositionality for distributed synthesis. The first technique relies on weakening the requirement posed on implementations. Instead of implementations that satisfy the given specification in every situation, we are interested in *best-effort* implementations, which are allowed to violate the specification in certain situations. Intuitively, such an implementation "gives its best" to meet the goal but is not guaranteed to do so. When carefully designing best-effort notions, the satisfaction of the overall system specification can still be ensured when using best-effort implementations for all processes. We introduce a notion of best effort for implementations, called *delay-dominance*, together with an automaton-based criterion such that whenever the criterion is satisfied, the best-effort notion induces a sound compositional synthesis algorithm for distributed systems. Utilizing best-effort implementations allows for posing an *implicit assumption* on the other processes, namely that they will not maliciously violate the specification. As the strategies for all processes are best-effort strategies, this implicit assumption is satisfied. For systems with complex interconnections between the processes, however, this implicit assumption does not suffice. Instead, more sophisticated assumptions on the concrete behavior of other processes might be necessary. We thus introduce a second technique for compositional distributed synthesis that automatically derives additional guarantees on the behavior of every process. These guarantees, so-called *certificates*, then provide essential information for the individual synthesis tasks: an implementation is only required to satisfy the specification if the other processes do not deviate from their guaranteed behavior.

In the second part of this thesis, we transfer the concept of compositional synthesis from distributed systems to systems consisting of a single process, so-called *monolithic systems*. The most challenging task in compositional monolithic synthesis is the decomposition of the system into smaller components. For distributed systems, the individual processes naturally serve as these components. For monolithic systems, however, suitable decomposition algorithms are necessary. As the success of compositional synthesis highly depends on the decomposition, we introduce two algorithms for selecting components. The first technique focuses on computing a decomposition such that, given a realizable specification for the system, the synthesis tasks for the individual components are guaranteed to succeed. For unrealizable specifications, our approach guarantees that the synthesis task for at least one component fails. The decomposition thus preserves both realizability and unrealizability. Moreover, our approach does not utilize any assumptions about the other components' behavior whatsoever. Therefore, the synthesis tasks for the resulting components can be performed immediately by classical monolithic synthesis algorithms. Due to its potential of revolutionizing monolithic reactive synthesis, the developers of the synthesis tool LTLSYNT [MC18] have integrated our decomposition algorithm into their most recent release [RSDP22], which successfully competed in the synthesis competition SyntComp. However, our decomposition algorithm only identifies multiple components if the specification consists of separate parts without any dependencies between each other. To also enable the use of compositional methods for monolithic synthesis for more complex specifications, we introduce a second decomposition algorithm that again utilizes best-effort implementations. This allows for finding independent implementations in more cases. Furthermore, we base the second approach on an incremental rather than a fully compositional synthesis algorithm, allowing components to rely on concrete implementations for previously synthesized components. The technique thus combines implicit assumptions on other components, stemming from the use of best-effort implementations, with explicit assumptions on the concrete behavior of previously synthesized components, stemming from the incremental nature of the synthesis algorithm. Combining both assumption types increases the number of individual components derived by the algorithm, thus resulting in more fine-grained system decompositions.

For both distributed and monolithic systems, the algorithms and techniques introduced in this thesis automate the manual efforts that have previously been required for compositional synthesis. We, therefore, obtain fully automated compositional synthesis approaches for distributed and monolithic systems, which render the developer's manual intervention obsolete. In an experimental evaluation, we show that our compositional approaches outperform classical, non-compositional synthesis algorithms significantly.

This thesis builds upon a range of concepts, such as reactive synthesis in both its distributed and monolithic form and compositionality. We elucidate these concepts in the following sections to highlight the main contributions of this thesis in more detail afterward.

1.1. The Reactive Synthesis Problem

Synthesis is one of the pillars of formal methods. It is the task of automatically deriving a correct-by-construction implementation for a system from a formal system specification. As it eliminates the need for manual, and thus error-prone, implementation tasks, it has the potential to revolutionize the process of developing correct systems. First formulated by Alonzo Church more than 60 years ago [Chu57], synthesis never lost its fascination and is still considered to be the grand vision of formal methods.

Formally, the synthesis problem asks whether there exists an implementation that satisfies a given formal specification and, if so, derives such an implementation. A synthesized implementation is correct by construction, i.e., it inherently satisfies the specification in all possible situations. If no implementation exists, a counterexample, which prevents the existence of any realizing implementation, is derived. This allows for detecting contradictory specifications early in the design process and guides the developer in refining the specification.

In this thesis, we focus on a particular class of systems, so-called *reactive systems* [HP84]. They continually interact with their environment and run for an indefinite amount of time. The interface of a reactive system is defined by a set of inputs, which model the environment behavior, and a set of outputs, which model the system behavior. Typical examples of reactive systems are hardware circuits, embedded controllers, and communication protocols. Consequently, we consider *reactive synthesis*, which seeks implementations of reactive systems that inherently satisfy the given formal specification, in the following. In general, we distinguish between *distributed systems*, which consist of several components, so-called processes, and *monolithic systems*, which consist of a single process. While we consider both types of reactive systems in this thesis, the remainder of this section focuses on monolithic systems as the reactive synthesis problem and its early solutions have been developed with single-process systems in mind.

The reactive synthesis problem was solved independently by Büchi and Landweber [BL69], utilizing a *game-based* approach, and Rabin [Rab72], utilizing an *automata-based* approach. In this thesis, we focus on the former one. It relies on the observation that synthesis can be conceived as a game between a system player and an environment player. The system player tries to satisfy the specification, while the environment player tries to violate it. The environment is thus interpreted to be adversarial. It is not limited other than in the alphabet of the inputs. The game proceeds in rounds. In each round of the game, the environment player first produces a valuation of the system's input variables to which the system has to react. Then, the system player chooses a valuation of the output variables in response to the inputs. Playing the game round by round results in an infinite sequence of valuations of both input and output variables. If this sequence satisfies the specification, then the system player wins. Otherwise, the environment player wins. Both players can observe the history of valuations played in the previous round and may base their decisions on these. The synthesis task is then to construct a strategy for the system player that defines how to choose the valuations of output variables in each step such that, for every behavior of the environment player, the system player wins the game. Such a strategy is called *winning*. A finite representation of a winning strategy then implements the reactive system. The specification is *realizable* for the considered system if a winning strategy for the system player exists. Otherwise, it is *unrealizable*.

In this thesis, we focus on specifications given in *linear-time temporal logic (LTL)* [Pnu77], arguably one of the most standard logics for describing reactive systems. Fur such system specifications, early synthesis algorithms relying on the game-based synthesis approach employ explicit game-solving. First, the LTL specification is translated into an equivalent nondeterministic Büchi automaton [VW94]. Afterward, the automaton is determinized [Saf88] and translated into an infinite two-player game between the system and the environment. There, every state is controlled by either the environment player or the system player and represents their possible choices of valuations of input or output variables, respectively. Environment states and system states alternate. Solving the two-player game determines whether the system player has a winning strategy. Whenever such a strategy exists, the system player also has a *memoryless* winning strategy, i.e., a strategy independent of the game's history except for the last state [EJ91]. Memoryless strategies are finitely representable. Hence, if the system player wins the game, it also has a winning strategy from which an implementation of the reactive system that inherently satisfies the specification can be derived.

While the vision of synthesis is tantalizing and elegant theoretical solutions exist, the synthesis problem is infamous for being hard to solve. For LTL specifications, for instance, it is known to be 2EXPTIME-complete [PR89a]. Despite the high complexity, there has been tremendous progress toward practical solutions. Several optimizations of the automaton construction have been introduced [SB00, GO01, BKRS12]. Exploiting the structure of the specification to construct relevant parts of the game on the fly and to reuse previous inconclusive solution attempts [MSL18, LMS20] has improved the performance of explicit game-based synthesis approaches significantly. Orthogonally, the development of *safraless* decision procedures [KV05], which avoid Safra's complicated determinization procedures for automata, has given rise to *symbolic* synthesis tools [Ehl11, BBF$^+$12]. They represent the state space of the game symbolically with, for instance, bounded decision diagrams (BDDs) [Ehl12] or antichains [FJR09]. *Bounded synthesis* [FS13] further improves the safraless approaches by bounding the size of the implementation to be synthesized and by iteratively increasing the bound until an implementation that realizes the specification is found. This ensures that size-optimal implementations are synthesized. Additionally bounding the number of cycles in the implementation to be synthesized yields structurally simpler results and thus improves the understandability of the synthesized implementations [FK16, FK17]. A different line of research focused on restricting the type of specification to fragments of LTL that allow for efficient synthesis algorithms. Most successfully, the GR(1) fragment [PPS06, BJP+12, KP10], which assumes the LTL formula to be split into a set of assumptions and a set of guarantees and for which polynomial time synthesis algorithms exist, has found many applications in practice.

The extensive research on reactive synthesis has led to a broad landscape of tools for synthesis from LTL specifications (see, e.g., BoSy [FFRT17], Strix [MSL18], ltlsynt [MC18, RSDP22], and sdf-hoa [Kha21]). Since 2014, the annual reactive synthesis competition *SyntComp* [BEJ14, JBB+17b, JBB+15, JBB+16, JB16, JBB+17a, JBC+19, JPA+22], in which the participating tools compete on, today, roughly 1000 standard benchmarks, facilitated the development of synthesis tools. Furthermore, synthesis has been successfully applied to industrial systems such as the AMBA AHB bus protocol, an industrial standard for the on-chip communication of functional blocks in system-on-a-chip designs [BGJ+07, Job07, GCH13, BJP+12].

Despite these milestones, however, the reactive synthesis tools lack scalability for large and complex systems. Furthermore, the success stories in practical synthesis applications are limited to selected, rather small examples whose specifications fall into "synthesis-friendly" fragments of LTL. For more detailed introductions to the reactive synthesis problem and the history of its solutions, see, for example, [Tho09, Fin16, BCJ18].

Synthesis of Distributed Systems. Given the advances in monolithic reactive systems, it is a natural next step to consider more complex multi-process systems. *Distributed systems* consist of several processes that repeatedly interact with each other and the system's environment. The system's processes cooperate to achieve a shared goal: ensuring that the requirements for the entire system are satisfied. The cooperation might be based on limited local knowledge about the global state of the system. The processes of a distributed system and their communication interfaces, i.e., which processes can communicate with each other through which variables, are defined by an *architecture*. In particular, an architecture thus defines which system and environment variables a process can observe. In monolithic synthesis, the specification refers to the inputs and outputs of a single-process system. By definition, the outputs are controlled by the system and the inputs are observable. This results in a game with *perfect information*. In distributed synthesis, in contrast, a process might not be able to observe all global system input but only a subset of them, resulting in a game with *incomplete information* [KV00].

Distributed synthesis is a generalization of monolithic synthesis. Given a specification of the behavior of the whole system, the task is to derive a set of implementations, one for each system process, such that their parallel composition satisfies the overall system specification. The distributed synthesis problem was introduced by Pnueli and Rosner, who also showed that the problem is, in general, undecidable [PR90]. The undecidability result has been extended to specifications that fall into the syntactic safety and reachability fragments of LTL [Sch14]. However, distributed synthesis is known to be decidable for some architectures such as pipelines [PR90], chains, and one-way rings [KV01]. The architecture-specific decidability results have been generalized to a comprehensive criterion, the existence of so-called *information-forks*, that characterizes all architectures with an undecidable synthesis problem [FS05]. Intuitively, an information fork is a situation in which two processes receive information from the environment, directly or indirectly, such that they cannot completely deduce the information received by the other process. Thus, thee processes cannot be ordered according to their informedness.

Both automata-based [KV01] and game-based [MW03] synthesis algorithms have been proposed for pipeline and ring architectures. More generally, for architectures without information forks, the distributed synthesis problem can be solved by iteratively eliminating processes from the architecture in the order of growing informedness. However, the complexity of this approach is nonelementary in the number of processes [FS05]. Despite the theoretical solutions, there thus do not exist tools that are capable of automatically synthesizing implementations for the processes of a distributed system from general LTL specifications until now.

In the classical Pnueli/Rosner setting considered above, the processes of a distributed system run synchronously, i.e., all processes make their moves simultaneously. Alternatively, one can consider an asynchronous setting [PR89b], in which each process can progress at an individual rate and can wait for other processes for synchronization when needed. Commonly, processes interact through shared variables in the synchronous setting, while a *causal memory model* is considered in the asynchronous setting. In the causal memory model [GLZ04a, GLZ04b, MTY05], all processes that are involved in a synchronization via a shared event exchange their entire causal history. Therefore, the involved processes have the exact same information in the particular moment of synchronization. All other processes remain uninformed. Distributed synthesis for asynchronous systems with causal memory is often formalized with games [GGMW13] based on Zielonka's asynchronous automata [Zie87], to which we refer as *control games* in the following. In control games, actions are either controllable or uncontrollable and can thus be restricted by all or none of the involved players. The players aim at fulfilling an objective against all possible unrestricted behavior together. For the special case of acyclic architectures, distributed synthesis with control games is decidable [MW14]. However, as in the synchronous setting, the complexity is nonelementary in the number of processes. Decidability results for control games have also been obtained for restrictions on the synchronization behavior [MT02, MTY05] or on the dependencies of actions [GLZ04b], and for decomposable games [Gim17]. Recently, general undecidability has been shown for six processes [Gim22].

Petri games [FO17] are a variant of the distributed synthesis problem for asynchronous systems with causal memory, where the processes of the distributed system are tokens on a Petri net (see, e.g., [NPW81, Rei85, Old91]). The processes synchronize when they participate in joint transitions. Similar to control games, the processes share their entire causal past, including previous synchronizations, upon synchronization. The environment is also modeled with tokens, and a system process can learn about the history of an environment process when synchronizing with the corresponding environment token. The equivalence of control games and Petri games has been established, and exponential upper and lower bounds for the translation in both directions have been provided [BFH19]. For safety objectives and systems with either a single environment token and a bounded number of system tokens [FO17] or a single system token and a bounded number of environment tokens [FG17], the synthesis problem for Petri games is decidable. Further decidability and undecidability results have been obtained for Petri games with global winning conditions [FGHO22].

The concept of bounded synthesis, i.e., searching for size-optimal solutions, has also been applied to Petri games [Fin15, FGHO17]. Bounded synthesis has been extended to *true concurrency*, which allows for utilizing the concurrent nature of Petri games [HM19]. There exists an online interface for bounded synthesis for Petri games [GHY21]. In this thesis, however, we focus on the synchronous Pnueli/Rosner setting for distributed synthesis.

1.2. Compositionality

In many fields of computer science, the principle of *compositionality* has proven to be an essential technique to obtain scalability. Compositional approaches break down a complex problem into smaller subproblems that are easier to solve. The solutions for the subproblems are then combined into a solution for the entire system. Already in the 1940s, *divide-and-conquer algorithms* such as the famous *merge sort*, invented by John von Neumann (see, e.g., [Knu73]), utilized the concept of compositionality. The advantages of compositionality have been recognized, for instance, in the design of cyber-physical systems [Tri16] as well as security protocols [Cre04]. Researchers extensively study and discuss the concept of compositionality as well as its applicability to and impact on various fields of computer science until today. For instance, dedicated workshops on compositionality in computer vision [JKA+20] and artificial intelligence [MM22] exist, to name just a few.

In the area of formal methods, compositionality had a strong influence on the applicability of formal verification in practice. It has long been recognized that compositionality is the key technique that makes a "significant difference" [dRLP98] for the scalability of verification algorithms. The main idea of compositional verification is to break down the analysis of a large and complex system into multiple smaller verification tasks (see, e.g., [GNP18, dRdBH+01, CLM89]). A local task then examines a single component of the system, abstracting the rest of the system into an assumption. The key technique of compositional verification is assume-guarantee reasoning [MC81, Jon83, Pnu84], which, given a decomposition of the system into components, associates each component with an assumption on its input and a guarantee on its output. The assumptions capture the connections and interdependencies between the system components. Although identifying a suitable decomposition as well as the assumptions and guarantees by hand has proven to be challenging (see, e.g., [Lam97]), it has been a manual task for decades. More recently, algorithms for automatic system decomposition [MWW08] and assumption generation [CGP03, GPB02, NMA08, SC07, GMF08, FSB06, FPS08] for compositional verification via assume-guarantee reasoning have been developed.

There has been extensive research on compositional verification in different settings. Compositional verification techniques has been considered for finite-state hardware controllers [CLM91] but also for infinite-state systems [McM99, DGM03]. There exist algorithms for compositionally verifying, for instance, concurrent [dRdBH+01, LT91, LMM21], real-time [Hoo91, dRH89, CMP94, LL95], hybrid [Pla11, ABB16], parameterized [FMS97, BR06, NT16], and probabilistic [LS92, FKP10] systems as well as protocols [LM92, ZH95, ACG+08]. Success stories of compositional verification include model-checking a processor microarchitecture [JM01], the parameterized verification of the FLASH cache coherence protocol via compositional model checking [McM01], the verification of a communication protocol for remotely operated vehicles [GM09], and, in combination with bounded model checking [BCCZ99, Bie21], discovering bugs in widely deployed software [CDS13].

Compositional Synthesis. Inspired by the success of compositional verification, applying compositional techniques to synthesis is naturally a promising path to pursue. However, developing successful compositional algorithms has proven much more challenging in synthesis than in verification. Recall that synthesis seeks a winning strategy for the system player in the game against the system's environment. Hence, if the system behaves according to a synthesized strategy, then it satisfies the specification for every environment behavior. In compositional synthesis, we consider individual components of the system in their environment. A component's environment particularly includes, in addition to the entire system's environment, the remaining components of the system. Therefore, we aim for a winning strategy for the player that controls the component's parts of the system in the game against not only the system's environment but also the remaining system components. Consequently, we consider the other components of the system to be adversarial.

However, a winning strategy in the synthesis task of an individual system component rarely exists: such a component strategy needs to guarantee the satisfaction of *all* system requirements for the entire system – even of those that specify system behavior outside of the control of the considered component – irrespective of whether or not the other system components cooperate in the goal of satisfying the full system specification. In practice, the satisfaction of the requirements for the entire system usually cannot be guaranteed by one component alone but requires collaboration between several components. Therefore, the individual components need to rely on assumptions about the other behavior of the other system components to be able to ensure that all requirements for the system are satisfied.

Based on this observation, Chatterjee and Henzinger introduced *assume-guarantee synthesis* [CH07] in 2007. It relies on the concept of assume-guarantee contracts, which establish an agreement between the system components on their behavior. A component provides a guarantee on its own behavior and, in return, makes an assumption on the behavior of the other components of the system. A strategy for a component is then required to satisfy the specification under the hypothesis that the other components respect the established assumptions formalized in the assume-guarantee contract while not deviating from its own guarantee. If such strategies exist for all system components, and if, for each component, its guarantee implies the assumptions made by other components on the behavior of the considered component, then a solution for the whole system is found. The parallel composition of all component strategies is then guaranteed to satisfy the specification of the entire system.

The concept of assume-guarantee synthesis has ignited plenty of research on synthesis with assume-guarantee contracts. There exist several algorithms for different variants of assume-guarantee synthesis [CH07, GK13, AMT15, BCJK15, BRS17, AKRV17]. Most of them, however, rely on the user to provide the assume-guarantee contracts or require that a strategy profile, on which the components can synchronize, is constructed prior to synthesis. Therefore, extensive manual efforts are required to use most assume-guarantee synthesis algorithms. Assume-guarantee distributed synthesis [MMSZ20], in contrast, circumvents the manual efforts by utilizing the concept of environment assumptions [CH07], which was initially introduced in the context of centralized reactive synthesis. This algorithm for assume-guarantee synthesis for distributed systems negotiates the assume-guarantee contract iteratively. In each iteration, it computes minimal environment assumptions according to [CH07] for each process of the distributed system and uses these assumptions as additional constraints on the behavior of the other components. In this way, the assumptions and guarantees of the system processes are refined until a valid assume-guarantee contract is found. The negotiation procedure is not guaranteed to terminate. In this assume-guarantee approach, assumptions are restricted to safety formulas describing the concrete behavior of the other system processes.

Preventing the need for constructing explicit assumptions on the other processes' behavior, synthesis with weaker strategy requirements than winning has been considered [FKL10, KPV14, CFGR16, DF11, DF14, DFR16, BRS17, AK20, LTVZ21]. In this thesis, we focus on the notion of *remorsefree dominance* [DF11]. Instead of requiring a strategy to satisfy the given specification in every situation, remorsefree dominance only requires a strategy to satisfy the specification in situations that are realistic in the sense that they might actually occur when components that all do their best to ensure the shared goal interact. More precisely, a remorsefree dominant strategy is allowed to violate the specification as long as no other strategy would have satisfied in the same situation, i.e., for the same environment behavior. In other words, if the violation of the specification is the fault of the component's environment, we do not blame the component for preventing the fulfillment of the shared goal of satisfying the overall system requirements. Hence, remorsefree dominance is a notion of *best effort* for strategies. A remorsefree dominant strategy, intuitively, "gives its best" to satisfy the specification; however, the satisfaction of the specification is not necessarily guaranteed. This corresponds to posing *implicit assumptions* on other components, namely that they will not maliciously violate the specification.

For *safety specifications*, a specific type of formal specifications that intuitively capture that "nothing bad happens" [Lam77], it has been shown that the parallel composition of remorsefree dominant strategies is again remorsefree dominant [DF14]. This property is called the compositionality of remorsefree dominance for safety properties. This observation immediately induces a compositional synthesis algorithm [DF14] for safety specifications that synthesizes remorsefree dominant strategies for the processes of a distributed system separately and then composes them to obtain a strategy for the entire system. Since remorsefree dominance is compositional for safety properties, the resulting system strategy is guaranteed to be remorsefree dominant. If the system specification is realizable, it follows that the compositionally synthesized strategy is also winning. However, the existence of remorsefree dominant strategies for the components is not always guaranteed. Furthermore, for more general specifications, soundness of the compositional synthesis algorithm cannot be guaranteed.

Bounded dominance [DF14] is a variant of remorsefree dominance that ensures compositionality not only for safety specifications but also for *liveness specifications*, which intuitively capture that "something good eventually happens" [Lam77]. Intuitively, bounded dominance reduces every specification to a safety property by introducing a measure of the strategy's progress with respect to the specification and by bounding the number of non-progress steps, i.e., steps in which no progress is made. While bounded dominance thus induces a sound compositional synthesis algorithm for general specifications, it has two major disadvantages. First, it requires a concrete bound on the number of non-progress steps, which is often challenging to determine. Second, not every bounded dominant strategy is dominant. If the bound is chosen too small, every strategy, also a non-dominant one, is trivially bounded dominant. Hence, it cannot be guaranteed that, for realizable specifications, the parallel composition of individually synthesized bounded dominant strategies is winning. Consequently, bounded dominance is not an optimal notion for compositional distributed synthesis.

1.3. Contributions

In this thesis, we develop fully automated techniques for the compositional synthesis of reactive systems. They render the extensive manual efforts, which have been required for utilizing compositional concepts in synthesis so far, unnecessary and circumvent the disadvantages of existing approaches. In the first part of this thesis, we focus on distributed systems and introduce algorithms for automatically deriving implicit and explicit assumptions on process behavior. In the second part of this thesis, we transfer the concept of compositional synthesis from distributed systems to monolithic systems and present suitable decomposition algorithms. We introduce approaches based on winning and best-effort strategies for both types of systems.

Implicit Assumptions for Distributed Synthesis. We present a new requirement for system strategies, called *delay-dominance*, that formalizes a notion of best effort while circumventing the weaknesses of both remorsefree dominance and bounded dominance. It introduces a progress measure on strategies with respect to a specification given as an alternating co-Büchi automaton. Based on this measure, delay-dominance then relates non-progress steps in a delay-dominant strategy to non-progress steps in an alternative strategy. We show that every delay-dominant strategy is also remorsefree dominant, resulting in the crucial property that, for realizable specifications, every delay-dominant strategy is winning. Furthermore, we introduce a criterion for specifications given as alternating co-Büchi automata such that compositionality of delay-dominance is guaranteed if the criterion is satisfied. We present a three-step construction of a universal co-Büchi automaton from an LTL formula that recognizes delay-dominant strategies. We show that the resulting automaton is of single-exponential size in the squared length of the LTL formula and can be used immediately for safraless synthesis approaches [KPV06] to synthesize delay-dominant strategies, yielding the result that synthesis of delay-dominant strategies is in 2EXPTIME. Based on delay-dominance, we introduce a compositional synthesis algorithm for distributed systems that utilizes implicit assumptions on the behavior of other system processes.

Explicit Assumptions for Distributed Synthesis. We introduce a compositional synthesis algorithm, called *certifying synthesis*, that, in addition to the strategies for the system processes, automatically derives guarantees, so-called *certificates*, on the behavior of every process. The certificates of the system processes constitute an assume-guarantee contract, thus providing essential information to the synthesis tasks for the individual processes. Our algorithm is an extension of bounded synthesis for monolithic systems [FS13] that incorporates the additional search for certificates into the synthesis task for the individual process strategies. We introduce two representations of certificates, as LTL formulas and finite-state machines. We prove the soundness and completeness of our synthesis algorithm for both of them. Furthermore, we present an approach for determining which processes are relevant for the considered one in the sense that assumptions on their behavior are required for a successful synthesis task. In this way, the number of considered certificates is reduced for each system process while soundness and completeness of certifying synthesis are preserved. Focusing on certificates represented by finite-state machines, both deterministic and nondeterministic ones, we reduce certifying synthesis to a SAT constraint-solving problem. We have developed a prototype of certifying synthesis and compared it to non-compositional distributed synthesis algorithms, showcasing the significant advantage of certifying synthesis for larger systems.

Assumption-free Decomposition for Monolithic Synthesis. We present an approach for decomposing a monolithic system into independent components. It is based on identifying independent parts of the system specification, which then define the components and their requirements[1]. Given a realizable specification for the system, winning strategies for the individual components can be synthesized independently. For an unrealizable system specification, the synthesis task of at least one component is unrealizable as well. Hence, the decomposition preserves both realizability and unrealizability of the monolithic synthesis task. The component synthesis tasks are classical monolithic synthesis task. We establish a sound and complete language-based criterion for determining whether two subspecifications are independent. We lift the independence criterion to temporal logics by introducing an approximate independence criterion for LTL formulas. We develop a decomposition algorithm for LTL formulas, which is based on a syntactic dependency analysis of the formula according to the independence criterion. We present two optimizations of the decomposition algorithm for formulas in assume-guarantee form, which identifies assumptions that can be dropped for the considered set of guarantees while preserving realizability and unrealizability. We have developed a prototype of our LTL decomposition algorithm and have evaluated it on top of state-of-the-art synthesis tools. The decomposition is nearly instantaneous and the synthesis time is reduced significantly if multiple components have been derived. The decomposition algorithm can thus be seen as a preprocessing technique for reactive synthesis algorithms and has already been integrated into the newest release [RSDP22] of the synthesis tool ltlsynt [MC18] by its developers. Furthermore, we illustrate the applicability of our specification decomposition algorithm for compositional synthesis to the domain of smart contracts.

Assumption-based Decomposition for Monolithic Synthesis. We introduce an incremental synthesis approach for monolithic systems based on remorsefree dominant strategies and the computation of a suitable system decomposition. In incremental synthesis, for every component, a remorsefree dominant strategy is synthesized *under the assumption* that the components that

[1]Decomposition algorithms for specifications have also been studied as part of Gideon Geier's Bachelor's thesis at Saarland University in 2020 [Gei20], which the author of this thesis supervised.

have been considered previously do not deviate from their already synthesized strategies. This allows for making both implicit assumptions, stemming from the use of remorsefree dominant strategies, and explicit assumptions, stemming from the incremental nature of the synthesis algorithm, about the other processes' behavior. We propose two techniques for identifying the components of the system as well as the order in which they are synthesized. The approaches offer different trade-offs between precision and computational cost. The first decomposition method is based on a semantic dependency analysis of the output variables of the system. The second one relies on a syntactic analysis of the structure of the specification. Both the semantic and syntactic decomposition approaches ensure the soundness and completeness of incremental synthesis. Furthermore, we present rules for simplifying the specification for the individual components by omitting irrelevant conjuncts while preserving realizability and unrealizability of the synthesis tasks. We have developed a prototype of the incremental synthesis algorithm and compared it to classical monolithic synthesis tools, demonstrating the advantage of incremental synthesis for larger but decomposable system.

1.4. Publications

This thesis is based on the following peer-reviewed publications:

[FP20a] Bernd Finkbeiner and Noemi Passing. Dependency-based compositional synthesis. In *Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Proceedings*, Vol. 12302 of *Lecture Notes in Computer Science*, pp. 447–463. Springer, 2020. doi: 10.1007/978-3-030-59152-6_25

[FGP21a] Bernd Finkbeiner, Gideon Geier, and Noemi Passing. Specification decomposition for reactive synthesis. In *NASA Formal Methods - 13th International Symposium, NFM 2021, Proceedings*, Vol. 12673 of *Lecture Notes in Computer Science*, pp. 113–130. Springer, 2021. doi: 10.1007/978-3-030-76384-8_8

[FP21a] Bernd Finkbeiner and Noemi Passing. Compositional synthesis of modular systems. In *Automated Technology for Verification and Analysis - 19th International Symposium, ATVA 2021, Proceedings*, Vol. 12971 of *Lecture Notes in Computer Science*, pp. 303–319. Springer, 2021. doi: 10.1007/978-3-030-88885-5_20

[FGP22] Bernd Finkbeiner, Gideon Geier, and Noemi Passing. Specification decomposition for reactive synthesis. *Innov. Syst. Softw. Eng.*, 2022. doi: 10.1007/s11334-022-00462-6

[FP22a] Bernd Finkbeiner and Noemi Passing. Compositional synthesis of modular systems. *Innov. Syst. Softw. Eng.*, 18(3):455–469, 2022. doi: 10.1007/s11334-022-00450-w

[FP22b] Bernd Finkbeiner and Noemi Passing. Synthesizing dominant strategies for liveness. In *42nd IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2022, Proceedings*, Vol. 250 of *LIPIcs*, pp. 37:1–37:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi: 10.4230/LIPIcs.FSTTCS.2022.37

[FHKP23] Bernd Finkbeiner, Jana Hofmann, Florian Kohn, and Noemi Passing. Reactive synthesis of smart contract control flows. In *Automated Technology for Verification and Analysis - 21st International Symposium, ATVA 2023, Proceedings*, 2023. (To appear)

Furthermore, this thesis contains material published in the following technical reports:

[FP20b] Bernd Finkbeiner and Noemi Passing. Dependency-based compositional synthesis (full version). 2020, arXiv: 2007.06941

[FGP21b] Bernd Finkbeiner, Gideon Geier, and Noemi Passing. Specification decomposition for reactive synthesis (full version). 2021, arXiv: 2103.08459

[FP21b] Bernd Finkbeiner and Noemi Passing. Compositional synthesis of modular systems (full version). 2021, arXiv: 2106.14783

[FHKP22] Bernd Finkbeiner, Jana Hofmann, Florian Kohn, and Noemi Passing. Reactive synthesis of smart contract control flows. 2022, arXiv: 2205.06039

[FP22c] Bernd Finkbeiner and Noemi Passing. Synthesizing dominant strategies for liveness (full version). 2022, arXiv: 2210.01660

1.5. Related Work

In this section, we discuss work closely related to the approaches introduced in this thesis. First, we consider compositional algorithms for the synthesis of distributed systems, particularly focusing on approaches that fall into the class of assume-guarantee synthesis as well as approaches that rely on best-effort strategies or environment assumptions. Afterward, we discuss compositional synthesis algorithms for monolithic systems and the use of specification decomposition in monolithic reactive synthesis tools.

Compositional Distributed Synthesis. There are several compositional approaches for the synthesis of distributed systems. In this paragraph, we only focus on those that do not fall into the class of *assume-guarantee synthesis* and that neither rely on *environment assumptions* nor *best-effort strategies*. We address the other algorithms in separate paragraphs.

Finkbeiner et al. introduce a compositional synthesis algorithm based on information-flow assumptions between the processes of a distributed system [FMM22]. Information-flow assumptions are hyperproperties [CS10], i.e., properties that relate multiple execution traces, that describe differences in the behavior of a system process that specific other system processes can observe. In contrast to the assumptions considered in this thesis and the assume-guarantee style algorithms by Majumdar et al. [MMSZ20] and Alur et al. [AMT15], which we address in the subsequent paragraph, information-flow assumptions are not behavioral, i.e., they do not restrict the behavior of other processes, but capture the information that a process can deduce from other processes. Computing information-flow assumptions for compositional synthesis can thus be seen as an orthogonal approach to deriving behavioral assumptions.

Semi-automatic distributed synthesis [SF07] is a compositional synthesis approach for distributed systems that heavily relies on the assistance of the developer. The authors show that it is possible to decompose a realizable specification into a conjunction of local properties that the individual processes of the distributed system can guarantee. The synthesis of a strategy for an individual system process can thus be done automatically once local properties are derived. The composition of the separately synthesized strategies then serves as a strategy for the entire system and is guaranteed to satisfy the full system specification. Strengthening the specification into a conjunction of such local specifications, however, remains a manual task.

In the setting of controller synthesis, Alur et al. propose a compositional synthesis algorithm for dynamically-decoupled multi-agent systems [AMT18]. Assuming that the specification is given in a conjunctive form, they exploit the observation that conjuncts usually only concern a small subset of agents. For each conjunct, a maximally permissive strategy is synthesized for the agents involved in the conjunct. Such a strategy does not unnecessarily fix a particular agent behavior. The resulting strategies for all conjuncts are intersected to identify potential conflicts. For conflict resolution, constraints on local subproblems, which must be satisfied to avoid conflict, are identified. These constraints are provided to the respective subproblems, and synthesis of a maximally permissive strategy is performed again with the updated objective. This process is repeated until a fixpoint in the strategies is reached.

Assume-Guarantee Synthesis. Chatterjee and Henzinger introduced the concept of assume-guarantee synthesis first [CH07]. It considers the synthesis of two-process systems with individual specifications such that the parallel composition of the process strategies realizes the conjunction of the specifications. In Chatterjee and Henzinger's formulation, the processes can be considered conditionally competitive, as they primarily try to realize their own objective and will only secondarily try to violate the other processes' objective. Each process assumes that the other process does not violate its own specification. The synthesis problem is then solved by computing a secure equilibrium [CHJ06], which ensures that the strategies for the processes realize their objectives and that differing from this strategy to violate the other process's objective can be penalized by the latter process violating the objective of the former process. Assume-guarantee synthesis has, among others, been extended to the setting of concurrent reactive programs with partial information [BCJK15], i.e., where variables can be local to a process and thus non-observable for other processes, and to a quantitive setting [AKRV17], in which the specification formalism is multi-valued, and the goal is to generate a system that maximizes the satisfaction value of the specification.

In a similar line of research, Brenguier et al. introduce assume-admissible synthesis [BRS17] based on the notion of admissible strategies [Ber07, Fae09, BRS14, BPRS17]. Admissibility is a common concept of best effort for strategies, which we further discuss in the respective paragraph on *best-effort strategies*. In assume-admissible synthesis, each process of the considered two-process system assumes that the other process plays an admissible strategy. Based on this assumption, the synthesis algorithm derives, for each process, an admissible strategy that is winning for its objective against all admissible strategies of the other process. Both assume-guarantee synthesis and assume-admissible synthesis are sound in the sense that the composition of synthesized strategies for the individual system processes is guaranteed to satisfy the conjunction of all system requirements.

Outside the setting of assume-guarantee synthesis, Chatterjee et al. present an algorithm for computing minimally restrictive assumptions on the environment behavior to obtain realizability of a given unrealizable specification [CHJ08]. In a similar direction, Alur et al. introduce a pattern-based refinement algorithm for unrealizable LTL formula in the GR(1) fragment [PPS06, BJP$^+$12] by adding assumptions on the behavior of the environment [AMT13]. Both algorithms have been used for assume-guarantee-style synthesis algorithms [MMSZ20, AMT15]. Majumdar et al. [MMSZ20] introduce assume-guarantee distributed synthesis based on minimally restrictive environment assumptions [CHJ08]. They synthesize assume-guarantee contracts using a negotiation algorithm. In each round of the negotiation, minimal assumptions are constructed for each process and then added as additional constraints to the synthesis tasks of the other processes. The assumptions and guarantees are refined iteratively until a valid assume-guarantee contract is found. The negotiation procedure is not guaranteed to terminate. The synthesis algorithm only considers assume-guarantee contracts consisting of safety assumptions and guarantees. The approach is implemented in the tool Agnes [Mal20], which currently only supports safety and deterministic Büchi objectives.

Alur et al. utilize their previous work [AMT13] on refining unrealizable GR(1) formulas for assume-guarantee-style compositional synthesis [AMT15]. The approach is only applicable to two-process systems with local specifications and an architecture with a serial connection of the processes, such as pipelines. Hence, for one process of the system, its local specification needs to be realizable irrespective of the behavior of the other process. The core of the synthesis algorithm is to refine the local process specifications by generating assumptions and guarantees for the two processes of the system to obtain realizability of the local objectives. The authors propose three different algorithms for using pattern-based refinement in the context of compositional synthesis based on the amount of information about the strategies of the process with realizable local objectives shared between the processes.

Greenyer and Kindler propose an assume-guarantee-style synthesis algorithm for monolithic systems [GK13]. As this paragraph focuses on distributed systems, we discuss their approach in more detail in paragraph *compositional monolithic systems*.

Synthesis of Best-Effort Strategies. There are several notions of best effort in strategies. Rationality is a requirement that is often posed on strategies [FKL10, KPV14, CFGR16]. Rational agents are assumed to satisfy their own local objectives. Fisman et al. introduce rational synthesis, where the system is assumed to be monolithic but where the environment consists of several partially controllable components, which are assumed to be rational agents [FKL10]. Rational synthesis then derives a strategy for the considered monolithic system and a profile of strategies that suggests a behavior of the environment components. The composition of the system strategy and the strategy profile is required to satisfy the system's objective. Furthermore, the strategy profile should be an equilibrium in the sense that the environment agents do not have an incentive to deviate from the strategy profile. The authors propose solutions to the rational synthesis problem for three common notions in algorithmic game theory, Nash equilibria, dominating strategies, and subgame-perfect Nash equilibria (see, e.g., [NRTV07]). Kupferman et al. extend rational synthesis as introduced by Fisman et al., in which agents are rational and cooperative, to the rational but non-cooperative setting [KPV14]. They show 2EXPTIME-completeness for rational synthesis in both the cooperative and the non-cooperative case. Furthermore, they extend the approach to quantitive system objectives. Condurache et al. study the complexity of rational synthesis in both the cooperative and the non-cooperative setting in more detail [CFGR16]. They provide tight complexity results for different kinds of system objectives – safety, reachability, Büchi, co-Büchi, parity, Streett, Rabin, and Muller as well as full LTL – and for both fixed and unfixed numbers of players.

The notion of admissibility of strategies [Ber07, Fae09, BRS14, BPRS17] is based on the classical game theoretical concept of weakly dominated strategies. Intuitively, a strategy weakly dominates another strategy if it performs as least as good as the dominated strategy in all situations and if there exists a situation in which it performs strictly better than the dominated strategy (see, e.g., [NRTV07]). A strategy is admissible if it is not weakly dominated by any other strategy. Berwanger lifts the admissibility notion to games played on graphs and provides existence results for admissible strategies in infinite multi-player games [Ber07]. Faella studies admissible strategies in infinite two-player games and their required memory [Fae09]. He shows that admissible strategies may require an unbounded amount of memory even for objectives for which memoryless strategies exist and introduces necessary and sufficient conditions for objectives to have memoryless admissible strategies. Furthermore, he presents an efficient way of computing admissible strategies from winning and cooperative strategies. Brenguier et al. build upon Berwanger's results [Ber07] and study the complexity of iterated elimination of dominated strategies in different types of ω-regular games [BRS14]. Additionally, they present the construction of an ω-automaton that recognizes all possible outcomes of admissible strategies, i.e., those strategies that survive the iterated elimination of dominated strategies. The automaton construction enables, for instance, solving the model-checking under admissibility problem. Assume-admissible synthesis [BRS17] is a compositional synthesis algorithm based on synthesizing admissible strategies for the individual processes, which we discuss in the paragraph on *assume-guarantee synthesis*.

Similar to admissibility, remorsefree dominance, which has first been introduced for reactive synthesis by Damm and Finkbeiner [DF11], relies on the notion of dominating strategies. Domination in remorsefree dominance, however, is defined in a slightly different manner than weak domination in admissibility: while it also requires a dominating strategy to perform at least as good as the dominated one, it does not require the former strategy to perform strictly better in some situation. A strategy is remorsefree dominant if it dominates all other strategies. Remorsefree dominance is thus strictly stronger notion than admissibility, i.e., every remorsefree dominant strategy is admissible while not every admissible strategy is remorsefree dominant [BRS17]. For realizable specifications, remorsefree dominant strategies are guaranteed to be winning. Remorsefree dominant strategies have been utilized for the compositional synthesis of distributed systems for safety specifications by computing remorsefree dominant strategies for the system processes separately [DF14]. This synthesis approach is restricted to safety specifications as, for liveness specifications, it is not guaranteed that the composition of two remorsefree dominant strategies is again remorsefree dominant.

Bounded dominance [DF14] is a variant of remorsefree dominance that addresses this problem. Intuitively, bounded dominance reduces a liveness specification to a safety property. It utilizes a progress measure on strategies and introduces a bound on the number of steps in which a strategy does not make progress with respect to the specification. While it is guaranteed that the composition of two bounded dominant strategies is again bounded dominant, it requires a concrete bound on the number of non-progress steps. Furthermore, a bounded dominant strategy is not necessarily remorsefree dominant. If the bound is chosen too small, every strategy, even one that unnecessarily violates the specification, is bounded dominant. Therefore, in contrast do remorsefree dominance, bounded dominant strategies are even for realizable specifications not necessarily winning.

Damm et al. generalize the compositional synthesis algorithm based on remorsefree dominant strategies [DF14] to settings where remorsefree dominant strategies only exist under certain assumptions about the future behaviors of other system processes [DFR16]. They propose an incremental synthesis algorithm based on automatically constructing such assumptions. The approach is only applicable for systems in which the processes can be ordered by their criticality. Less critical processes are then required to change their behavior to guarantee that the assumptions needed by more critical processes are satisfied.

Good-enough strategies [AK20] are similar to remorsefree dominant strategies [DF11] and only require a strategy to satisfy the specification for input sequences for which there exists an output sequence that satisfies the specification, i.e., only in situations in which the specification can be satisfied. The concept of good-enough strategies has been extended to a multi-valued correctness notion, allowing for specifying system quality [AK20]. Furthermore, Li et al. [LTVZ21] study good-enough strategies for LTL specifications over finite-traces [GV13], called LTL_f specifications. They propose two synthesis algorithms for good-enough strategies for system specifications given as LTL_f formulas, one via good-enough strategies for LTL specifications and one via a reduction to solving games played on deterministic Büchi automata.

Furthermore, the synthesis of best-effort strategies has been considered in the setting, where additional LTL assumptions on the environment behavior are provided to the synthesis task [AGL+20, AGR21]. Due to the existence of environment assumptions in the approach, we discuss it in the subsequent paragraph.

Synthesis under Environment Assumptions. Similar to the best-effort notions for strategies discussed in the previous paragraph, synthesis under environment assumptions aims at relaxing the requirements on a system strategy. In contrast to best-effort strategies, explicit assumptions on the environment are added to the synthesis task. We discuss different flavors of such assumptions restricting the environment behavior.

The approach of Chatterjee et al. for synthesizing minimally restrictive environment assumptions [CHJ08] and the pattern-based refinement of GR(1) formulas [AMT13], both already mentioned in the paragraph on *assume-guarantee synthesis*, construct LTL formulas that restrict the possible environment behavior. These formals can then be used as assumptions in the synthesis task to limit the possible input sequences. Similarly, Li et al. mine assumptions for synthesis from counterexamples obtained during synthesis for unrealizable specifications [LDS11].

Similar to [AMT13], they use template-based assumptions. To the best of our knowledge, assumption mining, as introduced in [LDS11], has not been used for compositional synthesis. The challenges of synthesis under environment assumptions formulated as LTL formulas are discussed in [BEJK14]. The authors propose four goals that should be met when considering synthesis under environment assumptions, casually formulated as “Be Correct!”, “Don’t Be Lazy!”, “Never Give Up!”, and “Cooperate!”.

Aminof et al. propose to see environment assumptions as non-empty sets of strategies instead of sets of traces [AGMR18]. Furthermore, they define the synthesis problem for LTL specifications under environment assumptions represented by sets of environment strategies. This concept is utilized in [AGL$^+$20], where the environment is formalizes by two distinct models and thus two environment assumptions, one capturing expected behavior and one that also includes exceptional behavior. The environment assumptions are again formalized as LTL formulas. A strategy is then required to win against the expected environment behaviors, and, in addition, it should try to satisfy the exceptional behaviors as far as possible. Note that this again resembles a notion of best effort, yet, in a different setting. The authors show that if a winning strategy exists against the expected environment behavior, then there is also one that additionally makes the best effort against the exceptional ones. Aminof et al. then show that computing such a strategy that is winning against expected environment behavior and a best-effort strategy against exceptional environment behavior is 2EXPTIME-complete and thus not harder than classical synthesis [AGR21].

Instead of formalizing explicit environment behavior, assumptions on the environment can also be conceptual such as assuming the environment to behave rational [FKL10, KPV14, CFGR16]) or admissible [Ber07, Fae09, BRS14, BPRS17]. We refer to the previous paragraph for more information on rational and admissible strategies and their synthesis problems.

Compositional Monolithic Synthesis. Compositional approaches to monolithic synthesis have been studied from different angles. Kupferman et al. introduce a compositional synthesis algorithm, extending their safraless synthesis algorithm [KV05], that is designed for incrementally adding requirements to a specification during system design [KPV06]. They reduce the LTL realizability problem to an emptiness problem of a nondeterministic Büchi tree automaton. Given a specification consisting of several conjuncts, their approach first checks realizability for the individual conjuncts. Then it reuses results from isolated realizability checks to reduce the state space of the automaton for the full specification. As their approach was developed with the incremental refinement of specifications in mind, it is only applicable to specifications in conjunctive form, i.e., conjuncts of LTL subspecifications.

Filiot et al. introduce a compositional algorithm to solve the LTL realizability and synthesis problems [FJR10, FJR11]. It relies on the author’s previous results that the LTL realizability problem can be reduced to solving a safety game [FJR09]. The authors show that the safety game for the realizability of an LTL formula in conjunctive form can be solved by solving safety games for the conjuncts of the formula independently. For each subspecification, a separate safety game is constructed, and a so-called master plan is computed. A master plan subsumes all winning strategies. The algorithm then composes the master plans for the subspecifications, resulting in a master plan for the full specification from which an implementation can then be extracted. For LTL formulas that are not in conjunctive form but consist of a set of assumptions and a set of guarantees, the formula is translated into conjunctive form. Hence, we obtain several conjuncts, one for each guarantee, which all contain all assumptions, resulting in an enormous blow-up of the specification length. The authors propose a simple yet incomplete heuristic for eliminating unnecessary assumptions for the individual subspecifications. Realizability of the subspecifications might get lost when eliminating assumptions according to the heuristic. The main algorithm for LTL formulas in conjunctive form has been implemented in the tool Acacia+ [BBF+12], which is unfortunately not available anymore.

Kulkarni and Fu present a compositional distributed synthesis approach for LTL formulas in conjunctive form [KF18]. The algorithm is restricted to LTL formulas that can be expressed with deterministic finite-state automata, immediately enabling the use of safety games for synthesis. The approach is thus not applicable to liveness properties. Similar to the compositional synthesis algorithm by Filiot et al. [FJR10, FJR11], Kulkarni and Fu make use of the conjunctive nature of many LTL specifications and propose to systematically construct the winning regions of the safety games for the individual conjuncts separately. Furthermore, they introduce a method to compute the winning region of the safety game for the synthesis task of the entire system by combining the winning regions of the individual processes.

In a similar direction, Bansal et al. propose a compositional synthesis approach for specifications in Safety LTL [CMP92], a syntactic fragment of LTL that only allows safety properties, in conjunctive form that synthesizes a strategy for each conjunct separately and then composes them one by one [BGS+22]. The authors show that, for Safety LTL formulas, it suffices to consider a partial game arena instead of the exact one to ensure the satisfaction of the formula when building the conjunction with other formulas. This reduces the state space for subsequent operations. The algorithm derives a deterministic safety automaton for each Safety LTL conjunct separately. The authors propose two variants of splitting each automaton into a winning part and a losing part, allowing for reducing the size of the automaton by clustering the losing part into a single state. Lastly, the resulting automata are composed iteratively.

Independent of our specification decomposition algorithm for the compositional synthesis of monolithic systems [FGP21a, FGP22], which is presented in Chapter 5, Mavridou et al. [MKG+21] introduced a compositional realizability analysis in FRET [GPM+20, NAS20], a publicly available tool for writing, understanding, formalizing, and analyzing requirements by NASA's Ames Research Center. Their approach is based on similar ideas as our LTL decomposition algorithm, i.e., on identifying independent parts of the requirements by computing dependencies between individual requirements, building a dependency graph, and computing the strongly connected components. As specifications are written in FRETish [GPM+20], the requirement language of FRET, in their setting, however, the dependency analysis differs. The optimized handling of assumptions in specifications that consist of sets of assumptions and guarantees of our LTL decomposition technique cannot be easily integrated into their approach for FRET. For a more detailed comparison of both approaches, we refer to [MKG+21].

For specifications given as Live Sequence Charts [DH01], an expressive specification format that distinguishes between behaviors that may happen and that must happen, Kugler and Segall present two compositional synthesis approaches [KS09]. The first algorithm implements the sound composition of two synthesized strategies for subspecifications of the system specification. It is not complete, as it requires the existence of individual strategies for the subspecifications, regardless of any interconnections. The second approach computes an overapproximation of the maximal winning strategy, a so-called optimistic strategy, for each subspecification and then utilizes the first algorithm for composing them. Optimistic strategies follow a similar idea as master plans in [FJR10], yet, they may violate liveness constraints while master plans do not. The resulting composed strategy is again an optimistic strategy. However, it might not be a valid strategy since the choices of the system and environment are not guaranteed to alternate. In particular, the strategy might rely on entering an infinite loop of system events. While the authors briefly describe a sound and complete extension of their algorithms, they neither formalize nor implement it.

In the context of controller synthesis, Greenyer and Kindler propose a compositional monolithic synthesis algorithm from specifications given as Modal Sequence Charts [HM08] based on assume-guarantee contracts [GK13]. The approach heavily relies on manual interventions. In particular, it requires the developer to identify a suitable decomposition of the system into component and an assume-guarantee contract consisting of small enough properties so that compositional synthesis has an advantage over classical monolithic synthesis. In the same setting, Baier et al. present an algorithm for incrementally synthesizing most general controllers for LTL specifications in conjunctive form [BKK11]. The already synthesized most-general controllers are provided for the later synthesis tasks. The authors only show the existence of most general controllers for the individual synthesis tasks for safety and co-safety objectives.

Specification Decomposition in Reactive Synthesis Tools. Several reactive synthesis tools for monolithic systems decompose the given system specification into subspecifications. The game-based tool Strix [MSL18] uses decomposition to identify suitable automaton types for internal representation. Furthermore, it recognizes isomorphic parts of the specification to avoid redundant synthesis tasks. The synthesis tools Unbeast [Ehl11] and Safety-First [SS13], in contrast, analyze the specification to identify safety subspecifications, which can be synthesized more efficiently. All three tools do not perform fully independent synthesis tasks for the derived subspecifications. Therefore, they do not implement compositional monolithic synthesis approaches. As outlined in the previous paragraph, Acacia+ [BBF+12] implements the compositional synthesis approach by Filiot et al. [FJR10, FJR11], yet, it is not available anymore. The developers of the synthesis tool ltlsynt [MC18] included our specification decomposition approach for monolithic synthesis [FGP21a, FGP22], presented in Chapter 5, as an optimization in their most recent release [RSDP22].

1.6. Structure of This Thesis

This thesis is structured into two parts. The former presents compositional synthesis algorithms for distributed systems, utilizing both implicit and explicit assumptions on the behavior of other system processes. The latter introduces decomposition techniques for monolithic systems, thus enabling compositional monolithic synthesis. As the problems considered in both parts have the same origin, the required foundations coincide. Therefore, they are jointly presented in Chapter 2. We particularly focus on system architectures and specifications as well as the representation of system strategies and implementations. Furthermore, we formalize both the monolithic and distributed reactive synthesis problem and present synthesis approaches for both winning and remorsefree dominant strategies. Both parts of this thesis are sufficiently self-contained to be read independently. We conclude the thesis in Chapter 7 with a discussion of the results of this thesis and open problems.

Part I: Distributed Systems. In Chapter 3, we present the compositional distributed synthesis algorithm based on implicit assumptions. First, we present existing approaches to compositional synthesis that utilize variants of remorsefree dominance as a best-effort notion for strategies and discuss their unsuitability for liveness specifications. Afterward, we introduce delay-dominance as a new strategy requirement with a game-based definition and show that every delay-dominant strategy is also remorsefree dominant. We establish a criterion for alternating co-Büchi automata and prove that, if the criterion is satisfied, compositionality of delay-dominance is guaranteed. Then, we introduce a three-step construction of a universal co-Büchi automaton that can be immediately used for synthesizing delay-dominant strategies with safraless synthesis algorithms. Lastly, we present a compositional synthesis approach for distributed systems based on synthesizing separate delay-dominant strategies for the system's processes.

In Chapter 4, we introduce a compositional synthesis algorithm for distributed systems that automatically derives guarantees on the behavior of the processes, so-called certificates, which constitute an assume-guarantee contract. We introduce a running example, which we will use throughout the chapter. Afterward, we introduce the main concept of compositional synthesis with certificates, focusing on certificates formalized with LTL formulas, and prove its soundness and completeness. Next, we establish how certificates can be modeled with deterministic finite-state machines and show soundness and completeness of the resulting synthesis algorithm. Then, we present a reduction of compositional synthesis with certificates represented by deterministic finite-state machines to a SAT constraint-solving problem. We introduce two optimizations of the synthesis algorithm. First, we present a criterion for identifying which certificates are relevant for a process and prove that soundness and completeness are preserved when only considering relevant certificates. Second, we permit nondeterminism in certificates represented by finite-state machines. We show soundness and completeness of the resulting approach and discuss the necessary changes in the SAT encoding. Lastly, we present an experimental evaluation of our approach.

Part II: Monolithic Systems. In Chapter 5, we present a decomposition algorithm for monolithic systems that ensures, given a realizable system specification, realizability of the resulting synthesis subtasks for all derived components. For an unrealizable system specification, unrealizability of the synthesis subtasks of at least one component is guaranteed. First, we introduce the concept of modular monolithic synthesis. Afterward, we establish a language-based independence criterion for subspecifications and prove that soundness and completeness of modular monolithic synthesis are guaranteed for decompositions satisfying the criterion.

Next, we lift the language-based criterion to the temporal logic level by introducing a syntactic independence criterion for LTL specifications that approximates the language-based criterion in the sense that it might yield coarser decompositions than necessary. We prove that, nevertheless, soundness and completeness of modular synthesis are guaranteed. We present an algorithm for identifying system components based on the LTL independence criterion. Afterward, we introduce optimizations of the algorithm for specifications in both strict and non-strict assume-guarantee forms. We present an experimental evaluation of our approach, utilizing it as a preprocessing technique for state-of-the-art synthesis tools. Lastly, we present the applicability of our decomposition algorithm to smart contract specifications.

In Chapter 6, we introduce an incremental synthesis algorithm for distributed systems based on the synthesis of remorsefree dominant strategies. After presenting a running example, which we use throughout the chapter, we introduce the incremental synthesis algorithm. Next, we present the concept of semantic dependencies between output variables and prove that the absence of such dependencies guarantees the success of the individual synthesis tasks in incremental synthesis. Consequently, we introduce a decomposition algorithm based on semantic dependencies, ensuring soundness and completeness of incremental synthesis, and an optimization, which allows for even more fine-grained decompositions, next. Afterward, we present the concept of syntactic dependencies, which conservatively overapproximate semantic dependencies, and a corresponding syntactic decomposition algorithm. We show that the absence of syntactic dependencies again ensures the success of the individual synthesis tasks in incremental synthesis and that hence soundness and completeness of incremental synthesis are preserved. Next, we introduce rules for simplifying the specifications for the individual synthesis tasks for the components by omitting conjuncts that do not affect the resulting strategies or their existence. Lastly, we present an experimental evaluation of our approach.

Chapter 2

Foundations

In this section, we lay the foundations and fix the notations for the remainder of this thesis. We introduce the type of systems that we consider as well as concepts for specifying requirements on them. We present formalisms for modeling system strategies and introduce reactive synthesis as a mechanism to derive such strategies from a system specification automatically.

2.1. Notation

Given an alphabet Σ, we denote the *set of infinite words* over 2^Σ with $(2^\Sigma)^\omega$ and the *set of finite words* over 2^Σ with $(2^\Sigma)^*$. We define $(2^\Sigma)^\infty = (2^\Sigma)^* \cup (2^\Sigma)^\omega$ to be the set of finite and infinite words over 2^Σ. The *length* of a finite word $\sigma \in (2^\Sigma)^*$ is denoted with $|\sigma|$. The length of an infinite word $\sigma \in (2^\Sigma)^\omega$ is ∞.

For a finite or infinite word $\sigma \in (2^\Sigma)^\infty$ and $k \in \mathbb{N}_0$ with $k \leq |\sigma|$, we denote the symbol of σ at point in time k with σ_k. Note that the first symbol of σ is σ_0 and thus σ_{k-1} denotes the k-th symbol of σ. Given a word $\sigma \in (2^\Sigma)^\infty$ and $k \in \mathbb{N}_0$ with $k \leq |\sigma|$, the *prefix* of length k of σ is denoted with $\sigma_{|k} = \sigma_0 \dots \sigma_{k-1}$. We denote the set of all prefixes of σ of arbitrary length with $\mathrm{Pref}(\sigma)$. Given words $\sigma \in \Sigma^*$ and $\sigma' \in \Sigma^\infty$ with $|\sigma'| > 0$, the *concatenation* of σ and σ' is defined by $\sigma \cdot \sigma' = \sigma_0 \dots \sigma_{|\sigma|-1}\sigma'_0 \dots \sigma'_{|\sigma'|-1}$. If $\sigma' \in \Sigma^*$ holds, then we have $\sigma \cdot \sigma' \in \Sigma^*$ with $|\sigma \cdot \sigma'| = |\sigma| + |\sigma'|$. Otherwise, $\sigma \cdot \sigma' \in \Sigma^\omega$ holds. Irrespective of the alphabet, we represent the *empty word* with ε and $\varepsilon \cdot \sigma = \sigma$ holds for all $\sigma \in (2^\Sigma)^\infty$.

For a word $\sigma \in (2^\Sigma)^\infty$ and a set $X \subseteq \Sigma$, we define $\sigma \cap X = (\sigma_0 \cap X)(\sigma_1 \cap X) \dots (\sigma_{|\sigma|-1} \cap X)$. We have $\sigma \cap X \in (2^X)^\infty$. For two words $\sigma \in (2^\Sigma)^\infty$ and $\sigma' \in (2^{\Sigma'})^\infty$ with $\Sigma \cap \Sigma' = \emptyset$ and $|\sigma| = |\sigma'|$, we define $\sigma \cup \sigma' = (\sigma_0 \cup \sigma'_0)(\sigma_1 \cup \sigma'_1) \dots (\sigma_{|\sigma|-1} \cup \sigma'_{|\sigma|-1})$. We have $\sigma \cup \sigma' \in (2^{\Sigma \cup \Sigma'})^\infty$.

For a k-tuple $a = (a_1, \dots, a_k)$ we define the *projection* to the i-th component of a as $\#_i(a) = a_i$.

2.2. Monolithic and Distributed Systems

In this thesis, we consider *reactive systems* [HP84]. Such systems continually receive inputs from their environment and react to them by producing outputs. Furthermore, they run for an indefinite amount of time, i.e., they do not terminate. Thus, a reactive system has an infinite input/output behavior. Since we only consider reactive systems, we call reactive systems simply systems in the remainder of this thesis. Every system consists of n system processes $p_1, \ldots, p_n$, which may interact with each other as well, and a process p_{env} modeling the system's environment. We capture the design of a system with its *architecture*:

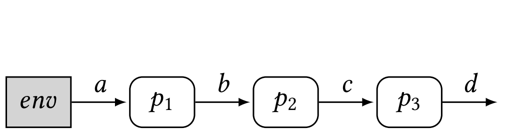
Definition 2.1 (System Architecture).
An *architecture* $\mathcal{A}$ is a tuple $\mathcal{A} = (P, V, I, O)$, where P is a set of processes consisting of the environment process p_{env} and a set $P^- = P \setminus \{p_{env}\}$ of n system processes, V is a finite set of variables, $I = \langle I_1, \ldots, I_n \rangle$ assigns a set $I_i \subseteq V$ of inputs to each system process $p_i \in P^-$, and $O = \langle O_1, \ldots, O_n \rangle$ assigns a set $O_i \subseteq V$ of outputs to each process $p_i \in P$. For every $p_i \in P^-$, the inputs and outputs are disjoint, i.e., $I_i \cap O_i = \emptyset$. The processes have pairwise disjoint output variables, i.e., $O_i \cap O_j = \emptyset$ holds for all $p_i, p_j \in P$ with $i \neq j$. The variables V of the whole system are the inputs and outputs of all processes, i.e., $V = \bigcup_{p_i \in P^-} I_i \cup \bigcup_{p_i \in P} O_i$.

Given an architecture $\mathcal{A} = (P, V, I, O)$, we denote all *variables of a system process* $p_i \in P^-$ with $V_i = I_i \cup O_i$. Intuitively, a system process controls its outputs and can observe its inputs. All other variables of the system, however, are unobservable. Thus, all variables a system process $p_i \in P^-$ can interact with are captured in its variables V_i. We denote all system output variables with $O^- = \bigcup_{p_i \in P^-} O_i$ and all system input variables with $I^- = \bigcup_{p_i \in P^-} I_i$.

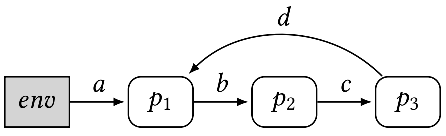
For an architecture $\mathcal{A} = (P, V, I, O)$ with $|P^-| \geq 2$, the *parallel composition* $p_1 \mid\mid p_2$ of two system processes $p_1, p_2 \in P^-$ is a process with inputs $I_{p_1 || p_2} = (I_1 \cup I_2) \setminus (O_1 \cup O_2)$ and outputs $O_{p_1 || p_2} = O_1 \cup O_2$. Although such composed processes are not directly part of the architecture $\mathcal{A}$, we call them processes in the remainder of this thesis as well. Whenever the context is clear, we do not distinguish between composed processes and system processes. We denote the set of all system processes and all processes composed from one or more system processes with $\mathbb{P}$. That is, we define $\mathbb{P} = \left\{ p_{i_1} \mid\mid \ldots \mid\mid p_{i_m} \mid \{p_{i_1}, \ldots, p_{i_m}\} \in 2^{P^-} \setminus \emptyset \right\}$. For all processes $p_i \in \mathbb{P}$, we denote their sets of variables, inputs, and outputs with V_i, I_i, and O_i, respectively, irrespective of whether p_i is a system process or a process composed from one or more system processes. While this introduces ambiguity in general, it is, in this thesis, always clear from the context whether the sets V_i, I_i, and O_i refer to the variables, inputs, and outputs, respectively, of the i-th *system* process $p_i \in P^-$ or of the i-th process $p_i \in \mathbb{P}$.

We call an architecture $\mathcal{A} = (P, V, I, O)$ *distributed* if $|P^-| > 1$ holds, i.e., if it contains at least two system processes. Otherwise, it is called *monolithic*. In the remainder of this thesis, we assume that an architecture, either distributed (Part I) or monolithic (Part II), is given.

Example 2.1. In Figure 2.1, two distributed architectures $\mathcal{A}_1$ and $\mathcal{A}_2$ are depicted. Both consist of three system processes p_1, p_2, and p_3 and the environment process p_{env}. Furthermore, we have $I = \{a\}$ and $O = \{b, c, d\}$ for both architectures. For architecture $\mathcal{A}_1$ depicted in Figure 2.1a, we obtain the sets $I_1 = \{a\}$, $I_2 = \{b\}$, and $I_3 = \{c\}$ of input variables of the system processes as well as the sets $O_1 = \{b\}$, $O_2 = \{c\}$, and $O_3 = \{d\}$ of outputs. For architecture $\mathcal{A}_2$ depicted in Figure 2.1b, we obtain the sets $I_1 = \{a, d\}$, $I_2 = \{b\}$, and $I_3 = \{c\}$ of input variables of the system processes as well as the sets $O_1 = \{b\}$, $O_2 = \{c\}$, and $O_3 = \{d\}$ of outputs. Clearly, in both architectures, the sets of inputs and outputs of a system process are disjoint and the



(a) Pipeline architecture $\mathscr{A}_1$



(b) One-way ring architecture $\mathscr{A}_2$

Figure 2.1.: Two distributed system architectures $\mathscr{A}_1$ and $\mathscr{A}_2$.

system processes do not share output variables. The parallel compositions $p_2 \mid\mid p_3$ of the two system processes p_2 of p_3 of $\mathscr{A}_1$ is defined by the set $I_{p_2 \mid\mid p_3} = \{b\}$ of input variables and the set $O_{p_1 \mid\mid p_2} = \{c, d\}$ of output variables. △

2.3. Linear-time Properties

Linear-time properties describe requirements of a system. Given a finite set of atomic propositions Σ, a *linear-time property* P is an ω-language over 2^Σ, i.e., it is a set of infinite words over 2^Σ. Hence, $P \subseteq (2^\Sigma)^\omega$ holds. In the following, we consider two types of linear-time properties.

A *safety property* is a linear-time property that intuitively describes that "nothing bad happens" [Lam77]. A typical safety property is, for instance, that a system never reaches an unsafe state. Formally, safety properties are defined as follows:

Definition 2.2 (Safety Property)**.**
A *safety property* is a linear-time property $P \subseteq (2^\Sigma)^\omega$ such that for all words $\sigma \in (2^\Sigma)^\omega \setminus P$, there exists a finite prefix $\eta \in (2^\Sigma)^*$ of σ such that

$$P \cap \left\{\sigma' \in (2^\Sigma)^\omega \mid \eta \in \mathrm{Pref}(\sigma')\right\} = \emptyset.$$

Thus, for every word $\sigma \in (2^\Sigma)^\omega$ that does not lie in P, there exists a finite prefix $\eta \in (2^\Sigma)^*$ of σ such that all infinite extensions of η, i.e., sequences $\hat{\sigma} \in (2^\Sigma)^\omega$ with $\hat{\sigma}_0 \dots \hat{\sigma}_{|\eta|-1} = \eta$, do not lie in P either. We call η a *bad prefix* for σ.

In contrast, a *liveness property* is a linear-time property that, intuitively, describes that "something good eventually happens" [Lam77]. A typical liveness property is, for instance, that a system eventually terminates. Formally, liveness properties are defined as follows:

Definition 2.3 (Liveness Property)**.**
A *liveness property* is a linear-time property $P \subseteq (2^\Sigma)^\omega$ such that

$$\left\{\eta \in (2^\Sigma)^* \mid \exists \sigma \in P.\ \eta \in \mathrm{Pref}(\sigma)\right\} = (2^\Sigma)^*.$$

Hence, for every finite sequence $\eta \in (2^\Sigma)^*$ there exists an infinite extension $\sigma \in (2^\Sigma)^\omega$ of η that lies in P. Not every linear-time property is a safety or liveness property. However, for every linear-time property $P \subseteq (2^\Sigma)^\omega$, there exists an equivalent linear-time property $P' \subseteq (2^\Sigma)^\omega$ that is a conjunction of a safety and a liveness property [AS87].

2.4. Linear-time Temporal Logic

Linear-time temporal logic (LTL) [Pnu77] is a common specification language for linear-time properties. For a finite set of atomic propositions Σ and $a \in \Sigma$, the syntax of LTL is given by

$$\varphi, \psi = q\mathit{true} \mid a \mid \neg\varphi \mid \varphi \vee \psi \mid \bigcirc \varphi \mid \varphi \mathcal{U} \psi.$$

We derive the usual Boolean operators $\mathit{false} = q\neg \mathit{true}$, $\varphi \wedge \psi = q\neg(\neg\varphi \vee \neg\psi)$, $\varphi \rightarrow \psi = q\neg\varphi \vee \psi$, and $\varphi \leftrightarrow \psi = (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$. In addition to the temporal operators *next* $\bigcirc \varphi$ and *until* $\varphi \mathcal{U} \psi$, we use the derived operators *eventually* $\Diamond \varphi = q\mathit{true} \mathcal{U} \varphi$, *globally* $\Box \varphi = q\neg \Diamond \neg\varphi$, and *weak until* $\varphi \mathcal{W} \psi = q\Box\varphi \vee (\varphi \mathcal{U} \psi)$.

The satisfaction relation $\sigma, k \models \varphi$ for an infinite word $\sigma \in (2^\Sigma)^\omega$, a point in time $k \in \mathbb{N}_0$, and an LTL formula φ is defined by

$\sigma, k \models \mathit{true}$		
$\sigma, k \models a$	iff	$a \in \sigma_k$
$\sigma, k \models \neg\psi$	iff	$\sigma, k \not\models \psi$
$\sigma, k \models \varphi \vee \psi$	iff	$\sigma, k \models \varphi$ or $\sigma, k \models \psi$
$\sigma, k \models \bigcirc \varphi$	iff	$\sigma, k+1 \models \varphi$
$\sigma, k \models \varphi \mathcal{U} \psi$	iff	$\exists j \geq k.\ \sigma, j \models \psi \wedge \forall k \leq \ell \leq j.\ \sigma, \ell \models \varphi.$

An infinite word $\sigma \in (2^\Sigma)^\omega$ *satisfies* and LTL formula φ if, and only if, $\sigma, 0 \models \varphi$ holds. We also write $\sigma \models \varphi$ for $\sigma, 0 \models \varphi$. The *language* $\mathcal{L}(\varphi)$ of an LTL formula φ is the set of infinite words that satisfy φ, i.e., $\mathcal{L}(\varphi) = \{\sigma \in (2^\Sigma)^\omega \mid \sigma \models \varphi\}$. We denote the set of atomic propositions occurring in an LTL formula φ with $\mathit{prop}(\varphi) \subseteq \Sigma$. The length of an LTL formula φ is denoted with $|\varphi|$. We represent a conjunctive LTL formula $\varphi = \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_j$ also by the set of its conjuncts, i.e., by $\{\psi_1, \psi_2, \ldots, \psi_j\}$. In the remainder of this thesis, we only consider systems whose requirements are described with LTL formulas. Therefore, we use the terms *specification* and *LTL formula* as synonyms.

Example 2.2. Consider the LTL formula $\varphi = \Diamond a \wedge \Diamond b$ over atomic propositions $\{a, b\}$. It describes that both atomic propositions a and b need to be set to *true* eventually. The infinite words $\sigma = \emptyset\emptyset\{a, b\}\emptyset^\omega$ and $\sigma' = \{a\}\{b\}\emptyset^\omega$ both satisfy φ, i.e., we have $\sigma \models \varphi$ and $\sigma' \models \varphi$. The infinite word $\sigma'' = \{a\}^\omega$, in contrast, violates φ, i.e., we have $\sigma'' \not\models \varphi$. The language $\mathcal{L}(\varphi)$ of φ is a liveness property. △

2.5. ω-Automata

Automata are another concept for expressing system requirements. Since we consider reactive systems and thus need to specify infinite temporal behavior, we utilize ω-automata. In this thesis, we consider both *alternating* automata and *non-alternating* automata. For the latter, we consider both nondeterministic and universal branching. Moreover, we consider two different types of acceptance conditions of ω-automata: *Büchi* and *co-Büchi* acceptance.

First, we introduce nondeterministic and universal ω-automata. Afterward, we present alternating ω-automata, permitting both types of branching. Although non-alternating automata are technically special cases of alternating automata, we define them separately for ease of presentation. Lastly, we present the Büchi and co-Büchi acceptance conditions.

2.5.1. Nondeterministic and Universal ω-Automata

Intuitively, ω-automata are similar to finite automata but read infinite sequences instead of finite ones. Consequently, the acceptance condition of an ω-automaton is also defined on infinite sequences. In the following, we focus on non-alternating ω-automata. We use the terms ω-automaton and non-alternating ω-automation as synonyms. Formally, non-alternating ω-automata are defined as follows.

Definition 2.4 (Non-Alternating ω-Automaton)**.**
Let Σ be a finite alphabet. An *ω-automaton* over Σ is a tuple $\mathcal{A} = (Q, q_0, \delta, Acc)$, where Q is a finite set of states, $q_0 \in Q$ is the designated initial state, $\delta : Q \times 2^\Sigma \times Q$ is a transition relation, and $Acc \subseteq Q^\infty$ is an acceptance condition.

An ω-automaton $\mathcal{A} = (Q, q_0, \delta, Acc)$ is called *complete* if, and only if, there is at least one successor state for every source state and every valuation of variables, i.e., for all $q \in Q$ and $\iota \in 2^\Sigma$, there is some $q' \in Q$ with $(q, \iota, q') \in \delta$. It is called *deterministic* if, and only if, there is at most one successor state for every source state and every valuation of variables, i.e., for all $q \in Q$ and $\iota \in 2^\Sigma$, we have $q' = q''$ for all $q', q'' \in Q$ with both $(q, \iota, q') \in \delta$ and $(q, \iota, q'') \in \delta$.

If $\mathcal{A}$ is not deterministic, we distinguish two branching types: *nondeterministic branching* and *universal branching*. A non-alternating ω-automaton has either purely nondeterministic or purely universal branching. We call an ω-automaton with nondeterministic branching also *nondeterministic ω-automaton*. An ω-automaton with universal branching is also called *universal ω-automaton*. Depending on the branching type of the automaton, we interpret the choice for a successor state to be either existential or universal. Therefore, we define the runs of both automata types similarly, namely as sequences, and interpret them differently, namely as sequences and trees, respectively, when characterizing the acceptance of a word:

Definition 2.5 (Run of a non-Alternating ω-Automaton)**.**
Let $\mathcal{A} = (Q, q_0, \delta, Acc)$ be a non-alternating ω-automaton with alphabet Σ. Let $\sigma \in (2^\Sigma)^\infty$ be a sequence. A *run* of $\mathcal{A}$ induced by σ is a sequence $r = q_0 q_1 \ldots \in Q^\infty$ with $(q_k, \sigma_k, q_{k+1}) \in \delta$ for all $k \geq 0$ with $k + 1 < |\sigma|$. A run r is accepting if, and only if, $r \in Acc$ holds.

The runs produced by a complete ω-automaton on an infinite sequence are also infinite. An infinite word $\sigma \in (2^\Sigma)^\omega$ can induce several runs for both nondeterministic and universal ω-automata $\mathcal{A}$. The set of all such runs is denoted with $Runs(\mathcal{A}, \sigma)$. In order to accept an infinite word $\sigma \in (2^\Sigma)^\omega$, a nondeterministic ω-automaton $\mathcal{A}$ is required to have *some* accepting run for σ. For a universal ω-automaton, in contrast, *all* runs of $\mathcal{A}$ induced by σ need to be accepting. Formally, a nondeterministic ω-automaton $\mathcal{A}$ accepts a word $\sigma \in (2^\Sigma)^\omega$ if, and only if, there exists a run $r \in \mathit{Runs}(\mathcal{A}, \sigma)$ that is accepting. A universal ω-automaton $\mathcal{A}$ accepts a word $\sigma \in (2^{\Sigma})^{\omega}$ if, and only if, all runs $r \in \mathit{Runs}(\mathcal{A}, \sigma)$ are accepting. Irrespective of the branching type, the *language* $\mathcal{L}(\mathcal{A})$ of an ω-automaton $\mathcal{A}$ is the set of all accepted words, i.e., $\mathcal{L}(\mathcal{A}) = \{\sigma \in (2^{\Sigma})^{\omega} \mid \mathcal{A} \text{ accepts } \sigma\}$. To further distinguish nondeterministic and universal ω-automata, we often interpret the set of runs of a universal automaton as a single tree. We formalize trees when introducing alternating ω-automata in the next section.

2.5.2. Alternating ω-Automata

An *alternating* ω-automaton allows for both *existential* and *universal* choices. Intuitively, existential choices can be seen as "or"-choices that allow for choosing *one* of the possible successor states as the actual successor, while universal choices are "and"-choices, where *all* of the possible successor states constitute an actual successor. Therefore, the transition function of an alternating automaton yields a *positive Boolean formula* over the set of states.

The positive Boolean formulas over a set X, denoted $\mathbb{B}^{+}(X)$, are the formulas built from elements of X, conjunction, disjunction, *true*, and *false*. A set $Y \subseteq X$ satisfies a positive Boolean formula $\xi \in \mathbb{B}^{+}(X)$, denoted $Y \models \xi$, if, and only if, the truth assignment which assigns *true* to all variables in Y and *false* to all variables in $X \setminus Y$ satisfies ξ. We assume that the elements of $\mathbb{B}^{+}(X)$ are given in *disjunctive normal form* (DNF), i.e., as a disjunction of one or more conjunctions of literals of X. Since every propositional formula can be converted into an equivalent one in disjunctive normal form, this assumption does not restrict the possible elements of $\mathbb{B}^{+}(X)$. We represent a propositional formula $\bigvee_i \bigwedge_j c_{i,j}$ in disjunctive normal form also in its set notation $\bigcup_i \{\bigcup_j \{c_{i,j}\}\}$. Formally, an alternating ω-automaton is then defined as follows:

Definition 2.6 (Alternating ω-Automaton).
Let Σ be a finite alphabet. An *alternating ω-automaton* over Σ is a tuple $\mathcal{A} = (Q, q_0, \delta, \mathit{Acc})$, where Q is a finite set of states, q_0 is the designated initial state, $\delta : Q \times \Sigma \to \mathbb{B}^{+}(Q)$ is a transition function, and $\mathit{Acc} \subseteq Q^{\infty}$ is an acceptance condition.

Since alternating ω-automata allow for universal choices, their runs form *trees* instead of sequences. Intuitively, a run tree's branching then represents the automaton's universal choices. Note that due to the existence of additional existential choices, every alternating ω-automaton induces a set of run trees on every input sequence. Formally, a tree is defined as follows:

Definition 2.7 (Σ-labeled Tree).
Let Σ be a finite alphabet and let D be a set of *directions*. A *tree* $\mathbb{T}$ *over* D is a prefix-closed subset of D^*, i.e., $\mathbb{T} \subseteq D^*$ holds and if we have $x \cdot d \in \mathbb{T}$ then $x \in \mathbb{T}$ holds as well. We refer to the elements $x \in \mathbb{T}$ of $\mathbb{T}$ as *nodes*. The *depth* of a node x is denoted with $|x|$. The empty sequence ε is called the *root*. The children of a node $x \in \mathbb{T}$ are the nodes $\mathit{children}(x) = \{x \cdot d \in \mathbb{T} \mid d \in D\}$. A *$\Sigma$-labeled tree* $(\mathbb{T}, \ell)$ *over* D consists of a tree $\mathbb{T}$ over directions D and a labeling function $\ell : \mathbb{T} \to \Sigma$. A *branch* of $(\mathbb{T}, \ell)$ is a maximal sequence $\ell(x_0)\ell(x_1)\ldots \in \Sigma^{\infty}$ with $x_0 = \varepsilon$ and $x_{j+1} \in \mathit{children}(x_j)$ for every $j \in \mathbb{N}_0$.

For every node $x \in \mathbb{T}$ of a tree $\mathbb{T}$, there exists a unique finite sequence of nodes in $\mathbb{T}$ that, starting from the root ε of $\mathbb{T}$, reaches node x. We call this sequence the *prefix of x in $\mathbb{T}$* and denote it with $\mathit{pref}(\mathbb{T}, x,)$. Furthermore, we denote the set of all branches of a Σ-labeled tree $(\mathbb{T}, \ell)$ with $\mathit{Branches}(\mathbb{T}, \ell)$ and the set of infinite branches of $(\mathbb{T}, \ell)$ with $\mathit{Branches}_{\mathit{Inf}}(\mathbb{T}, \ell)$.

A run tree of an alternating ω-automaton $\mathcal{A} = (Q, q_0, \delta, \mathit{acc})$ is then a tree that is labeled in the states of $\mathcal{A}$, i.e., in Q. The labeling function is defined according to $\mathcal{A}$'s transition function δ. Intuitively, every universal choice in $\mathcal{A}$ defined by δ yields a new branch in the run tree, while every existential choice induces a new run tree. Formally, a run tree of an alternating ω-automaton is defined as follows:

Definition 2.8 (Run of an Alternating ω-Automaton)**.**
Let $\mathcal{A} = (Q, q_0, \delta, \mathit{Acc})$ be an alternating ω-automaton with alphabet Σ. Let $\sigma \in (2^{\Sigma})^{\omega}$ be an infinite sequence. A *run tree* of $\mathcal{A}$ induced by σ is a Q-labeled tree $(\mathbb{T}, \ell)$ with $\ell(\varepsilon) = q_0$ and $\{\ell(x') \mid x' \in \mathit{children}(x)\} \models \delta(\ell(x), \sigma_{|x|})$ for all $x \in \mathbb{T}$. A run tree $(\mathbb{T}, \ell)$ is accepting if, and only if, $b \in \mathit{Acc}$ holds for all of $(\mathbb{T}, \ell)$'s branches b.

An alternating ω-automaton $\mathcal{A}$ over alphabet Σ induces several run trees on a single infinite sequence $\sigma \in (2^{\Sigma})^{\omega}$ if existential choices occur during a run of $\mathcal{A}$ on σ. Slightly overloading notation, we denote the set of all such run trees by $\mathit{Runs}(\mathcal{A}, \sigma)$. In order to accept an infinite word $\sigma \in (2^{\Sigma})^{\omega}$, an alternating ω-automata is only required to induce *some* accepting run tree for σ. Formally an alternating ω-automaton $\mathcal{A}$ over alphabet Σ accepts an infinite word $\sigma \in (2^{\Sigma})^{\omega}$ if, and only if, there exists a run tree $r \in \mathit{Runs}(\mathcal{A}, \sigma)$ of $\mathcal{A}$ induced by σ that is accepting. The *language* $\mathcal{L}(\mathcal{A})$ of an alternating ω-automaton $\mathcal{A}$ over alphabet Σ is the set of all accepted words, i.e., we have $\mathcal{L}(\mathcal{A}) = \{\sigma \in (2^{\Sigma})^{\omega} \mid \mathcal{A} \text{ accepts } \sigma\}$.

2.5.3. Büchi and co-Büchi Acceptance Conditions

There are several types of acceptance conditions for ω-automata. In this thesis, we focus on the *Büchi* and *co-Büchi* conditions. Both these acceptance conditions are defined over the set $\mathrm{Inf}(r) \subseteq Q$ of states of the automaton $\mathcal{A} = (Q, q_0, \delta, \mathit{Acc})$ that occur infinitely often in a run r of $\mathcal{A}$. Formally, $\mathrm{Inf}(r)$ is given by $\mathrm{Inf}(r) = \{q \in Q \mid \forall k \geq 0. \exists k < j \leq |r|.\ r_j = q\}$. We now define both Büchi and co-Büchi acceptance based on $\mathrm{Inf}(r)$.

Intuitively, the *Büchi* acceptance condition states that a certain set $F \subseteq Q$ of so-called *accepting states* needs to be visited infinitely often. Formally:

Definition 2.9 (Büchi Acceptance Condition)**.**
Let $\mathcal{A} = (Q, q_0, \delta, \mathit{Acc})$ be an ω-automaton over alphabet Σ. Let $F \subseteq Q$ be a set of *accepting states.* A run r of $\mathcal{A}$ is accepted by the *Büchi condition* if, and only if, $F \cap \mathrm{Inf}(r) \neq \emptyset$ holds. Hence, we define

$$\mathit{Acc}_{\mathit{Büchi}}(F) = \{r \in Q^{\infty} \mid F \cap \mathrm{Inf}(r) \neq \emptyset\}.$$

By definition of $\mathrm{Inf}(r)$, a finite run can never visit any state infinitely often. Hence, a finite run only satisfies the Büchi acceptance condition if the set of accepting states is empty. We call an alternating, nondeterministic, or universal ω-automaton with Büchi acceptance condition *alternating Büchi automaton* (ABA), *nondeterministic Büchi automaton* (NBA), or *universal Büchi automaton* (UBA), respectively.

Specifications given as LTL formulas can be translated into alternating and nondeterministic Büchi automata with a linear and exponential blow-up in the automaton size, respectively:

Proposition 2.1 ([MSS88])**.** *Let φ be an LTL formula. There exists an alternating Büchi automaton $\mathcal{A}_\varphi$ with $O(|\varphi|)$ states such that $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$ holds.*

Proposition 2.2 ([KV05])**.** *Let φ be an LTL formula. There exists a nondeterministic Büchi automaton $\mathcal{A}_\varphi$ with $O(2^{|\varphi|})$ states such that $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$ holds.*

The *co-Büchi* condition, in contrast, intuitively requires that a certain set $F \subseteq Q$ of so-called *rejecting states* must be visited only finitely many times. Formally:

Definition 2.10 (Co-Büchi Acceptance Condition)**.**
Let $\mathcal{A} = (Q, q_0, \delta, Acc)$ be an ω-automaton over alphabet Σ. Let $F \subseteq Q$ be a set of *rejecting states.* A run r of $\mathcal{A}$ is accepted by the *co-Büchi condition* if, and only if, $F \cap \mathrm{Inf}(r) = \emptyset$ holds. Hence, we define

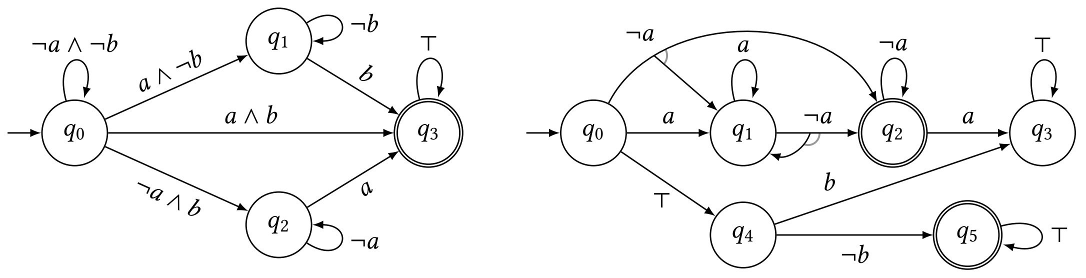
$$Acc_{\textit{co-Büchi}}(F) = \{r \in Q^\infty \mid F \cap \mathrm{Inf}(r) = \emptyset\}\,.$$

By definition of $\mathrm{Inf}(r)$, a finite run can never visit any state infinitely often. Hence, a finite run trivially satisfies the co-Büchi acceptance condition for any set of rejecting states. We call an alternating, nondeterministic, or universal ω-automaton with co-Büchi acceptance condition *alternating co-Büchi automaton* (ACA), *nondeterministic co-Büchi automaton* (NCA), or *universal co-Büchi automaton* (UCA), respectively.

Similar to Büchi automata, LTL formulas can also be translated into alternating and universal *co-Büchi* automata with a linear and exponential blow-up in the automaton size, respectively. These results follow from Propositions 2.1 and 2.2, i.e., the respective results for Büchi automata, and the observation that the Büchi and co-Büchi acceptance conditions, as well as nondeterministic and universal branching, are dual:

Proposition 2.3. *Let φ be an LTL formula. There exists an alternating co-Büchi automaton $\mathcal{A}_\varphi$ with $O(|\varphi|)$ states such that $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$ holds.*

Proof. There exists an alternating Büchi automaton $\mathcal{B}_{\neg\varphi} = (Q, q_0, \delta, Acc_{\text{Büchi}}(F))$ with $O(|\neg\varphi|)$ states such that $\mathcal{L}(\mathcal{B}_{\neg\varphi}) = \mathcal{L}(\neg\varphi)$ holds by Proposition 2.1. We construct an alternating co-Büchi automaton $\mathcal{A}_\varphi$ from $\mathcal{B}_{\neg\varphi}$ as follows: $\mathcal{A}_\varphi = (Q, q_0, \delta', Acc_{\textit{co-Büchi}}(F))$, where δ' is the transition function obtained from δ when replacing all conjunctions with disjunctions and vice versa. Hence, $\mathcal{A}_\varphi$ is a copy of $\mathcal{B}_{\neg\varphi}$ with *dual transition function* and, as the accepting states of $\mathcal{B}_{\neg\varphi}$ are interpreted as rejecting states in $\mathcal{A}_\varphi$, with *dual acceptance condition.* Due to the duality of the Büchi and co-Büchi acceptance conditions as well as nondeterministic and universal branching, $\mathcal{L}(\mathcal{A}_\varphi) = \overline{\mathcal{L}(\neg\varphi)} = \mathcal{L}(\varphi)$ follows. Since $O(|\neg\varphi|) = O(|\varphi|)$ holds, the universal co-Büchi automaton $\mathcal{A}_\varphi$ has $O(|\varphi|)$ states. □



(a) Büchi automaton $\mathcal{A}_\varphi$ for $\varphi = \diamondsuit a \wedge \diamondsuit b$. (b) Co-Büchi automaton $\mathcal{A}_\psi$ for $\psi = \square\diamondsuit a \vee \bigcirc b$.

Figure 2.2.: Non-alternating Büchi automaton $\mathcal{A}_\varphi$ and alternating co-Büchi automaton $\mathcal{A}_\psi$.

Proposition 2.4. *Let φ be an LTL formula. There exists a universal co-Büchi automaton $\mathcal{A}_\varphi$ with $O(2^{|\varphi|})$ states such that $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$ holds.*

Proof. There is a nondeterministic Büchi automaton $\mathcal{B}_{\neg\varphi} = (Q, q_0, \delta, Acc_{\text{Büchi}}(F))$ with $O(|\neg\varphi|)$ states such that $\mathcal{L}(\mathcal{B}_{\neg\varphi}) = \mathcal{L}(\neg\varphi)$ holds by Proposition 2.2. We construct a universal co-Büchi automaton $\mathcal{A}_\varphi$ from $\mathcal{B}_{\neg\varphi}$ as follows: $\mathcal{A}_\varphi = (Q, q_0, \delta, Acc_{\textit{co-Büchi}}(F))$. Hence, $\mathcal{A}_\varphi$ is a copy of $\mathcal{B}_{\neg\varphi}$, where nondeterministic transitions are interpreted as universal ones and accepting states as rejecting ones. Therefore, due to the duality of the Büchi and co-Büchi acceptance conditions as well as nondeterministic and universal branching, $\mathcal{L}(\mathcal{A}_\varphi) = \overline{\mathcal{L}(\neg\varphi)} = \mathcal{L}(\varphi)$ follows. Since $O(|\neg\varphi|) = O(|\varphi|)$ holds, $\mathcal{A}_\varphi$ has $O(|\varphi|)$ states. □

In the remainder of this thesis, we only consider ω-automata with Büchi and co-Büchi acceptance conditions. For ease of presentation, we, therefore, denote the acceptance condition simply with the set F of accepting or rejecting states, respectively, whenever the acceptance type is clear from the context. We represent an alternating or non-alternating ω-automaton $\mathcal{A} = (Q, q_0, \delta, F)$ over alphabet Σ with Büchi or co-Büchi acceptance as a directed graph with vertex set Q and a symbolic representation of the transition relation δ as propositional formulas $\mathbb{B}(\Sigma)$. In alternating ω-automata, we depict universal choices by connecting the transitions with a gray arc. In both alternating and non-alternating ω-automata, the accepting or rejecting states in F are marked with double circles.

Example 2.3. First, consider the LTL formula $\varphi = \diamondsuit a \wedge \diamondsuit b$, which describes that both atomic propositions a and b need to be set to *true* eventually. Consider the non-alternating Büchi automaton $\mathcal{A}_\varphi$ shown in Figure 2.2a. It is deterministic and thus induces for each infinite sequence $\sigma \in (2^{\{a,b\}})^\omega$ only a single run. The only accepting state is q_3, which is a sink state, i.e., once entering it, it can never be left again. Thus, all runs that do not enter q_3 eventually are rejecting. Clearly, all sequences $\sigma \in (2^{\{a,b\}})^\omega$ for which a and b are set to *true* eventually induce a run that enters q_3. Hence, $\mathcal{A}_\varphi$ accepts the same language as φ.

Next, consider the LTL formula $\psi = \square\diamondsuit a \vee \bigcirc b$. It requires that either atomic proposition b is set to *true* in the next time step or atomic proposition a is set to *true* infinitely often. Consider the alternating co-Büchi automaton $\mathcal{A}_\psi$ depicted in Figure 2.2b. It contains both existential and universal transitions. First, consider a sequence $\sigma \in (2^{\{a,b\}})^\omega$ with $b \in \sigma_1$, i.e., where b is set to *true* in the second time step. Among others, σ induces a run tree with a single branch in which we, starting from q_0, move to q_4 and then to q_3. The non-rejecting state q_3 is never left. Hence, an accepting run tree of $\mathcal{A}_\psi$ induced by such sequences σ exists. Second, consider a sequence $\sigma \in (2^{\{a,b\}})^\omega$ with $\sigma \models \square\diamond a$ and $\sigma \not\models \bigcirc b$. Note that although σ induces a run tree with a single branch that moves from q_0 to q_4 in the first time step, this run tree is rejecting since $b \notin \sigma_1$ holds by assumption and therefore the run tree enters the rejecting sink q_5. Since an alternating co-Büchi automaton only requires the existence of some accepting run tree, this particular run tree does not result in $\mathcal{A}_\psi$ rejecting σ. In the following, we can thus ignore this run tree and focus on the upper part of the automaton. Note that this part contains only universal choices and thus induces only a single run tree, yet, with possibly multiple branches. This run tree can only be rejecting if it contains a branch that visits q_2 infinitely often. This can only be the case if a is always set to *false* from some point in time on. Hence, such a branch cannot be induced by a σ. Since sequences that violate $\square\diamond a$ set a to *false* from some point on, such sequences, in contrast, only induce run trees that contain at least one rejecting branch. Thus, for a word $\sigma \in (2^{\{a,b\}})^\omega$ with $\sigma \not\models \square\diamond a \vee \bigcirc b$, all run trees of $\mathcal{A}_\psi$ induced by σ are rejecting while, for a word $\sigma \in (2^{\{a,b\}})^\omega$ with $\sigma \models \square\diamond a \vee \bigcirc b$ induces some accepting run tree. Therefore, $\mathcal{A}_\psi$ accepts the same language as ψ. △

2.6. System Models and Strategies

We model a reactive system with a *finite-state transducer.* Transducers are a particular type of finite-state machines that read infinite sequences over input variables and, in every step, change their internal state and produce a valuation of output variables. A *system strategy* defines the behavior of a reactive system. It maps a history of valuations of input variables to a valuation of output variables. We first introduce finite-state transducers as our model for reactive systems. Afterward, we formalize system strategies and connect them with our system model.

2.6.1. Finite-State Transducers

We consider finite-state transducers as a model for reactive systems. Thus, we consider transducers that read infinite sequences of valuations of input variables I of the system and output valuations of output variables O of the system in every step. Formally, we define finite-state (Γ, Υ)-transducers for finite sets Γ, Υ, where in our context $\Gamma = 2^I$ and $\Upsilon = 2^O$.

> **Definition 2.11** (Finite-state (Γ, Υ)-Transducer).
> Let Γ and Υ be finite input and output alphabets, respectively. A *finite-state* (Γ, Υ)*-transducer* $\mathcal{T} = (T, T_0, \tau, \ell)$ consists of a finite set of states t, a designated set of initial states $T_0 \subseteq T$, a transition relation $\tau : T \times \Gamma \times T$, and a labeling relation $\ell : T \times \Gamma \times \Upsilon$.

In the remainder of this thesis, we assume, without loss of generality, that all states of a finite-state transducer are reachable. A finite-state (Γ, Υ)-transducer $\mathcal{T} = (T, T_0, \tau, \ell)$ is called *transition-deterministic* if, and only if, $|T_0| \leq 1$ holds and if for all states $t \in T$ and all inputs $\iota \in \Gamma$, there exists *at most* one state $t' \in T$ such that $(t, \iota, t') \in \tau$ holds. It is called *labeling-deterministic* if, and only if, for all states $t \in T$ and all inputs $\iota \in \Gamma$, there exists *at most* one output $o \in \Upsilon$ such that $\ell(t, \iota) = o$ holds. If $\mathcal{T}$ is both transition-deterministic and labeling-deterministic, then we call it simply *deterministic*. The finite-state transducer $\mathcal{T}$ is called *transition-complete* if $|T_0| \geq 1$ holds and if for all states $t \in T$ and all inputs $\iota \in \Gamma$, there exists *at least* one state $t' \in T$ such that $(t, \iota, t') \in \tau$ holds. It is called *labeling-complete* if for all states $t \in T$ and all inputs $\iota \in \Gamma$, there exists some output $o \in \Upsilon$ such that $(t, \iota, o) \in \ell$ holds. If $\mathcal{T}$ is both transition-complete and labeling-complete, then we call it simply *complete*.

We distinguish between two types of finite-state transducers, *Moore* transducers and *Mealy* transducers. For the former, the labeling depends only on the state, not on the input. Formally, for all states $t \in T$, we have $\{o \in \Upsilon \mid (t, \iota, o) \in \ell\} = \{o \in \Upsilon \mid (t, \iota', o) \in \ell\}$ for all input valuations $\iota, \iota' \in \Gamma$. Slightly misusing notation, we thus consider the labeling relation ℓ for a Moore transducers $\mathcal{T} = (T, T_0, \tau, \ell)$ to be of type $T \times \Upsilon$. Furthermore, for labeling-deterministic Moore transducers, we then also write $\ell(t) = o$ instead of $(t, o) \in \ell$. For Mealy transducers, in contrast, the labeling of a transition may also depend on the input valuation, i.e., we might have $\{o \in \Upsilon \mid (t, \iota, o) \in \ell\} \neq \{o \in \Upsilon \mid (t, \iota', o) \in \ell\}$ for input valuations $\iota, \iota' \in \Gamma$ with $\iota \neq \iota'$.

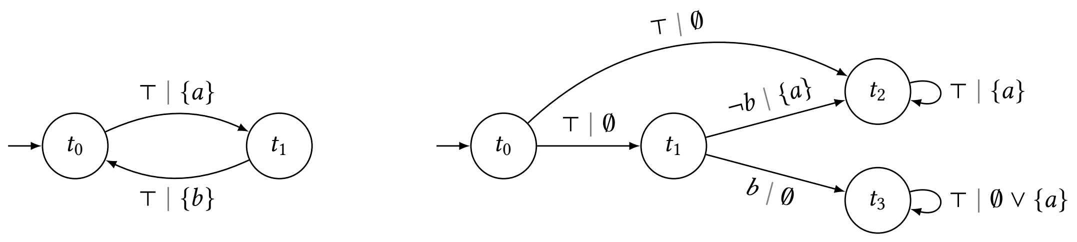
Given an infinite input word $\gamma = \gamma_0\gamma_1 \ldots \in \Gamma^\omega$, a finite-state (Γ, Υ)-transducer $\mathcal{T} = (T, T_0, \tau, \ell)$ defines a set $\mathit{Paths}(\mathcal{T}, \gamma)$ of finite or infinite sequences $\pi = (t_0, v_0)(t_1, v_1) \ldots \in (T \times \Upsilon)^\infty$ of states and outputs of $\mathcal{T}$ that describes the possible internal changes of states of $\mathcal{T}$ as well as its outputs when reading γ, the so-called *paths*:

$$\begin{aligned} \mathit{Paths}(\mathcal{T}, \gamma) = q\,\{(t_0, v_0)(t_1, v_1) \ldots \in (T \times \Upsilon)^\infty \mid t_0 \in T_0 \land \\ \forall k \geq 0.\ (t_k, \gamma_k, t_{k+1}) \in \tau \ \land\ (t_k, \gamma_k, v_k) \in \ell\}\,. \end{aligned}$$

Note that if $\mathcal{T}$ produces a finite path $\pi \in (T \times \Upsilon)^*$ on input sequences $\gamma \in \Gamma^\omega$, then $\mathcal{T}$ is incomplete. The finiteness of π can be due to both transition-incompleteness and labeling-incompleteness. If π ends at point in time k due to transition-incompleteness, then the last pair (t_k, v_k) of π is built from the target state of the last successful transition as well as its labeling. If π ends at point in time k due to labeling-incompleteness, then (t_k, γ_k) is built from the target state of the last successful transition that has a labeling.

In this thesis, however, we consider labeling-complete transducers only. Hence, whenever the transducer $\mathcal{T}$ produces a finite path π on γ, then π is finite due to transition-incompleteness. Therefore, in particular, the last pair (t_k, v_k) of π is always built from the target state of the last successful transition as well as its labeling.

For every path $\pi \in \mathit{Paths}(\mathcal{T}, \gamma)$ of $\mathcal{T}$ induced by $\gamma \in \Gamma^\omega$ with $\pi = (t_0, v_0)(t_1, v_1) \ldots$, there exists a sequence $\rho = (\gamma_0, v_0)(\gamma_1, v_1) \ldots \in (\Gamma \times \Gamma)^\infty$ of valuations of input and output variables that captures the inputs of all successful transitions together with the outputs of the respective source states. Hence, intuitively, ρ combines γ with the output sequence of $\mathcal{T}$ defined by π. Note, however, that if a path $\pi \in \mathit{Paths}(\mathcal{T}, \gamma)$ is finite, the last pair state t_k occurring in π is the target state of the last successful transition since we only consider labeling-complete transducers. Thus, in particular, it is not the source state of a further successful transition. Therefore, for a finite path π, the corresponding sequence ρ only considers the labelings of the



(a) Deterministic $(2^{\{c\}}, 2^{\{a,b\}})$-transducer $\mathcal{T}_1$ with Moore semantics.

(b) Nondeterministic $(2^{\{b\}}, 2^{\{a\}})$-transducer $\mathcal{T}_2$ with Mealy semantics.

Figure 2.3.: Finite-state transducers $\mathcal{T}_1$ and $\mathcal{T}_2$. Both are complete.

states up to point in time $|\pi| - 2$ and hence $|\rho| = \max\{0, |\pi| - 1\}$ if π is finite. If π is infinite, in contrast, ρ is clearly infinite as well. The sequence ρ is called *trace*. Formally, the set of all traces of $\mathcal{T}$ for input $\gamma \in \Gamma^\omega$ is defined by

$$\mathit{Traces}(\mathcal{T}, \gamma) = q\,\{\rho \in (\Gamma \times \Upsilon)^\omega \mid \exists \pi \in \mathit{Paths}(\mathcal{T}, \gamma). \forall 0 \leq k < |\pi| - 1.\ \rho_k = (\gamma_k, \#_2(\pi_k))\}\,.$$

In our context, where we have $\Gamma = 2^I$ and $\Upsilon = 2^O$, we merge the input γ_k and output υ_k of a trace $\rho \in \mathit{Traces}(\mathcal{T}, \gamma)$ of $\mathcal{T}$ at a point in time $k \geq 0$ with $0 \leq k < |\rho|$ into a single set $\sigma_k = \gamma_k \cup \upsilon_k \in 2^{I \cup O}$. Since $I \cap O = \emptyset$ holds by definition of architectures, γ_k and υ_k are always non-contradictory, and thus building their union is uniquely possible. Slightly overloading notation, we call the merged trace simply trace as well and define

$$\mathit{Traces}(\mathcal{T}, \gamma) = q\,\{\sigma \in (\Gamma \cup \Upsilon)^\omega \mid \exists \pi \in \mathit{Paths}(\mathcal{T}, \gamma). \forall 0 \leq k < |\pi| - 1.\ \sigma_k = \gamma_k \cup \#_2(\pi_k)\}$$

directly. The set of all infinite traces produced by $\mathcal{T}$ on some infinite input sequence is defined by $\mathit{Traces}(\mathcal{T}) = \bigcup_{\gamma \in \Gamma^\omega} \mathit{Traces}(\mathcal{T}, \gamma)$.

We depict a finite-state $(2^I, 2^O)$-transducer $\mathcal{T} = (T, T_0, \tau, \ell)$ as a directed graph with vertex set T and a symbolic representation of the transition relation τ as propositional formula $\mathbb{B}(I)$. The labeling relation is depicted on the edges of the graph as well: for a transition $(t, \iota, t') \in \tau$, we add the disjunction of all $o \in 2^O$ with $(t, \iota, o) \in \ell$ to the respective edge representing a transition $(t, \iota, t') \in \tau$. The labeling is separated from the propositional formula describing the transition – and thus the input valuations – using a gray pipe.

Example 2.4. In Figure 2.3, two complete finite-state transducers are shown. Figure 2.3a depicts the $(2^{\{c\}}, 2^{\{a,b\}})$-transducer $\mathcal{T}_1 = (T_1, T_{1,0}, \tau_1, \ell_1)$ with $T_1 = \{t_0, t_1\}$, $T_{1,0} = \{t_0\}$, both $(t_0, \iota, t_1) \in \tau_1$ and $(t_1, \iota, t_0) \in \tau_1$ for all $\iota \in 2^{\{c\}}$, and both $(t_0, \iota, \{a\}) \in \ell_1$ and $(t_1, \iota, \{b\}) \in \ell_1$ for all $\iota \in 2^{\{c\}}$. Clearly, $\mathcal{T}_1$ is deterministic and has Moore semantics. Furthermore, its set of traces is given by $\mathit{Traces}(\mathcal{T}_1) = (\{a\}\{b\})^\omega \cup (2^{\{c\}})^\omega$. Figure 2.3b illustrates the $(2^{\{b\}}, 2^{\{a\}})$-transducer $\mathcal{T}_2 = (T_2, T_{2,0}, \tau_2, \ell_2)$ with $T_2 = \{t_0, t_1, t_2, t_3\}$, $T_{2,0} = \{t_0\}$ as well as $(t_0, \iota, t_1) \in \tau_2$, $(t_0, \iota, t_2) \in \tau_2$, $(t_2, \iota, t_2) \in \tau_2$, $(t_3, \iota, t_3) \in \tau_2$, $(t_0, \iota, \emptyset) \in \ell_2$, $(t_2, \iota, \{a\}) \in \ell_2$, and both $(t_3, \iota, \emptyset) \in \ell_2$ and $(t_3, \iota, \{a\}) \in \ell_2$ for all $\iota \in 2^{\{b\}}$, and $(t_1, \emptyset, t_2) \in \tau_2$, $(t_1, \{b\}, t_3) \in \tau_2$, $(t_1, \emptyset, \{a\}) \in \ell_2$, and $(t_1, \{b\}, \emptyset) \in \ell_2$. Clearly, $\mathcal{T}_2$ is nondeterministic, for instance due to the transitions with source state t_0, and has Mealy semantics due to the labeling in state t_2. △

We defined system models, and thus transducers, irrespective of the system's architecture. For a monolithic architecture $\mathscr{A} = (P, V, I, O)$, we consider a finite-state $(2^{I_1}, 2^{O_1})$-transducer modeling the single system process $p_1 \in P^-$. For a distributed architecture $\mathscr{A} = (P, V, I, O)$, we consider finite-state $(2^{I_i}, 2^{O_i})$-transducer modeling all respective system processes $p_i \in P^-$. Moreover, we are interested in a finite-state $(2^{O_{env}}, 2^{O^-})$ transducer that models the entire system, i.e., the interplay of all system processes. Note that the transducer for the whole system reads infinite words of output variables of the environment process p_{env} and is labeled in the union of the output variables of all system processes.

To model distributed systems with several processes, we often only model the processes individually, i.e., with individual finite-state $(2^{I_i}, 2^{O_i})$-transducers $\mathcal{T}_i$. To argue about the behavior of the entire system and thus to obtain the finite-state $(2^{O_{env}}, 2^{O^-})$ transducer $\mathcal{T}$ that models the full system, we, therefore, need to *compose* the individual transducers $\mathcal{T}_i$. In the following, we present how such a joined transducer $\mathcal{T}$ can be constructed: we define the *parallel composition* of two finite-state transducers.

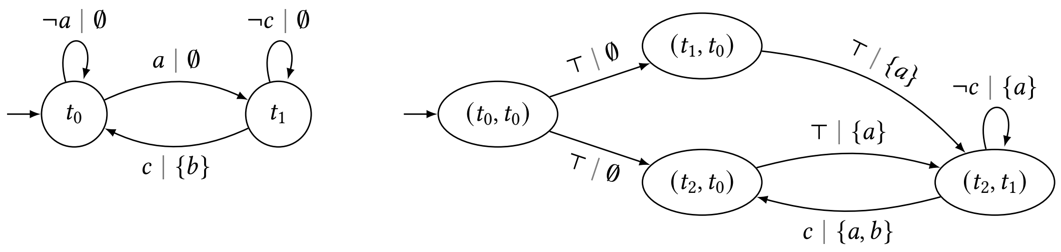
Definition 2.12 (Parallel Composition of Finite-State Transducers).
Let I_1, I_2, O_1, and O_2 be finite sets of input and output variables with $I_1 \cap O_1 = \emptyset$ and $I_2 \cap O_2 = \emptyset$. Let $\mathcal{T}_1 = (T_1, T_{1,0}, \tau_1, \ell_1)$ be a finite-state $(2^{I_1}, 2^{O_1})$-transducer and let $\mathcal{T}_2 = (T_1, T_{2,0}, \tau_2, \ell_2)$ be a finite-state $(2^{I_2}, 2^{O_2})$-transducer. The *parallel composition* of $\mathcal{T}_1$ and $\mathcal{T}_2$, denoted $\mathcal{T}_1 \mid\mid \mathcal{T}_2$, is the finite-state $(2^{(I_1 \cup I_2) \setminus (O_1 \cup O_2)}, 2^{O_1 \cup O_2})$-transducer $\mathcal{T}_{1,2} = (T, T_0, \tau, \ell)$ with

- $T = T_1 \times T_2$,
- $T_0 = T_{1,0} \times T_{2,0}$,
- $((u, v), \iota, (u', v')) \in \tau$ if, and only if, there are $o_1 \in 2^{O_1}$ and $o_2 \subseteq 2^{O_2}$ with $(u, \iota_1, o_1) \in \ell_1$ and $(v, \iota_2, o_2) \in \ell_2$ such that $(u, \iota_1, u') \in \tau_1$ and $(v, \iota_2, v') \in \tau_2$ hold, where $\iota_1 = (\iota \cup o_2) \cap I_1$ and $\iota_2 = (\iota \cup o_1) \cap I_2$, and
- $((u, v), \iota, o) \in \ell$ if, and only if, $(u, (\iota \cup o) \cap I_1, o \cap O_1) \in \ell_1$ and $(v, (\iota \cup o) \cap I_2, o \cap O_2) \in \ell_1$.

Without loss of generality, we assume in the remainder of this thesis that unreachable states are removed from $\mathcal{T}_1 \mid\mid \mathcal{T}_2$. Intuitively, the parallel composition $\mathcal{T}_1 \mid\mid \mathcal{T}_2$ of two finite-state transducers $\mathcal{T}_1$ and $\mathcal{T}_2$ is the *product* of $\mathcal{T}_1$ and $\mathcal{T}_2$. Note that the output variables of one of the transducers can be the input variables of the other one. This is carefully handled in the definition of the transition and labeling relations of $\mathcal{T}_1 \mid\mid \mathcal{T}_2$.

Example 2.5. Reconsider the nondeterministic finite-state $(2^{\{b\}}, 2^{\{a\}})$-transducer $\mathcal{T}_2$ with Mealy semantics from Figure 2.3b. Furthermore, consider the deterministic finite-state $(2^{\{c\}}, 2^{\{b\}})$-transducer $\mathcal{T}_3$ with Mealy semantics depicted in Figure 2.4a. The parallel composition $\mathcal{T}_2 \mid\mid \mathcal{T}_3$ of $\mathcal{T}_2$ and $\mathcal{T}_3$ is shown in Figure 2.4b. It is a nondeterministic finite-state $(2^{\{c\}}, 2^{\{a,b\}})$-transducer with Mealy semantics. △

Note that the parallel composition $\mathcal{T}_1 \mid\mid \mathcal{T}_2$ of two finite-state transducers $\mathcal{T}_1$ and $\mathcal{T}_2$ can be nondeterministic or incomplete even if $\mathcal{T}_1$ and $\mathcal{T}_2$ are both deterministic and complete; for instance, if both $\mathcal{T}_1$ and $\mathcal{T}_2$ have Mealy semantics and one of $\mathcal{T}_1$'s outputs is an input of $\mathcal{T}_2$ and



(a) Deterministic $(2^{\{c\}}, 2^{\{b\}})$-transducer $\mathcal{T}_3$ with Mealy semantics.

(b) Nondeterministic $(2^{\{c\}}, 2^{\{a,b\}})$-transducer $\mathcal{T}_2 \mid\mid \mathcal{T}_3$ with Mealy semantics, parallel composition of $\mathcal{T}_2$ and $\mathcal{T}_3$.

Figure 2.4.: Finite-state transducer $\mathcal{T}_3$ and the parallel composition with $\mathcal{T}_2$ from Figure 2.3b.

vice versa. If both $\mathcal{T}_1$ and $\mathcal{T}_2$ are Moore transducers, however, their parallel composition is guaranteed to be deterministic as long as they both are deterministic. Furthermore, $\mathcal{T}_1 \mid\mid \mathcal{T}_2$ is complete as long as both $\mathcal{T}_1$ and $\mathcal{T}_2$ are complete:

Lemma 2.1. *Let I_1, I_2, O_1, and O_2 be finite sets with $I_1 \cap O_1 = \emptyset$ and $I_2 \cap O_2 = \emptyset$. Let $\mathcal{T}_1$ be a finite-state $(2^{I_1}, 2^{O_1})$-transducer and let $\mathcal{T}_2$ be a finite-state $(2^{I_2}, 2^{O_2})$-transducer. Let both $\mathcal{T}_1$ and $\mathcal{T}_2$ have Moore semantics. Then $\mathcal{T}_1 \mid\mid \mathcal{T}_2$ has Moore semantics as well. If $\mathcal{T}_1$ and $\mathcal{T}_2$ are deterministic, then so is $\mathcal{T}_1 \mid\mid \mathcal{T}_2$. If $\mathcal{T}_1$ and $\mathcal{T}_2$ are complete, then $\mathcal{T}_1 \mid\mid \mathcal{T}_2$ is transition-complete. If $\mathcal{T}_1$ and $\mathcal{T}_2$ are labeling-complete and $O_1 \cap O_2 = \emptyset$ holds, then $\mathcal{T}_1 \mid\mid \mathcal{T}_2$ is labeling-complete as well.*

Proof. Let $\mathcal{T}_1 = (T_1, T_{1,0}, \tau_1, \ell_1)$, $\mathcal{T}_2 = (T_2, T_{2,0}, \tau_2, \ell_2)$, and $\mathcal{T}_1 \mid\mid \mathcal{T}_2 = (T, T_0, \tau, \ell)$. For the sake of readability, let $I = (I_1 \cup I_2) \setminus (O_1 \cup O_2)$ and let $O = O_1 \cup O_2$. Since both $\mathcal{T}_1$ and $\mathcal{T}_2$ have Moore semantics by assumption, it follows immediately from the definition of the labeling relation ℓ that $\mathcal{T}$ has Moore semantics as well.

First, let both $\mathcal{T}_1$ and $\mathcal{T}_2$ be deterministic. Then, we have $|T_{1,0}| \leq 1$ and $|T_{2,0}| \leq 1$ and thus, by definition of transducer composition, $|T_0| \leq 1$ holds. Let $(u, v), (u', v') \in T$ and let $\iota \in 2^I$ such that $((u, v), \iota, (u', v')) \in \tau$ holds. Then, by construction of τ, there exist $o_1 \in 2^{O_1}$ and $o_2 \subseteq 2^{O_2}$ with $(u, \iota_1, o_1) \in \ell_1$ and $(v, \iota_2, o_2) \in \ell_1$ such that $(u, \iota_1, u') \in \tau_1$ and $(v, \iota_2, v') \in \tau_2$ hold, where $\iota_1 = (\iota \cup o_2) \cap I_1$ and $\iota_2 = (\iota \cup o_1) \cap I_2$. Since $\mathcal{T}_1$ and $\mathcal{T}_2$ are deterministic by assumption and thus, in particular, transition-deterministic, $u' \in T_1$ and $v' \in T_2$ are the only successor states of u and v in $\mathcal{T}_1$ and $\mathcal{T}_2$ for input ι_1 and ι_2, respectively. Since $\mathcal{T}_1$ and $\mathcal{T}_2$ have Moore semantics by assumption, their labeling relations are independent of the input. Since they are labeling-deterministic, the labeling relations assign only a single valuation of output variables to the respective state. Hence, o_1 and o_2 are the only valuations of output variables that can satisfy $(u, \iota_1, o_1) \in \ell_1$ and $(v, \iota_2, o_2) \in \ell_1$. Therefore, ι_1 and ι_2 are unique for (u, v) and ι and hence $(u', v') \in T$ is the only successor state of (u, v) for ι in $\mathcal{T}$. Thus, $\mathcal{T}$ is deterministic.

Second, let both $\mathcal{T}_1$ and $\mathcal{T}_2$ be labeling-complete. Then, we have $|T_{1,0}| \geq 1$ and $|T_{2,0}| \geq 1$ and thus, by definition of transducer composition, $|T_0| \geq 1$ holds. Let $(u, v) \in T$ and let $\iota \in 2^I$. Since both $\mathcal{T}_1$ and $\mathcal{T}_2$ are labeling-complete and since they have Moore semantics by assumption, there are outputs $o_1 \in 2^{O_1}$ and $o_2 \in 2^{O_2}$ such that $(u, o_1) \in \ell_1$ and $(v, o_2) \in \ell_2$ holds. If $O_1 \cap O_2 = \emptyset$ holds, then clearly $o_1 \cup o_2$ is well-defined, i.e., it is non-contradictory, and thus, by definition of transducer composition, we have $((u, v), \iota, o_1 \cup o_2) \in \ell$ for all $\iota \in 2^I$. Hence, $\mathcal{T}$

is then labeling-complete as well. If both $\mathcal{T}_1$ and $\mathcal{T}_2$ are, in addition to labeling-completeness, also transition-complete, then there exist transitions $(u, \iota_1, u') \in \tau_1$ and $(v, \iota_2, v') \in \tau_2$, where $\iota_1 = (\iota \cup o_2) \cap I_1$ and $\iota_2 = (\iota \cup o_1) \cap I_2$. Thus, by definition of transducer composition, there exists a state $(u', v') \in T$ such that $((u, v), \iota, (u', v')) \in \tau$ holds. Hence, $\mathcal{T}$ is then transition-complete as well, even if $O_1 \cap O_2 \neq \emptyset$ holds. □

Furthermore, if the parallel composition $\mathcal{T}_1 \mid\mid \mathcal{T}_2$ of two complete finite-state transducers $\mathcal{T}_1$ and $\mathcal{T}_2$ produces infinite traces only, then its traces $\sigma \in \mathit{Traces}(\mathcal{T}_1 \mid\mid \mathcal{T}_2)$ are, restricted to the respective variables, traces of both $\mathcal{T}_1$ and $\mathcal{T}_2$ as well:

Lemma 2.2. *Let I_1, I_2, O_1, and O_2 be finite sets with $I_1 \cap O_1 = \emptyset$ and $I_2 \cap O_2 = \emptyset$. Let $\mathcal{T}_1$ be a complete finite-state $(2^{I_1}, 2^{O_1})$-transducer and let $\mathcal{T}_2$ be a complete finite-state $(2^{I_2}, 2^{O_2})$-transducer. Let $V_1 = I_1 \cup O_1$, $V_2 = I_2 \cup O_2$, and $V = V_1 \cup V_2$. If all traces of $\mathcal{T}_1 \mid\mid \mathcal{T}_2$ are infinite, then we have*

$$\mathit{Traces}(\mathcal{T}_1 \mid\mid \mathcal{T}_2) = \left\{\sigma \in (2^V)^\omega \mid \sigma \cap V_1 \in \mathit{Traces}(\mathcal{T}_1) \wedge \sigma \cap V_2 \in \mathit{Traces}(\mathcal{T}_2)\right\}.$$

Proof. Let $\mathcal{T}_1 = (T_1, T_{1,0}, \tau_1, \ell_1)$, $\mathcal{T}_2 = (T_2, T_{2,0}, \tau_2, \ell_2)$, and $\mathcal{T}_1 \mid\mid \mathcal{T}_2 = (T, T_0, \tau, \ell)$. First, let $\sigma \in (2^V)^\omega$ be an infinite sequence such that both $\sigma \cap V_1 \in \mathit{Traces}(\mathcal{T}_1)$ and $\sigma \cap V_2 \in \mathit{Traces}(\mathcal{T}_2)$ hold. Let $\gamma = \sigma \cap ((I_1 \cup I_2) \setminus (O_1 \cup O_2))$, let $\gamma^1 = \sigma \cap I_1$, and let $\gamma^2 = \sigma \cap I_2$. Then, since both transducers $\mathcal{T}_1$ and $\mathcal{T}_2$ are complete by assumption, it holds that, for all $i \in \{1, 2\}$, transducer $\mathcal{T}_i$ produces an infinite path $\pi^i \in \mathit{Paths}(\mathcal{T}_i, \gamma^i)$ for input sequence γ^i such that $\#_2(\pi^i_k) = \sigma_k \cap O_i$ holds for all points in time $k \geq 0$. Based on the paths π^1 and π^2, we construct an infinite sequence $\pi \in (T, 2^{O_1 \cup O_2})^\omega$ of pairs of states of $\mathcal{T}_1 \mid\mid \mathcal{T}_2$ and output valuations as follows:

$$\pi_k = ((\#_1(\pi^1_k)), \#_1(\pi^2_k)), \#_2(\pi^1_k) \cup \#_2(\pi^2_k)) \qquad \text{for all } k \geq 0$$

Since $\#_2(\pi^i_k) = \sigma_k \cap O_i$ holds for all $i \in \{1, 2\}$ and all $k \geq 0$, the union of the outputs of $\mathcal{T}_1$ and $\mathcal{T}_2$ is well-defined as no conflicts can occur. Thus, π is well-defined. We first show that $\pi \in \mathit{Paths}(\mathcal{T}_1 \mid\mid \mathcal{T}_2, \gamma)$ holds, i.e., that we have $\#_1(\pi_0) \in T_0$ and $(\#_1(\pi_k), \gamma_k, \#_1(\pi_{k+1})) \in \tau$ as well as $(\#_1(\pi_k), \gamma_k, \#_2(\pi_k)) \in \ell$ for all points in time $k \geq 0$. By assumption, both $\pi^1 \in \mathit{Paths}(\mathcal{T}_1, \gamma^1)$ and $\pi^2 \in \mathit{Paths}(\mathcal{T}_2, \gamma^2)$ hold and hence we have $\#_1(\pi^i_0) \in T^i_0$ as well as both $(\#_1(\pi^i_k), \gamma^i_k, \#_1(\pi^i_{k+1})) \in \tau_i$ and $(\#_1(\pi^i_k), \gamma^i_k, \#_2(\pi^i_k)) \in \ell_i$ for all $i \in \{1, 2\}$ and all $k \geq 0$. Thus, by construction of π, in particular $\#_1(\pi_0) \in T_0$ holds. Since we have $\#_2(\pi^i_k) = \sigma_k \cap O_i$ for all $i \in \{1, 2\}$ and all $k \geq 0$, it follows from the construction of γ that $\gamma_k \cup \#_2(\pi^{3-i}_k) = \sigma_k \cap (O_{3-i} \cup ((I_1 \cup I_2) \setminus O_i))$ holds for all $i \in \{1, 2\}$ and all $k \geq 0$. Thus, we have $(\gamma_k \cup \#_2(\pi^{3-i}_k)) \cap I_i = (\sigma_k \cap (O_{3-i} \cup (I_1 \cup I_2) \setminus O_i))) \cap I_i$. Since $I_i \cap O_i = \emptyset$ holds by assumption for all $i \in \{1, 2\}$, we have $((I_1 \cup I_2) \setminus O_i) \cap I_i = (I_1 \cup I_2) \cap I_i = I_i$ and hence $(O_{3-i} \cup ((I_1 \cup I_2) \setminus O_i)) \cap I_i = I_i$ follows. Therefore, $(\gamma_k \cup \#_2(\pi^{3-i}_k)) \cap I_i = \gamma^i$ holds for all $i \in \{1, 2\}$ and all $k \geq 0$. Hence, we have $(\#_1(\pi^1_k), \#_1(\pi^2_k)), \gamma_k, (\#_1(\pi^1_{k+1}), \#_1(\pi^2_{k+1}))) \in \tau$ and $(\#_1(\pi^1_k), \#_1(\pi^2_k)), \gamma_k, \#_2(\pi^1_k) \cup \#_2(\pi^2_k)) \in \ell$ for all $k \geq 0$ by definition of the parallel composition of finite-state transducers and thus, by construction of π, both $(\#_1(\pi_k), \gamma_k, \#_1(\pi_{k+1})) \in \tau$ and $(\#_1(\pi_k), \gamma_k, \#_2(\pi_k)) \in \ell$ follow. Therefore, $\pi \in \mathit{Paths}(\mathcal{T}_1 \mid\mid \mathcal{T}_2, \gamma)$ holds. Moreover, by construction of π, we have $\#_2(\pi_k) = \#_2(\pi^1_k) \cup \#_2(\pi^2_k)$ for all $k \geq 0$ and thus, since $\#_2(\pi^i_k) = \sigma_k \cap O_i$ holds, we have $\#_2(\pi_k) = \sigma_k \cap (O_1 \cup O_2)$ for all $k \geq 0$. Since we have $\gamma = \sigma \cap ((I_1 \cup I_2) \setminus (O_1 \cup O_2))$ by definition, $\gamma_k \cup \#_2(\pi_k) = \sigma_k \cap V$ follows for all $k \geq 0$. Hence, since $\sigma \in (2^V)^\omega$ holds and by definition of traces, $\sigma \in \mathit{Traces}(\mathcal{T}_1 \mid\mid \mathcal{T}_2, \gamma)$ follows.

Second, let $\sigma \in Traces(\mathcal{T}_1 \mid\mid \mathcal{T}_2)$ be a trace of the parallel composition $\mathcal{T}_1 \mid\mid \mathcal{T}_2$ of $\mathcal{T}_1$ and $\mathcal{T}_2$. Let $\gamma = \sigma \cap ((I_1 \cup I_2) \setminus (O_1 \cup O_2))$, let $\gamma^1 = \sigma \cap I_1$, and let $\gamma^2 = \sigma \cap I_2$. Then, since all traces of $\mathcal{T}_1 \mid\mid \mathcal{T}_2$ are infinite by assumption, $\mathcal{T}_1 \mid\mid \mathcal{T}_2$ produces an infinite path $\pi \in Paths(\mathcal{T}_1 \mid\mid \mathcal{T}_2, \gamma)$ for input sequence γ such that $\#_2(\pi_k) = \sigma_k \cap (O_1 \cup O_2)$ holds for all points in time $k \geq 0$. We construct infinite sequences $\pi^1 \in (T_1, 2^{O_1})^\omega$ and $\pi^2 \in (T_2, 2^{O_2})^\omega$ from π as follows:

$$\begin{aligned} \pi^1_k &= (\#_1(\#_1(\pi_k)), \#_2(\pi_k) \cap O_1) && \text{for all } k \geq 0 \\ \pi^2_k &= (\#_2(\#_1(\pi_k)), \#_2(\pi_k) \cap O_2) && \text{for all } k \geq 0 \end{aligned}$$

We first show that $\pi^i \in Paths(\mathcal{T}_i, \gamma^i)$ holds for all $i \in \{1, 2\}$. Since $\pi \in Paths(\mathcal{T}_1 \mid\mid \mathcal{T}_2, \gamma)$, we have $\#_1(\pi_0) \in T_{1,0} \times T_{2,0}$ and thus $\#_1(\pi^i_0) \in T_{i,0}$ follows for all $i \in \{1, 2\}$ with the construction of π^i. Hence, it remains to show that, for all $i \in \{1, 2\}$, we have both $(\#_1(\pi^i_k), \gamma^i_k, \#_1(\pi^i_{k+1})) \in \tau_i$ and $(\#_1(\pi^i_k), \gamma^i_k, \#_2(\pi^i_k)) \in \ell_i$ for all points in time $k \geq 0$. Since $\pi \in Paths(\mathcal{T}_1 \mid\mid \mathcal{T}_2, \gamma)$ holds by assumption, we have both $(\#_1(\pi_k), \gamma_k, \#_1(\pi_{k+1})) \in \tau$ and $(\#_1(\pi_k), \gamma_k, \#_2(\pi_k)) \in \ell$ for all $k \geq 0$. Therefore, for all $k \geq 0$, there exist outputs $o_1 \subseteq O_1$ and $o_2 \subseteq O_2$ such that $o_1 \cup o_2 = \#_2(\pi_k)$ and both $(\#_1(\#_1(\pi_k)), (\gamma_k \cup o_{3-i}) \cap I_i, \#_1(\#_1(\pi_{k+1}))) \in \tau_i$ and $(\#_1(\#_1(\pi_k)), (\gamma_k \cup o_{3-i}) \cap I_i, o_i) \in \ell_i$ hold. Since $o_1 \cup o_2 = \#_2(\pi_k)$ holds, we clearly have $o_i = \#_2(\pi_k) \cap O_i$ for all $i \in \{1, 2\}$. Furthermore, by construction of π, we have $\#_2(\pi_k) = \sigma_k \cap (O_1 \cup O_2)$ and thus $o_i = \sigma_k \cap O_i$ follows. Hence, we have $\gamma_k \cup o_{3-i} = \sigma_k \cap (O_{3-i} \cup ((I_1 \cup I_2) \setminus O_i))$ by construction of γ. Thus, in particular, $(\gamma_k \cup o_{3-i}) \cap I_i = (\sigma_k \cap (O_{3-i} \cup ((I_1 \cup I_2) \setminus O_i))) \cap I_i$ holds. Since $I_i \cap O_i = \emptyset$ holds by assumption, we have $(I_1 \cup I_2) \setminus O_i) \cap I_i = (I_1 \cup I_2) \cap I_i = I_i$ and hence $(O_{3-i} \cup ((I_1 \cup I_2) \setminus O_i)) \cap I_i = I_i$ follows. Therefore, $(\gamma_k \cup o_{3-i}) \cap I_i = \gamma^i_k$ holds for all $i \in \{1, 2\}$ and all $k \geq 0$. Hence, both $(\#_1(\#_1(\pi_k)), \gamma^i_k, \#_1(\#_1(\pi_{k+1}))) \in \tau_i$ and $(\#_1(\#_1(\pi_k)), \gamma^i_k, \#_2(\pi^i_k)) \in \ell_i$ follow with the construction of the parallel composition of finite-state transducers for all $i \in \{1, 2\}$ and all $k \geq 0$. Thus, $\pi^i \in Paths(\mathcal{T}_i, \gamma^i)$ holds for all $i \in \{1, 2\}$. Moreover, by construction of the sequence π^i, we have $\#_2(\pi^i_k) = \#_2(\pi_k) \cap O_i$ and thus $\#_2(\pi^i_k) = (\sigma_i \cap (O_1 \cup O_2)) \cap O_i = \sigma_k \cap O_i$ follows for all $i \in \{1, 2\}$ and all $k \geq 0$. Since $\gamma^i = \sigma \cap I_i$ holds by definition, we thus have $\gamma^i_k \cup \#_2(\pi^i_k) = \sigma_k \cap V_i$ for all $k \geq 0$. Therefore, $\sigma \cap V_i \in Traces(\mathcal{T}_i, \gamma^i)$ follows for all $i \in \{1, 2\}$. □

2.6.2. System Strategies

While system models define *all* possible behaviors of a reactive system, a *system strategy* defines a concrete behavior of the system. Hence, it characterizes how the system concretely reacts to an input sequence. Therefore, in contrast to system models, system strategies are both *complete* and *deterministic*. That is, for every input sequence, there exists exactly one output sequence. Formally, we define a system strategy as follows:

Definition 2.13 (System Strategy and Computation).
Let I and O be finite sets of input and output variables with $I \cap O = \emptyset$. A *system strategy* is a function $s : (2^{I \cup O})^* \times 2^I \rightarrow 2^O$. The *computation* of strategy s on an infinite input sequence $\gamma \in (2^I)^\omega$, denoted $comp(s, \gamma)$, is the infinite word $\sigma \in (2^{I \cup O})^\omega$ with both $\sigma \cap I = \gamma$ and $s(\sigma_0 \ldots \sigma_k, \sigma_{k+1} \cap I) = \sigma_{k+1} \cap O$ for all points in time $k \geq 0$.

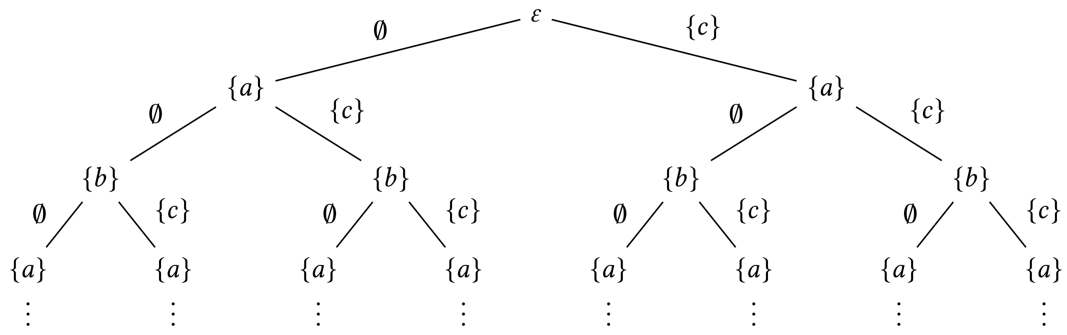


Figure 2.5.: Strategy tree for a strategy $s : (2^{I \cup O})^* \times 2^I \longrightarrow 2^O$ with $I = \{c\}$ and $O = \{a, b\}$ that alternates between outputting a and b, irrespective of the input. For ease of presentation we denote every node of the tree with its label.

Intuitively, a system strategy thus maps a history of valuations of input and output variables and the current input valuation to a valuation of output variables. Hence, the behavior of a system strategy $s : (2^{I \cup O})^* \times 2^I \longrightarrow 2^O$ is characterized by an infinite 2^O-labeled tree $(\mathbb{T}, \ell)$. It branches according to the valuations of I and its nodes $x \in \mathbb{T}$ are labeled with the strategic choice of s on x. An exemplary strategy tree for a strategy that alternates between outputting a and b, irrespective of the input c, is depicted in Figure 2.5.

When reading an infinite input sequence $\gamma \in (2^I)^\omega$, a strategy $s : (2^{I \cup O})^* \times 2^I \longrightarrow 2^O$ produces a unique infinite output sequence characterizing the system's behavior: the computation $\mathit{comp}(s, \gamma)$ of s on γ. The computations of a system strategy then define whether the strategy complies with a system specification given as a linear-time property:

Definition 2.14 (Specification Realization)**.**
Let V be a finite set of variables. Let $I \subseteq V$ and $O \subseteq V$ be finite sets of input and output variables with $I \cap O = \emptyset$. Let $s : (2^{I \cup O})^* \times 2^I \longrightarrow 2^O$ be a system strategy and let $L \subseteq (2^V)^\omega$ be a linear-time property. Then, s *realizes* L, denoted $s \models L$, if, and only if, $\mathit{comp}(s, \gamma) \cup \gamma' \in L$ holds for all $\gamma \in (2^I)^\omega$ and all $\gamma' \in (2^{V \setminus (I \cup O)})^\omega$.

Given a set V of variables and sets $I \subseteq V$ and $O \subseteq V$ of input and output variables with $I \cap O = \emptyset$, we call a linear-time property $L \subseteq (2^V)^\omega$ *realizable* if there exists some system strategy $s : (2^{I \cup O})^* \times 2^I \longrightarrow 2^O$ that realizes L. Overloading notation, we say that a system strategy s realizes an LTL formula φ instead of saying that s realizes φ's language $\mathcal{L}(\varphi)$. Throughout this thesis, we call system strategies simply *strategies* when the context is clear.

It is well-known that whenever there exists a system strategy that realizes an LTL formula, then there also exists one that is finitely representable [EJ91]. Since we only consider system requirements formalized in LTL in this thesis, we can thus always assume that there exist finitely representable strategies for realizable system objectives. A finite representation of a system strategy is called *implementation*. For simplicity, however, we use the terms strategy and implementation interchangeably when the context is clear.

In this thesis, we model system implementations as finite-state transducers. We also call transducers that represent system strategies *strategy transducers*. Let I and O be finite sets of inputs and outputs. A finite-state $(2^I, 2^O)$-transducer $\mathcal{T}$ that represents a system strategy $s : (2^{I \cup O})^* \times 2^I \rightarrow 2^O$ produces exactly the computations of s. Since a system strategy produces for every infinite word a unique infinite output word, a finite-state transducer that represents a system strategy needs to produce a unique trace and thus also a unique path for every infinite input word as well. Therefore, the transducer needs to be both *deterministic* and *complete* as it can otherwise produce multiple path – if it is nondeterministic – or no path at all – if it is incomplete. The unique path produced by a $\mathcal{T}$ on input sequence $\gamma \in (2^I)^\omega$ then needs to coincides with the computation of s on γ, i.e., with $comp(s, \gamma)$. Hence, for a transducer $\mathcal{T}$ to represent a system strategy s, we require that $Traces(\mathcal{T}, \gamma) = \{comp(s, \gamma)\}$ holds for all $\gamma \in (2^I)^\omega$. Therefore, it follows immediately from Definition 2.14, that a finite-state $(2^I, 2^O)$-transducer $\mathcal{T}$ *realizes* a linear-time property $L \in (2^V)^\omega$, where $I \cup O \subseteq V$, if it is deterministic and complete and if $Traces(\mathcal{T}) \cup (2^{V \setminus (I \cup O)})^\omega \subseteq L$ holds.

Note that we defined system strategies irrespective of the architecture of the system. For a monolithic architecture $\mathcal{A} = (P, V, I, O)$, a system strategy $s : (2^{I_1 \cup O_1})^* \times 2^{I_1} \rightarrow 2^{O_1}$ defines the behavior of the single system process $p_1 \in P^-$. For a distributed architecture $\mathcal{A} = (P, V, I, O)$, we consider strategies $s_i : (2^{I_i \cup O_i})^* \times 2^{I_i} \rightarrow 2^{O_i}$ defining the behavior of all respective system process $p_i \in P^-$. These strategies are also called *process strategies*. Moreover, we consider a system strategy $s : (2^{O_{env} \cup O^-})^* \times 2^{O_{env}} \rightarrow 2^{O^-}$ for the entire system, i.e., for the interplay of all system processes. We denote the parallel composition of two system strategies s_1 and s_2 with $s_1 \mid\mid s_2$ and define it in terms of the underlying finite-state transducers, i.e., $s_1 \mid\mid s_2$ is represented by $\mathcal{T}_1 \mid\mid \mathcal{T}_2$, where $\mathcal{T}_1$ and $\mathcal{T}_2$ are finite-state transducers representing s_1 and s_2, respectively.

2.6.3. Winning and Dominant System Strategies

In this thesis, we consider two types of system strategies: *winning* strategies and *remorsefree dominant* – or simply dominant – strategies. Given an LTL formula φ, a strategy is called winning for φ if it realizes the linear-time property $\mathcal{L}(\varphi)$:

Definition 2.15 (Winning Strategy).
Let V be a finite set of variables. Let $I \subseteq V$ and $O \subseteq V$ be finite sets of input and output variables with $I \cap O = \emptyset$. Let φ be an LTL formula over atomic propositions V and let $s : (2^{I \cup O})^* \times 2^I \rightarrow 2^O$ be a strategy. Then, s is *winning for* φ, if, and only if, $s \models \varphi$ holds.

Thus, a winning strategy s is required to satisfy the specification φ for every input sequence. Hence, an LTL formula is realizable if, and only if, there exists a winning strategy for it. In most settings, winning strategies are considered.

Remorsefree dominance [DF11] is a weaker requirement than winning. In contrast to winning strategies, remorsefree dominant strategies are allowed to violate the specification for an input sequence if no other strategy would have satisfied it in the same situation. In the remainder of this thesis, we call remorsefree dominant strategies also dominant strategies whenever the context is clear. Formally, remorsefree dominant strategies are defined as follows:

Definition 2.16 (Dominant Strategy [DF14]).
Let V be a finite set of variables. Let $I \subseteq V$ and $O \subseteq V$ be finite sets of input and output variables with $I \cap O = \emptyset$. Let φ be an LTL formula over atomic propositions V and let $s : (2^{I \cup O})^* \times 2^I \rightarrow 2^O$ be a strategy. A strategy $t : (2^{I \cup O})^* \times 2^I \rightarrow 2^O$ *is dominated by* s, denoted $t \preceq s$, if, and only if, for all $\gamma \in (2^{I_i})^\omega$ and all $\gamma' \in (2^{V \setminus (I \cup O)})^\omega$ either $comp(s, \gamma) \cup \gamma' \models \varphi$ or $comp(t, \gamma) \cup \gamma' \not\models \varphi$ holds. Strategy s is called *dominant* for φ if $t \preceq s$ holds for all alternative strategies $t : (2^{I \cup O})^* \times 2^I \rightarrow 2^O$.

Intuitively, a strategy s dominates a strategy t if it is "at least as good" as t. It is dominant for φ if it is at least as good as *every other possible strategy* and thus if it is "as good as possible". Dominance is, therefore, a notion of *best effort*: if a strategy s fails to satisfy the specification but there does not exists any better strategy, then s did it's best and thus should not feel any remorse concerning its behavior. A dominant strategy also dominates other dominant strategies, particularly itself. An LTL formula is called *admissible* if a dominant strategy exists for it.

Example 2.6. Consider the LTL formula $\varphi = \diamondsuit a \wedge \diamondsuit b$. Let $I = \{a\}$ and $O = \{b\}$. There does not exist a winning strategy s for φ: irrespective of the sequence of output valuations produced by s, the computation of s on some input sequence $\gamma \in (2^I)^\omega$ accurately reflects the valuations of input variables defined by γ. Thus, in particular, for an input sequence $\gamma \in (2^I)^\omega$ that does not set a to *true* at any point in time, i.e., with $a \notin \gamma_k$ for all $k \geq 0$, the computation $comp(s, \gamma)$ of any strategy s contains no a. Hence, for such input sequences, $comp(s, \gamma)$ violates φ irrespective of the choice of the strategy s. However, there exist dominant strategies for φ, for instance a strategy s that sets b to *true* in the very first time step: for input sequences that do not contain any a, all strategies violate φ as outlined above. For all other input sequences, the computation of s clearly satisfies φ. △

Every strategy that is winning for an LTL specification φ is clearly also remorsefree dominant for φ. Hence, if φ is realizable, then there exists a dominant strategy s for φ whose computation satisfies φ for every input sequence. Since every other dominant strategy t for φ needs to be at least as good as s, its computation thus needs to satisfy φ for every input sequence as well. Therefore, every dominant strategy for φ is winning for φ if φ is realizable:

Proposition 2.5 ([DF14]). *Let V be a finite set of variables. Let $I \subseteq V$ and $O \subseteq V$ be finite sets of input and output variables with $I \cap O = \emptyset$. Let φ be an LTL formula over atomic propositions V. If φ is realizable, then every dominant strategy $s : (2^{I \cup O})^* \times 2^I \rightarrow 2^O$ is winning for φ.*

2.7. Infinite Games

Games, particularly two-player games, are a common model in computer science. Since we consider reactive systems in this thesis and thus systems that do not terminate, we utilize *infinite* games, i.e., games that are played indefinitely. Infinite games can, for instance, be used for solving the reactive synthesis problem. An infinite game is played in an *arena*, which is represented by a graph whose set of vertices, the so-called positions, is partitioned into the positions of the two players Player 0 and Player 1. Formally:

Definition 2.17 (Game Arena)**.**
A game arena is a tuple $\mathbb{A} = (P, P_0, P_1, v_0, E)$, where P, P_0, P_1 are sets of positions with $P = P_0 \cup P_1$ and $P_0 \cap P_1 = \emptyset$, $v_0 \in P$ is the initial position, $E \subseteq P \times P$ is a set of edges such that for all positions $v \in P$, there exists a position $v' \in P$ such that $(v, v') \in E$ holds. Player i controls the positions in P_i.

In an arena $\mathbb{A} = (P, P_0, P_1, v_0, E)$, the players construct a *play*. A play is an infinite sequence $\rho \in P^\omega$ of positions such that $(\rho_k, \rho_{k+1}) \in E$ holds for all $k \geq 0$. The player owning a position chooses the edge on which the play is continued. That is, if a position $v \in P_i$, which is controlled by Player i and which is reached at point in time $k \geq 0$ in a play $\rho \in P^\omega$, has multiple outgoing edges $(v, v'), (v, v'') \in E$, then Player i chooses whether ρ continues with v' or v'', i.e., whether $\rho_{k+1} = v'$ or $\rho_{k+1} = v''$ holds. We call a play *initial* if it starts in the initial position, i.e., if $\rho_0 = v_0$ holds for the play $\rho \in P^\omega$. Note that since we require every position of an arena to have at least one outgoing edge, a play is guaranteed to be infinitely long. Based on game arenas and plays, an infinite game is defined as follows:

Definition 2.18 (Infinite Game)**.**
An *infinite game* $\mathbb{G} = (\mathbb{A}, \mathbb{W})$ consists of a game arena $\mathbb{A} = (P, P_0, P_1, v_0, E)$ and a winning condition $\mathbb{W} \in P^\omega$. A play $\rho \in P^\omega$ in $\mathbb{A}$ is *winning for Player 0* if $\rho \in \mathbb{W}$ holds and *winning for Player 1* otherwise.

Given a game $\mathbb{G} = (\mathbb{A}, \mathbb{W})$ with arena $\mathbb{A} = (P, P_0, P_1, v_0, E)$, a *strategy* for Player i intuitively defines the decisions Player i makes during a play. Formally, a strategy for Player i is a function $\mu : P^* \times P_i \to P$ such that for all positions $v \in P_i$ and all finite sequences $\nu \in P^*$ of positions, whenever $\mu(\nu, v) = v'$ holds, then we have $(v, v') \in E$. A play $\rho \in P^*$ is *consistent* with a player's strategy μ if, and only if, for all points in time $k \geq 0$, it holds that whenever we have $\rho_k \in P_i$, then $\rho_{k+1} = \mu(\rho_{|k}, \rho_k)$ holds. We denote the set of all plays that start in position v and that are consistent with μ with $\mathit{Plays}(\mathbb{G}, \mu, v)$. The set of initial plays that are consistent with μ, i.e., $\mathit{Plays}(\mathbb{G}, \mu, v_0)$, is also denoted with $\mathit{Plays}(\mathbb{G}, \mu)$. Note that, given a strategy μ for Player 0 and a strategy μ' for Player 1, there is a unique initial play that is consistent with both μ and μ', i.e., we have $|\mathit{Plays}(\mathbb{G}, \mu) \cap \mathit{Plays}(\mathbb{G}, \mu')| = 1$. A strategy for Player i is *winning* if, and only if, all initial and consistent plays are winning for Player i. Hence, for a winning strategy $\mu : P^* \times P_0 \to P$ for Player 0, we have $\rho \in \mathbb{W}$ for all plays $\rho \in \mathit{Plays}(\mathbb{G}, \mu)$. For a winning strategy $\mu : P^* \times P_1 \to P$ for Player 1, in contrast, we have $\rho \notin \mathbb{W}$ for all $\rho \in \mathit{Plays}(\mathbb{G}, \mu)$.

Example 2.7. Consider the game arena $\mathbb{A}$ depicted in Figure 2.6. Positions controlled by Player 1 are depicted as rectangles, positions with rounded edges are controlled by Player 0. Let $\mathbb{W}$ be a winning condition that states that the positions v_5 and v_7 should never be visited. These states are highlighted in violet. A winning strategy for Player 0 is highlighted in blue. It enforces every initial consistent play to reach v_3 without visiting v_5 or v_7 beforehand. In v_3, Player 1 does not have any choice other than moving to v_2. Position v_2, however, is controlled by Player 0 and the strategy described above enforces that a consistent play moves back to v_3. Hence, every initial consistent play loops between v_2 and v_3 forever while not visiting v_5 or v_6 before and therefore the strategy is winning. △

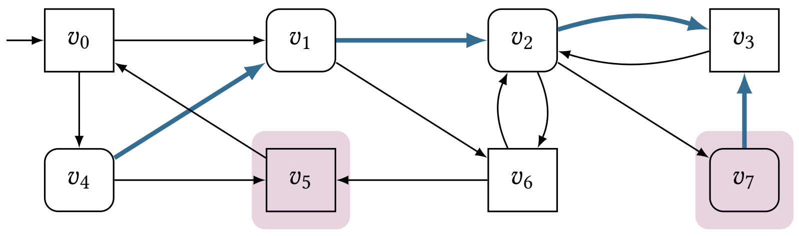


Figure 2.6.: A game arena $\mathbb{A}$. Positions controlled by Player 1 are depicted as rectangles, positions with rounded edges are controlled by Player 0. Positions v_5 and v_7, highlighted in violet, should be avoided. A winning strategy for Player 0 is depicted in blue.

2.8. Reactive Synthesis

Intuitively, reactive synthesis [Chu57] is the task of automatically deriving a correct-by-construction implementation for a reactive system from a formal specification. In this thesis, we only consider formal specification given in LTL. Furthermore, we consider both monolithic and distributed systems. Therefore, we formalize the reactive synthesis problem as follows.

Definition 2.19 (Reactive Synthesis Problem).
Let $\mathcal{A} = (P, V, I, O)$ be an architecture. Let φ be an LTL formula over atomic propositions V. The *reactive synthesis problem* is to derive system strategies $s_1, \dots, s_n$ for the system processes $p_1, \dots, p_n \in P^-$ such that $s_1 \mid\mid \dots \mid\mid s_n \models \varphi$ holds.

For monolithic architectures, the reactive synthesis problem for LTL specifications can be solved by first translating the LTL formula into a nondeterministic Büchi automaton. After determinizing the automaton with Safra's construction [Saf88], we can employ the game-based [BL69] or the automaton-based [Rab72] synthesis approach. In terms of complexity, synthesis results in a doubly-exponential running time. In fact, the reactive synthesis problem for monolithic architectures and LTL specifications is 2EXPTIME-complete:

Theorem 2.1 ([PR89a]). *Let $\mathcal{A} = (P, V, I, O)$ be a monolithic architecture. Let φ be an LTL formula over atomic propositions V. The question whether there exists a system strategy s for the single system process such that $s \models \varphi$ holds is 2EXPTIME-complete.*

In this thesis, we focus on *safraless* [KV05] synthesis approaches, which avoid Safra's construction for determinizing the nondeterministic Büchi automaton. In particular, we consider *bounded synthesis* [FS13], on which we elaborate in the following section.

2.8.1. Bounded Synthesis

Bounded synthesis [FS13] is a synthesis procedure for monolithic systems that derives size-optimal strategies from LTL specifications. It constructs a finite-state transducer that realizes the specification with a minimal number of states. To do so, bounded synthesis bounds the

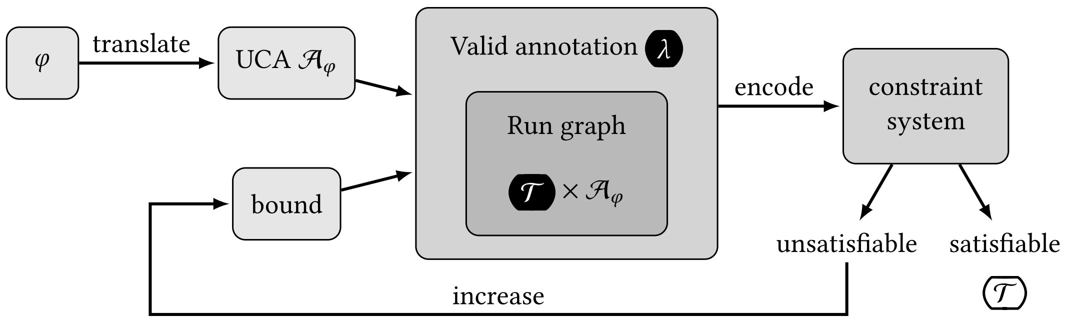


Figure 2.7.: Workflow of constraint-based bounded synthesis.

number of states of the desired transducer. Starting from bound 1, the bound is successively increased if there does not exist a transducer of the specified size that realizes the specification. This procedure is continued until a solution is found. There exists an upper bound on the size of a finite-state transducer realizing the specification [FS13]. Thus, bounded synthesis is guaranteed to terminate. If there does not exist a transducer of the size of the upper bound that realizes the specification, then the specification is unrealizable. Bounded synthesis is, for instance, implemented in the tools Unbeast [Ehl11], Acacia+ [BBF⁺12], and BoSy [FFT17].

Constraint-based bounded synthesis reduces bounded synthesis to a constraint-solving problem. In particular, it encodes the existence of a finite-state transducer of the specified size that realizes the specification into a constraint system. There exist SMT, SAT, QBF, and DQBF encodings [FS13, FFRT17]. In the following, we describe the general workflow of constraint-based bounded synthesis. An overview is depicted in Figure 2.7. First, bounded synthesis translates a given LTL specification φ into a universal co-Büchi automaton $\mathcal{A}_\varphi$ of size $\mathcal{O}(2^{|\varphi|})$ with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$ (see Proposition 2.4). Recall that a finite-state transducer $\mathcal{T}$ representing a system strategy s realizes φ if, and only if, every trace generated by $\mathcal{T}$ lies in the language of φ. Thus, since $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$ holds, it follows from the definition of the acceptance of universal co-Büchi automata that $\mathcal{T}$ realizes φ if, and only if, every trace generated by $\mathcal{T}$ induces only runs of $\mathcal{A}_\varphi$ with finitely many visits to rejecting states. The runs of $\mathcal{A}_\varphi$ induced by a finite-state transducer $\mathcal{T}$ are captured by the unique *run graph* of $\mathcal{A}_\varphi$ and $\mathcal{T}$:

Definition 2.20 (Run Graph)**.**
Let $\mathscr{A}$ be a monolithic architecture and let I and O be the inputs and outputs of the single process. Let $\mathcal{A} = (Q, q_0, \delta, F)$ be a universal co-Büchi automaton over alphabet $I \cup O$. Let $\mathcal{T} = (T, T_0, \tau, \ell)$ be a deterministic and complete finite-state $(2^I, 2^O)$-transducer. The *run graph* $\mathbb{G} = (\mathcal{V}, \mathcal{E})$ of $\mathcal{A}$ and $\mathcal{T}$, denoted $\mathcal{T} \times \mathcal{A}$, is defined by

- $\mathcal{V} = T \times Q$, the set of vertices, and
- $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$, the edge relation, with $((t,q),(t',q')) \in \mathcal{E}$ if, and only if,

$$\exists \iota \in 2^I . \exists o \in 2^O .\ (t, \iota, t') \in \tau \wedge (t, \iota, o) \in \ell \wedge (q, \iota \cup o, q') \in \delta.$$

A vertex $v = (t, q) \in \mathcal{V}$ of a run graph $\mathbb{G} = (\mathcal{V}, \mathcal{E})$ is *rejecting* if, and only if, $q \in F$, i.e., q is a rejecting state in the universal co-Büchi automaton $\mathcal{A}$. A *run* is a path of $\mathbb{G}$ that starts in the initial vertex (t, q_0) with $t \in T_0$. Note that since $\mathcal{T}$ is deterministic, we have $|T_0| = 1$ and thus the initial vertex of the run graph is unique. We call a run *accepting* if it is either finite or contains only finitely many rejecting vertices. A run graph is *accepting* if every run is accepting.

The run graph of a finite-state transducer $\mathcal{T}$ and a universal co-Büchi automaton $\mathcal{A}$ then captures whether or not $\mathcal{T}$ realizes $\mathcal{A}$'s language:

Lemma 2.3 ([FS13]). *Let $\mathscr{A}$ be a monolithic architecture and let I and O be the inputs and outputs of the single process. Let $\mathcal{A}$ be a universal co-Büchi automaton over alphabet $I \cup O$. Let $\mathcal{T}$ be a deterministic and complete finite-state $(2^I, 2^O)$-transducer. Then the run graph $\mathcal{T} \times \mathcal{A}$ is accepting if, and only if, $\mathcal{T}$ realizes $\mathcal{L}(\mathcal{A})$.*

Hence, we can utilize the run graph of a candidate transducer $\mathcal{T}$ and the universal co-Büchi automaton $\mathcal{A}_\varphi$ with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$ in order to determine whether or not $\mathcal{T}$ realizes φ. More precisely, we annotate the run graph $\mathcal{T} \times \mathcal{A}_\varphi$. An *annotation* is a function $\lambda : \mathcal{V} \rightarrow \mathbb{N} \cup \{\bot\}$, where $\mathcal{V}$ is the set of vertices of $\mathcal{T} \times \mathcal{A}_\varphi$. Hence, λ maps the vertices of the run graph to either unreachable $\bot$ or to a natural number $k \in \mathbb{N}$.

Definition 2.21 (Valid Annotation [FS13]).
Let $\mathscr{A}$ be a monolithic architecture and let I and O be the inputs and outputs of the single process. Let $\mathcal{A} = (Q, q_0, \delta, F)$ be a universal co-Büchi automaton over alphabet $I \cup O$ and let $\mathcal{T} = (T, T_0, \tau, \ell)$ be a deterministic and complete finite-state $(2^I, 2^O)$-transducer. Let $\mathbb{G} = (\mathcal{V}, \mathcal{E})$ be the run graph of $\mathcal{T}$ and $\mathcal{A}$ and let $\lambda : \mathcal{V} \rightarrow \mathbb{N} \cup \{\bot\}$ be an annotation of $\mathbb{G}$. Then, λ is *valid* if, and only if,

- the initial vertex of $\mathbb{G}$ is reachable and thus annotated with a natural number, i.e., $\lambda((t, q_0)) \neq \bot$ holds for the single state $t \in T_0$, and
- if a vertex $v \in \mathcal{V}$ is annotated with a natural number, i.e., $\lambda(v) = k \neq \bot$, then every successor vertex $v' \in \mathcal{V}$ with $(v, v') \in \mathcal{E}$ is annotated with a greater natural number $\lambda(v')$, which needs to be strictly greater than k if v' is rejecting, i.e., $\lambda(v') \triangleright_{v'} k$, where

$$\triangleright_{v'} = \begin{cases} > & \text{if } v' \text{ is rejecting} \\ \geq & \text{otherwise.} \end{cases}$$

Intuitively, a valid annotation counts how often a rejecting state is visited. The first requirement of Definition 2.21 ensures that the initial vertex is annotated "reachable". The second requirement ensures that the annotation strictly increases whenever a rejecting vertex is reached. For non-rejecting states, in contrast, the annotation can remain unchanged.

Recall that if there exists a run in the run graph $\mathbb{G}$ that is not accepting, then it contains infinitely many rejecting vertices. Since $\mathbb{G}$ has a finite set of vertices, the run thus contains a cycle with a rejecting vertex. Hence, in a valid annotation, the annotation of this vertex would need to strictly increase whenever it is visited, i.e., an infinite number of times. Therefore, there does not exist a valid annotation of a run graph that is not accepting and hence a valid annotation of the run graph $\mathcal{T} \times \mathcal{A}_\varphi$ serves as a witness of $\mathcal{T}$ realizing φ:

Theorem 2.2 ([FS13])**.** *Let $\mathscr{A}$ be a monolithic architecture and let I and O be the inputs and outputs of the single process. Let $\mathcal{A}$ be a universal co-Büchi automaton over alphabet $I \cup O$. Let $\mathcal{T}$ be a deterministic and complete finite-state $(2^I, 2^O)$-transducer. There is a valid annotation λ of the run graph $\mathcal{T} \times \mathcal{A}$ if, and only if, $\mathcal{T}$ realizes $\mathcal{L}(\mathcal{A})$.*

Hence, the bounded synthesis problem for a concrete bound $b \in \mathbb{N}$ reduces to finding a valid annotation of the run graph of the universal co-Büchi automaton representing the specification and some candidate finite-state transducer with b states. This search can be encoded into a constraint system. If the constraint system is realizable, a deterministic and complete finite-state transducer realizing the specification can be extracted immediately from the solution of the constraint system. Otherwise, the bound is either increased (see Figure 2.7) or, if the upper bound is reached, unrealizability of the specification is deduced.

In the original formulation of bounded synthesis, an encoding of bounded synthesis into an SMT constraint-solving problem has been provided [FS13]. To be able to use state-of-the-art constraint solvers, encodings in related domains have been proposed: SAT, QBF, and DQBF [FFRT17]. The purely propositional SAT encoding expands the universal quantification over the vertices of the run graph and uses binary arithmetic to encode the ordering constraints of the valid annotation. The QBF encoding, also called the input-symbolic encoding, allows for handling the input variables, i.e., the propositions controlled by the environment, symbolically: a universal quantification over the input variables is added, resulting in a exponentially more succinct encoding. Adding further universal quantification over the states of the candidate finite-state transducer as well as of the universal co-Büchi automaton allows for representing both of them symbolically. For this, a DQBF encoding is necessary.

Theorem 2.3 ([FS13, FFRT17])**.** *Let φ be an LTL formula and let $\mathcal{A}$ be a universal co-Büchi automaton with $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\varphi)$. Let $b \in \mathbb{N}$ be a bound. There exist SMT, SAT, QBF, and DQBF constraint systems $\mathbb{C}^{SMT}_{b,\varphi}$, $\mathbb{C}^{SAT}_{b,\varphi}$, $\mathbb{C}^{QBF}_{b,\varphi}$, and $\mathbb{C}^{DQBF}_{b,\varphi}$ respectively, such that $\mathbb{C}^{\mathbb{D}}_{b,\varphi}$ is satisfiable for $\mathbb{D} \in \{SAT, SMT, QBF, DQBF\}$ if, and only if, φ is realizable with a deterministic and complete finite-state transducer with b states.*

In practice, the SAT, QBF, and DQBF encodings all outperform the original SMT encoding. Moreover, the more symbolic encodings, i.e., QBF and DQBF, are perform better than the purely propositional SAT encoding [FFRT17, FFT17, Ten19]. Note that the performance of the DQBF encoding highly depends on the performance of the underlying solver [Ten19]: while the SMT encoding outperforms the DQBF encoding when using iDQ [FKBV14] (as in the experiments performed in [FFRT17, FFT17]), the performance of the DQBF encoding sharply increases when using dCAQE [TR19] instead. Nevertheless, the QBF encoding has shown to be the most performant one in practice [FFRT17, FFT17, Ten19]. For large benchmarks in terms of states of the finite-state transducer and the automaton, however, the DQBF encoding has an advantage over the QBF encoding [Ten19].

The concept of bounded synthesis has been extended to the synthesis of distributed systems [FS07] and respective SAT, QBF, and DQBF encodings have been developed [Bau17].

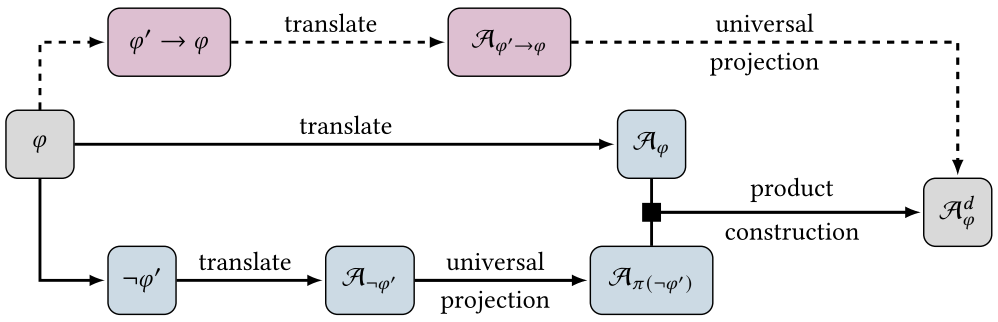


Figure 2.8.: Construction of the automaton $\mathcal{A}^d_{\varphi}$ accepting computations of remorsefree dominant strategies for LTL specification φ. The construction by Damm and Finkbeiner [DF14] is depicted in blue, the one by Steiger [Ste13] in violet and with dashed edges.

Furthermore, for monolithic bounded synthesis, the ranking function used in valid annotations has been extended to Büchi, parity, Rabin, and Streett acceptance conditions, resulting in a bounded synthesis algorithm for CTL* specifications [KB17].

2.8.2. Synthesizing Remorsefree Dominant Strategies

In contrast to winning strategies, remorsefree dominant strategies [DF11] are allowed to violate the specification in situations in which every other strategy would have violated the specification as well (see Section 2.6.3). When synthesizing remorsefree dominant strategies rather than winning once, it is thus crucial to identify such situations. In the following, we describe the extension of bounded synthesis [FS13] to remorsefree dominant strategies.

Recall that constraint-based bounded synthesis relies on encoding the search for a winning strategy, i.e., a strategy whose computation satisfies the given LTL specification for every input sequence, into a constraint system. To do so, bounded synthesis translates the LTL specification into a universal co-Büchi automaton $\mathcal{A}_{\varphi}$ with $\mathcal{L}(\mathcal{A}_{\varphi}) = \mathcal{L}(\varphi)$ and encodes the search for a strategy s such that, for every input sequence $\gamma \in (2^I)^{\omega}$, the runs of $\mathcal{A}_{\varphi}$ induced by $comp(s, \gamma)$ visit only finitely many rejecting states.

To utilize the existing algorithms for this check also for synthesizing remorsefree dominant strategies, we encode the notion of remorsefree dominance into the universal co-Büchi automaton. Hence, we construct a universal co-Büchi automaton $\mathcal{A}^d_{\varphi}$ that recognizes dominant strategies. More precisely, $\mathcal{A}^d_{\varphi}$ recognizes whether the specification φ is satisfied and whether no strategy at all would satisfy φ in the same situation. An overview of the construction of $\mathcal{A}^d_{\varphi}$, which follows [DF11, DF14], is provided in Figure 2.8, highlighted in blue.

First, a universal co-Büchi automaton $\mathcal{A}_{\varphi}$ with $\mathcal{L}(\mathcal{A}_{\varphi}) = \mathcal{L}(\varphi)$ is constructed from φ (see Proposition 2.3). This automaton recognizes situations in which the specification φ is satisfied. Second, we construct a universal co-Büchi automaton identifying situations in which no strategy can satisfy φ as follows. Let φ' be a copy of φ, where every output variable $o \in O$ is replaced by a fresh variable $o' \notin I \cup O$. Intuitively, the primed variables define the outputs of an alternative strategy. We build the automaton $\mathcal{A}_{\neg\varphi'}$ with $\mathcal{L}(\mathcal{A}_{\neg\varphi'}) = \mathcal{L}(\neg\varphi')$ that, intuitively, accepts sequences that define an alternative strategy that violates the specification for the given input sequence. To consider all alternative strategies instead of only a single one, we universally project to the unprimed variables in $\mathcal{A}_{\neg\varphi'}$. Intuitively, the resulting automaton quantifies universally over the primed variables since it always considers both valuations. Formally, the universal projection is defined as follows:

Definition 2.22 (Universal Projection)**.**
Let $\mathcal{A} = (Q, Q_0, \delta, F)$ be a universal co-Büchi automaton over alphabet Σ and let $X \subset \Sigma$. The *universal projection of* $\mathcal{A}$ *to* X is the universal co-Büchi automaton $\pi_X(\mathcal{A}) = (Q, Q_0, \pi_X(\delta), F)$ over alphabet X, where $\pi_X(\delta) = \{(q, a, q') \in Q \times 2^X \times Q \mid \exists b \in 2^{\Sigma \setminus X}. \ (q, a \cup b, q') \in \delta\}$.

Intuitively, the projected automaton $\pi_X(\mathcal{A})$ for a universal co-Büchi automaton $\mathcal{A}$ over alphabet Σ and a set $X \subset \Sigma$ contains the transitions of $\mathcal{A}$ for *all* possible valuations of the variables in $\Sigma \setminus X$. Hence, for a sequence $\sigma \in (2^X)^\omega$, all runs of $\mathcal{A}$ on sequences extending σ with some valuation of the variables in $\Sigma \setminus X$, i.e., sequences $\sigma' \in (2^\Sigma)^\omega$ with $\sigma' \cap X = \sigma$, are also runs of the projected automaton $\pi_X(\mathcal{A})$. Since both $\mathcal{A}$ and $\pi_X(\mathcal{A})$ are universal automata, $\pi_X(\mathcal{A})$ thus accepts a sequence $\sigma \in (2^X)^\omega$ if, and only if, $\mathcal{A}$ accepts all sequences extending σ with some valuation of the variables in $\Sigma \setminus X$:

Lemma 2.4. *Let $\mathcal{A}$ be a universal co-Büchi automaton over alphabet Σ. Let $X \subset \Sigma$ be a set and let $\sigma \in (2^X)^\omega$. Then, $\pi_X(\mathcal{A})$ accepts σ if, and only if $\mathcal{A}$ accepts all $\sigma' \in (2^\Sigma)^\omega$ with $\sigma' \cap X = \sigma$.*

Proof. First, suppose that the projected automaton $\pi_X(\mathcal{A})$ accepts σ. Then, by definition of universal co-Büchi automata, all paths $\pi \in \mathit{Paths}(\pi_X(\mathcal{A}), \sigma)$ of $\pi_X(\mathcal{A})$ induced by σ visit only finitely many rejecting states. Suppose that there is an infinite sequence $\sigma' \in (2^\Sigma)^\omega$ with $\sigma' \cap X = \sigma$ that is rejected by $\mathcal{A}$. Then, there is a path $\pi' \in \mathit{Paths}(\mathcal{A}, \sigma')$ of $\mathcal{A}$ induced by σ' that contains infinitely many rejecting states. By definition of the universal projection and since $\sigma' \cap X = \sigma$ holds, there thus also exists a path $\pi'' \in \mathit{Paths}(\pi_X(\mathcal{A}), \sigma)$ of $\pi_X(\mathcal{A})$ induced by σ such that $\#_1(\pi'_k) = \#_1(\pi''_k)$ as well as $\#_2(\pi'_k) \cap X = \#_2(\pi''_k)$ holds for all points in time $k \geq 0$. Since π' and π'' agree on their first component and thus on the visited states, it follows that π'' contains infinitely many visits to rejecting states as well. Hence, π'' is a path of $\pi_X(\mathcal{A})$ induced by σ with infinitely many visits to rejecting states; contradicting that $\pi_X(\mathcal{A})$ accepts σ.

Second, suppose that $\mathcal{A}$ accepts all sequences $\sigma' \in (2^\Sigma)^\omega$ with $\sigma' \cap X = \sigma$. Then, by definition of universal co-Büchi automata, all paths $\pi \in \mathit{Paths}(\mathcal{A}, \sigma')$ of $\mathcal{A}$ induced by some $\sigma' \in (2^\Sigma)^\omega$ with $\sigma' \cap X = \sigma$ visit rejecting states only finitely often. Suppose that $\pi_X(\mathcal{A})$ rejects σ. Then, there is a path $\pi' \in \mathit{Paths}(\pi_X(\mathcal{A}), \sigma)$ in $\pi_X(\mathcal{A})$ induced by σ that contains infinitely many visits to rejecting states. By definition of the universal projection, there exists some $\sigma' \in (2^\Sigma)^\omega$ with $\sigma' \cap X = \sigma$ that induces a path $\pi'' \in \mathit{Paths}(\mathcal{A}, \sigma')$ in $\mathcal{A}$ such that $\#_1(\pi'_k) = \#_1(\pi''_k)$ as well as $\#_2(\pi'_k) = \#_2(\pi''_k) \cap X$ holds for all points in time $k \geq 0$. Since π' and π'' agree on their first component and thus on the visited states, it follows that π'' contains infinitely many visits to rejecting states as well. Hence, since π'' is induced by σ', the automaton $\mathcal{A}$ rejects σ'; contradicting that $\mathcal{A}$ accepts all sequences $\sigma'' \in (2^\Sigma)^\omega$ with $\sigma'' \cap X = \sigma$. □

We utilize this property to obtain a universal co-Büchi automaton $\mathcal{A}_{\pi(\neg\varphi')}$ from $\mathcal{A}_{\neg\varphi'}$ that considers *all* possible alternative strategies instead of only a particular one: we project to the unprimed variables, i.e., to $I \cup O$, thereby quantifying universally over the alternative strategies. Combining the two automata $\mathcal{A}_\varphi$ and $\mathcal{A}_{\pi(\neg\varphi')}$ in a product construction then yields the desired universal co-Büchi automaton $\mathcal{A}^d_\varphi$ that recognizes remorsefree dominant strategies as the language of the product automaton is the union of the languages of $\mathcal{A}_\varphi$ and $\mathcal{A}_{\pi(\neg\varphi')}$.

Using the resulting automaton $\mathcal{A}^d_\varphi$ instead of the universal co-Büchi automaton $\mathcal{A}_\varphi$ in bounded synthesis then allows for synthesizing remorsefree dominant strategies. By Proposition 2.3, both intermediate automata $\mathcal{A}_\varphi$ and $\mathcal{A}_{\neg\varphi'}$ are of size $\mathcal{O}(2^{|\varphi|})$. Since universal projection does not alter the state space of an automaton, $\mathcal{A}_{\pi(\neg\varphi')}$ has also $\mathcal{O}(2^{|\varphi|})$ states. Therefore, it follows that $\mathcal{A}^d_\varphi$ has $\mathcal{O}(2^{|\varphi|})$ states as well. Synthesizing remorsefree dominant strategies is, as synthesis of winning strategies, 2EXPTIME-complete:

Theorem 2.4 ([DF14])**.** *Let $\mathscr{A} = (P, V, I, O)$ be a monolithic architecture. Let φ be an LTL formula over atomic propositions V. The question whether a property given as an LTL formula is admissible in a single-process architecture is 2EXPTIME-complete. A remorsefree dominant strategy can be computed in doubly-exponential time.*

Steiger [Ste13] introduced a synthesis approach for remorsefree dominant strategies, which slightly differs from the automaton construction from [DF11] presented above but results in equivalent automata for synthesis. Instead of first constructing two different automata, one for recognizing situations in which the specification is satisfied and one for recognizing situations in which the specification cannot be satisfied by any strategy, his approach immediately constructs a single automaton that accounts for both situations. In particular, he encodes the combination of both automata directly into the specification and then constructs a single automaton out of it. This allows for outsourcing more of the construction work to existing tools for automata generation such as Spot [DLF+16, DRC+22].

More precisely, given an LTL specification φ, Steiger proposes to construct the *modified specification* $\psi = \varphi' \rightarrow \varphi$, where φ' is, similar to the synthesis approach presented above, a copy of φ, where every occurrence of an output variable of the system is replaced with a fresh variable. Then, a universal co-Büchi automaton $\mathcal{A}_\psi$ with $\mathcal{L}(\mathcal{A}_\psi) = \mathcal{L}(\psi)$ is constructed. Intuitively, $\mathcal{A}_\psi$ accepts sequences that either satisfy the specification φ or that define an alternative strategy that violates the specification for the given input sequence. To consider all alternative strategies instead of only a single one, we universally project to the unprimed variables in $\mathcal{A}_\psi$. The resulting automaton is then the desired automaton $\mathcal{A}^d_\varphi$ that recognizes remorsefree dominant strategies. An overview of Steiger's construction is integrated into Figure 2.8, highlighted in violet and with dashed edges.

Part I.

Distributed Systems

Chapter 3

Synthesizing Best-Effort Strategies for Liveness Properties

In this chapter, we study best-effort strategies for the compositional synthesis of distributed systems. The naïve compositional distributed synthesis approach is to synthesize *winning strategies* for the system processes separately. Usually, however, processes need to collaborate in order to achieve the overall system's correctness. For instance, a particular input sequence may prevent the satisfaction of the specification no matter how a single process reacts, yet, the other processes of the system ensure that, in the interplay of their strategies, this input sequence will never be produced. Therefore, separate winning strategies rarely exist.

Remorsefree dominance [DF11], a weaker notion than winning, allows for making *implicit assumptions* on the behavior of the other system processes. A remorsefree dominant strategy, or simply dominant strategy, is allowed to violate the specification as long as no other strategy would have satisfied it in the same situation. Hence, a remorsefree dominant strategy is a best-effort strategy as we do not blame it for violating the specification if the violation is not its fault. Intuitively, a dominant strategy thus implicitly assumes that the other system processes will not violate the overall specification. Searching for dominant rather than winning strategies then allows us to find strategies that do not necessarily satisfy the specification in all situations but in all that are *realistic* in the sense that they actually can occur during the interaction of the system processes if all of them play best-effort strategies. Therefore, remorsefree dominant strategies have been utilized for compositional distributed synthesis algorithms [DF14].

However, the parallel composition of dominant strategies is only guaranteed to be dominant for safety properties [DF14]. For liveness specifications, in contrast, dominance of the parallel composition cannot be ensured. Thus, remorsefree dominance is, in general, not a compositional notion and therefore it is not suitable for compositional synthesis for non-safety specifications. Consider, for example, a system with two processes p_1 and p_2 that send messages to each other, denoted with atomic propositions m_1 and m_2, respectively. Both processes are required to send their message eventually, i.e., the specification is given by $\varphi = \Diamond m_1 \wedge \Diamond m_2$. For p_i, it is dominant to wait for the other process to send the message m_{3-i} before sending its own message m_i: if p_{3-i} sends its message eventually, p_i does so as well, satisfying φ. If p_{3-i} never sends its message, φ is violated, no matter how p_i reacts, and thus the violation of φ is not p_i's fault. If both p_1 and p_2 play this strategy, however, combining them yields a system strategy that never sends any message since both processes wait indefinitely on each other. At the same time, strategies for the entire system that satisfy φ clearly exist, for instance, a strategy that sends both messages in the very first time step.

Bounded dominance [DF14] is a variant of remorsefree dominance that ensures compositionality for general properties. Intuitively, it reduces every specification φ to a safety property by introducing a measure of the strategy's progress with respect to φ and by bounding the number of non-progress steps, i.e., steps in which no progress is made. However, bounded dominance has two major disadvantages: first, it requires an explicit bound on the number of non-progress steps, and it is, in many cases, challenging to determine a suitable bound. Second, not every bounded dominant strategy is remorsefree dominant: if the concrete bound n is chosen too small, every strategy, also one that is not remorsefree-dominant, is trivially n-dominant.

In this chapter, we introduce a new strategy requirement, called *delay-dominance*, that builds upon the ideas of bounded dominance but circumvents the aforementioned weaknesses. Similar to bounded dominance, it introduces a progress measure on strategies with respect to the specification. However, it does not require a concrete bound on the number of non-progress steps, i.e., steps, in which no progress with respect to the specification is made, but relates such steps in the delay-dominant strategy s to non-progress steps in an alternative strategy t: intuitively, a strategy s delay-dominates a strategy t if, whenever s makes a non-progress step, t makes a non-progress step *eventually* as well. A strategy s is delay-dominant if it delay-dominates every other strategy t. In this way, we ensure that a delay-dominant strategy satisfies the specification at least as "fast" as all other strategies in all situations in which the specification can be satisfied. Delay-dominance considers specifications given as alternating co-Büchi automata and the progress measure is defined in terms of visits to rejecting states. We introduce a two-player game, the so-called *delay-dominance game*, which is vaguely leaned on the delayed simulation game for alternating Büchi automata [FW05], to formally define delay-dominance: the winner of the game determines whether or not a strategy s delay-dominates a strategy t on a given input sequence.

We show that every delay-dominant strategy is also remorsefree dominant. Furthermore, we introduce a *bad prefix criterion* for alternating co-Büchi automata such that, if the criterion is satisfied, compositionality of delay-dominance is guaranteed. The criterion is satisfied for many automata, both ones describing safety properties and ones describing liveness properties. Thus, delay-dominance overcomes the weaknesses of both remorsefree and bounded dominance. Note that since delay-dominance relies, as bounded dominance, on the automaton structure, there are realizable specifications for which no delay-dominant strategy exists. However, we experienced that this rarely occurs in practice when constructing the automaton from an LTL formula with standard algorithms. Moreover, if a delay-dominant strategy exists, it is guaranteed to be winning if the specification is realizable. Hence, the parallel composition of delay-dominant strategies for all processes in a distributed system is winning for the whole system as long as the specification is realizable and the compositionality criterion is satisfied. Therefore, delay-dominance is a suitable notion for compositional synthesis.

We thus introduce a synthesis approach for delay-dominant strategies that immediately enables a compositional synthesis algorithm for distributed systems, namely synthesizing delay-dominant strategies for the system processes separately. We present a three-step construction of a universal co-Büchi automaton $\mathcal{A}_{i,\mathcal{A}_\varphi}$ from an LTL formula φ that recognizes delay-dominant strategies for a system process $p_i \in P^-$. The automaton $\mathcal{A}_{i,\mathcal{A}_\varphi}$ can immediately be used for safraless synthesis [KV05] approaches such as bounded synthesis [FS13] to synthesize delay-dominant strategies for single processes. We show that the size of $\mathcal{A}_{i,\mathcal{A}_\varphi}$ is single-exponential in the squared length of φ. Thus, synthesis of delay-dominant strategies is, similar to synthesis of winning or remorsefree dominant strategies, in 2EXPTIME.

Publications and Structure. This chapter is based on work published in the proceedings of the *42nd IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science* [FP22b] and its extended version [FP22c]. The author of this thesis is the lead author of the publications.

This chapter is structured as follows. First, we recap the compositional synthesis approach for distributed systems based on remorsefree dominant strategies [DF14] and discuss its relationship with liveness properties. In particular, we elucidate the shortcomings of both remorsefree dominance and bounded dominance concerning compositional synthesis in Section 3.1.2. Afterward, in Section 3.2, we introduce delay-dominance, our new strategy requirement, and show that every delay-dominant strategy is also remorsefree dominant. Furthermore, we study the compositionality of delay-dominance in Section 3.3. In particular, we consider a bad-prefix criterion for alternating co-Büchi automata and prove that whenever the criterion is satisfied by the automaton representing the specification, then compositionality of delay-dominance is guaranteed. In Section 3.4, we present a three-step construction of a universal co-Büchi automaton that recognizes delay-dominant strategies for single processes and that can immediately be used for synthesizing delay-dominant strategies with safraless synthesis approaches. Lastly, we present the resulting compositional synthesis algorithm for distributed systems utilizing delay-dominant strategies in Section 3.5

3.1. Compositional Synthesis with Dominance

Given an LTL specification φ and a distributed architecture $\mathscr{A}$, the synthesis problem asks whether there exist system strategies $s_1, \dots, s_n$ for the system processes $p_1, \dots, p_n \in P^-$ such that $s_1 \mid\mid \dots \mid\mid s_n \models \varphi$ holds and, if so, derives such strategies. Classical distributed synthesis algorithms directly synthesize the parallel composition of the process strategies, i.e., $s_1 \mid\mid \dots \mid\mid s_n$, together to be able to ensure that $s_1 \mid\mid \dots \mid\mid s_n \models \varphi$ holds. This, however, leads to huge state and search spaces, resulting in poor scalability of the algorithms. *Compositional* distributed synthesis approaches aim at breaking down the synthesis problem for the entire distributed system into synthesis subtasks for the individual processes.

The naïve compositional synthesis approach for distributed systems is given in Algorithm 3.1. For each system process $p_i \in P^-$, it tries to derive a *winning strategy* for the overall system specification φ (line 5), i.e., a system strategy $s_i : (2^{I_i \cup O_i})^\omega \times 2^{I_i} \rightarrow 2^{O_i}$ for process p_i such that $comp(s_i, \gamma) \cup \gamma' \models \varphi$ holds for all input sequences $\gamma \in (2^{I_i})^\omega$ and all sequences $\gamma' \in (2^{V \setminus V_i})^\omega$ of valuations of variables that cannot be observed by p_i. If such a strategy exists, it is stored in the

Algorithm 3.1: Naïve Compositional Distributed Synthesis

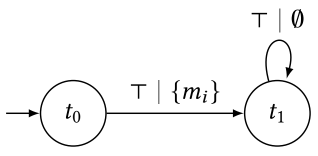
```
   Input: $\varphi$: LTL, A: Architecture
   Output: realizable: Bool, strategies: List Strategy
1  processes ← getSystemProcesses(A)
2  foreach p ∈ processes do
3      pInp ← getProcessInputs(A,p)
4      pOut ← getProcessOutputs(A,p)
5      (pRealizable, pStrategy) ← synthesize($\varphi$, pInp, pOut)
6      if pRealizable then
7          strategies.append(pStrategy)
8      else
9          return (false, [])
10 return (true, strategies)
```

list of process strategies (line 7). Otherwise the algorithm is aborted by returning (false, []) (line 9). If all synthesis tasks succeeded, the list of process strategies is returned (line 10).

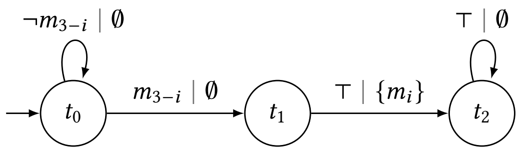
Suppose that the algorithm succeeds and returns a list of strategies $s_1, \dots, s_n$, one for each system process. In that case, it follows immediately from the construction of the strategies, particularly the fact that they are winning, that $s_1 \mid\mid \dots \mid\mid s_n \models \varphi$ holds. Every process strategy ensures that the overall system specification φ is satisfied for *all* input sequences and *all* sequences of variables, which are not observable by the considered process. Thus, in particular, the satisfaction of φ is guaranteed for the sequences that are produced in the interplay of all process strategies. Hence, the synthesis algorithm is sound.

A critical shortcoming of the naïve compositional synthesis approach is, however, that requiring the individual process strategies to be winning for the full system specification is for almost all system architectures and specification too hard of a requirement. A process strategy is required to guarantee the satisfaction of all system requirements – even of those that specify the behavior of other processes – irrespective of whether or not the other processes cooperate in achieving the goal of satisfying the specification. Hence, in most cases, the synthesis task fails for at least one of the processes, resulting in the compositional synthesis algorithm not finding a solution. Identifying the parts of the specification that indeed specify requirements posed on the considered process and only considering them in the process's synthesis task can improve the applicability of the naïve compositional synthesis algorithm. However, identifying such parts is a non-trivial task due to complex interconnections between specification parts, which might result in indirect requirements on a process that are not easy to identify for the developer. Furthermore, often the requirements on the behavior of two processes cannot be decoupled entirely, again resulting in the problem that a process strategy also needs to guarantee that requirements on the behavior of another process are satisfied.

Therefore, we focus on *weakening* the strategy requirement, i.e., not requiring a strategy to satisfy the specification for all input sequences, in this chapter. In this way, we are able to derive strategies for individual system processes in many situations, even if the specification



(a) Strategy transducer $\mathcal{T}_i^s$ for process p_i.



(b) Strategy transducer $\mathcal{T}_i^t$ for process p_i.

Figure 3.1.: Two Moore transducers $\mathcal{T}_i^s$ and $\mathcal{T}_i^t$ representing strategies s_i and t_i for system process $p_i \in P^-$ from the running example. The former sends m_i in the first time step, the latter waits for receiving m_{3-i} before sending m_i.

poses requirements on the behavior of other processes. Consequently, the hard and, currently, manual task of identifying parts of the specification that affect the considered process is often not necessary for successful synthesis. In this chapter, we focus on the strategy requirement *remorsefree dominance* [DF11]. Recall that a dominant strategy is allowed to violate the specification for input sequences for which no other strategy would have satisfied the specification either. It is thus a weaker requirement than winning, and therefore dominant strategies exist in more cases than winning ones.

Example 3.1. Consider the message-sending system with two processes from the introduction, where two processes p_1 and p_2 send messages m_1 and m_2 to each other. Message m_i is sent by process p_i and received by process p_{3-i}. Consequently, the inputs of p_i are given by $I_i = \{m_{3-i}\}$ and the outputs by $O_i = \{m_i\}$. The system specification $\varphi = \diamondsuit m_1 \land \diamondsuit m_2$ formalizes that both processes must send their messages eventually. Throughout this chapter, we will use the message-sending system as a running example.

The satisfaction of the specification $\varphi = \diamondsuit m_1 \land \diamondsuit m_2$ cannot be ensured by any of the processes alone since it poses requirements on output variables of both of them. In order to realize φ, a strategy $s_i : (2^{\{m_1,m_2\}})^* \times 2^{\{m_{3-i}\}} \rightarrow 2^{\{m_i\}}$ for process p_i needs to satisfy φ for every input sequence $\gamma \in (2^{\{m_{3-i}\}})^\omega$. Yet, p_i can only control variable m_i, variable m_{3-i} is uncontrollable and thus its valuation is, for every point in time, defined by s_i's input sequence γ. Therefore, for the sequence $\gamma \in (2^{\{m_{3-i}\}})^\omega$ with $\gamma = \emptyset^\omega$, we have $comp(s_i, \gamma) \cap \{m_{3-i}\} = \emptyset^\omega$ for all strategies s_i for process p_i. Hence, there does not exist a strategy for process p_i that realizes φ and thus there does not exist a winning strategy for p_i and φ.

Let $s_i : (2^{\{m_1,m_2\}})^* \times 2^{\{m_{3-i}\}} \rightarrow 2^{\{m_i\}}$ be a strategy for process p_i that outputs m_i in the very first time step. A finite-state $(2^{\{m_{3-i}\}}, 2^{\{m_i\}})$-transducer $\mathcal{T}_i^s$ with Moore semantics representing s_i is depicted in Figure 3.1a. Clearly, for this strategy s_i, we have $comp(s_i, \gamma) \models \diamondsuit m_i$ for all input sequences $\gamma \in (2^{\{m_{3-i}\}})^\omega$. Furthermore, for all input sequences $\gamma \in (2^{\{m_{3-i}\}})^\omega$ that contain at least one m_{3-i}, i.e., with $\gamma_k \cap \{m_{3-i}\} \neq \emptyset$ for at least one point in time $k \geq 0$, clearly $comp(s_i, \gamma) \models \diamondsuit m_{3-i}$ holds as well and therefore $comp(s_i, \gamma)$ satisfies φ for such input sequences. For $\gamma \in (2^{\{m_{3-i}\}})^\omega$ with $\gamma = \emptyset^\omega$, in contrast, we have $comp(s_i, \gamma) \not\models \varphi$ as shown above. However, *every* strategy $t_i : (2^{\{m_1,m_2\}})^* \times 2^{\{m_{3-i}\}} \rightarrow 2^{\{m_i\}}$ for p_i violates φ for input sequence $\gamma = \emptyset^\omega$ as argued above. Thus, s_i dominates every alternative strategy $t_i : (2^{\{m_1,m_2\}})^* \times 2^{\{m_{3-i}\}} \rightarrow 2^{\{m_i\}}$ for p_i and therefore s_i is dominant for p_i. △

In the following, we recap a compositional distributed synthesis approach for safety specifications [DF14], which utilizes remorsefree dominant strategies rather than winning ones. It thus succeeds in more cases than the naïve compositional distributed synthesis approach. Afterward, we discuss limitations of the approach for liveness properties and, in this way, lay the foundations for the remainder of this chapter.

3.1.1. Compositionality of Dominance for Safety Properties

Since remorsefree dominant strategies exist in more cases than winning ones, it is a straightforward extension of the naïve compositional synthesis approach to try to synthesize individual *remorsefree dominant* strategies for the system processes rather than winning ones [DF14]. The resulting compositional distributed synthesis algorithm is similar to the naïve compositional synthesis algorithm (see Algorithm 3.1), yet, it replaces the synthesis task for winning strategies in line 5 with a synthesis task for dominant strategies. This allows us to synthesize strategies for the processes of a distributed system compositionally although no winning strategies for the individual processes exist and hence this compositional approach succeeds in more cases. Note, however, that although this approach finds solutions in more cases than the naïve approach that tries to synthesize individual winning strategies, it is nevertheless still incomplete since the existence of individual dominant strategies is not guaranteed [DF14].

For compositional distributed synthesis, it is crucial that the strategies for the individual processes can be recomposed to obtain a strategy for the whole system. This property is called *compositionality*. Note here that since we are considering arbitrary system architectures, system processes are allowed to observe and, in particular, react to output variables of other system processes. Therefore, we need to consider process strategies that can be represented with Moore transducers in compositional synthesis as otherwise it is not guaranteed that the parallel composition of the process strategies is complete (see Section 2.6.1).

Given an LTL specification φ, the parallel composition of individual process strategies that are winning for φ is guaranteed to be winning for φ as well as outlined above for the soundness of the naïve compositional synthesis algorithm. For the parallel composition of dominant strategies, in contrast, arguing about – and even achieving – compositionality is much more challenging: realizing the specification and thus satisfying it in all situations has the advantage that nothing better can be achieved; even when considering the whole system and not only individual processes. Dominant strategies, however, are allowed to violate the specification. Although there then is no better strategy on the individual process-level, it is not obvious that nothing better can be achieved when considering the entire system since then not only the behavior of the individual process but that of all other processes can be controlled.

For safety specifications φ, compositionality of remorsefree dominant strategies has been shown [DF14], i.e., the parallel composition of two remorsefree dominant strategies for φ is guaranteed to be dominant for φ as well:

Theorem 3.1 ([DF14]). *Let φ be an LTL formula over atomic propositions V. Let s_1 and s_2 be system strategies for processes $p_1 \in \mathbb{P}$ and $p_2 \in \mathbb{P}$, respectively, and assume that both s_1 and s_2 are dominant for φ. If φ is a safety property, then $s_1 \mid\mid s_2$ is dominant for $p_1 \mid\mid p_2$ and φ.*

Intuitively, dominant strategies are compositional for safety properties since if the parallel composition $s_1 || s_2$ of two dominant strategies s_1 and s_2 violates the specification φ on some input sequence $\gamma \in (2^{(I_1 \cup I_2) \setminus (O_1 \cup O_2)})^\omega$, then there exists a smallest bad prefix η of $comp(s_1 || s_2, \gamma)$, i.e., a smallest prefix such that every extension of it violates φ. Note that the existence of this smallest bad prefix η relies on the fact that the considered specification is a safety property, which, by definition, allows for bad prefixes (see Section 2.3) and consequently also for smallest bad prefixes. The last position of the smallest bad prefix then allows for blaming at least one of the strategies for the violation of φ: the blamable strategy s_i produces an output at this point in time such that φ is violated, irrespective of future inputs as well as future behavior of both process strategies. If there exists an alternative strategy t for $p_1 || p_2$ that does not violate φ, i.e., if the parallel composition $s_1 || s_2$ of s_1 and s_2 is not dominant, then the strategy s_i that is blamable for the violation cannot be remorsefree dominant either: we can then extract a strategy t_i for the corresponding process p_i from the strategy t, which then dominates s_i since t_i does not violate φ on the input sequence resulting from the computation of the full strategy t on γ, while, by the definition of smallest bad prefixes, strategy s_i does. Consequently, since s_1 and s_2 are both dominant for φ by assumption, there cannot exist such an alternative strategy t for $p_1 || p_2$ and therefore $s_1 || s_2$ is dominant.

Since dominant strategies are compositional for safety properties, the parallel composition of separately synthesized process strategies is indeed a useful strategy for the entire system. It is guaranteed to be dominant and, if the specification is realizable, it is even guaranteed to be winning by Proposition 2.5. Hence, Theorem 3.1 enables a compositional synthesis approach for safety properties that synthesizes individual dominant strategies for the system processes (see Section 2.8.2 for an introduction to the synthesis of dominant strategies). In the next section, we study the relationship of dominant strategies and liveness properties and, in particular, the compositionality of dominant strategies for liveness properties.

3.1.2. Dominant Strategies and Liveness Properties

For safety properties, remorsefree dominance is a compositional notion. For liveness properties, however, it is not: compositionality of dominant strategies for safety properties relies heavily on the fact that if the computation of the parallel composition $s_1 || s_2$ of two dominant strategies s_1 and s_2 violates the specification φ on some input sequence γ, then there exists a smallest bad prefix of $comp(s_1 || s_n, \gamma)$. This prefix then allows for blaming at least one of the strategies s_1 and s_2 for the violation of φ and therefore for concluding that $s_1 || s_2$ is dominant.

Liveness properties, however, do not have a bad prefix by definition since every finite prefix can be extended such that the resulting infinite sequence satisfies the liveness property (see Section 2.3). Thus, in particular, there does not exist some point in time at which a strategy needs to show a specific behavior to be dominant. Hence, it might be the case that both p_1 and p_2 have to show a specific behavior *eventually* to satisfy the specification φ, but both can postpone it indefinitely by waiting for the other process to show the behavior first. We cannot blame any process for postponing this behavior since waiting for the other process is dominant. If both processes do so, however, φ is violated while there might exist alternative strategies for the full system $p_1 || p_2$ that satisfy φ.

Example 3.2. Consider the message-sending system from the running example. For system process $p_i \in P^-$, let t_i be a strategy that waits for receiving m_{3-i} before sending its own message m_i. A finite-state Moore transducer representing such a strategy is depicted in Figure 3.1b. This strategy is dominant for p_i and φ: for input sequences $\gamma \in (2^{\{m_{3-i}\}})^\omega$ in which m_{3-i} occurs at some point in time, the computation $comp(t_i, \gamma)$ contains m_{3-i} as well. Therefore, it satisfies $\diamondsuit m_{3-i}$. Furthermore, t_i sets m_i to *true* one step afterward. Therefore, $comp(t_i, \gamma)$ satisfies $\diamondsuit m_i$ as well and hence $comp(t_i, \gamma) \models \varphi$ follows. For input sequences $\gamma' \in (2^{\{m_{3-i}\}})^\omega$ in which m_{3-i} never occurs, the computation of ever strategy for p_i on γ' does not contain any m_{3-i}, thus violating $\diamondsuit m_{3-i}$ and hence also violating φ. Therefore, t_i is allowed to violated φ on γ'.

The parallel composition of such strategies t_1 and t_2 for both processes, however, does not send any message and thus violates φ. None of the strategies can be blamed for not sending the corresponding message since there is no concrete time step at which φ enforces that one of the processes needs to send its message in order to be dominant. Yet, there clearly exist strategies for the entire system $p_1 \mid\mid p_2$ that satisfy φ, for instance a strategy that sends both m_1 and m_2 in the very first time step. Hence, $t_1 \mid\mid t_2$ is not dominant. △

The parallel composition of dominant strategies is thus not guaranteed to be dominant for liveness properties. Therefore, composing separately synthesized dominant strategies does not necessarily yield a useful strategy for the overall system. Hence, synthesizing individual dominant strategies is not sound for liveness properties. Therefore, the extension of the naïve compositional synthesis algorithm with dominant strategies presented in the previous section is not a suitable synthesis approach for general specifications.

Bounded dominance [DF14] is a variant of dominance that is compositional for both safety and liveness properties. Intuitively, it reduces every specification φ to a safety property by introducing a bound on the number of steps in which a strategy *does not make progress* with respect to φ. The progress measure is not defined on the LTL formula φ itself but on a universal co-Büchi automaton $\mathcal{A}_\varphi$ that accepts the same language as φ. The *measure* $m_{\mathcal{A}_\varphi}(comp(s_i, \gamma,))$ of a strategy s_i for system process $p_i \in P^-$ on an input sequence $\gamma \in (2^{I_i})^\omega$ is then the supremum of the number of rejecting states of the runs of $\mathcal{A}_\varphi$ induced by $comp(s_i, \gamma)$. Slightly overloading notation, we call the set of runs induced by all sequences $comp(s_i, \gamma) \cup \gamma'$ for $\gamma' \in (2^{V \setminus V_i})^\omega$, i.e., the set $\{r \mid r \in Runs(\mathcal{A}_\varphi, comp(s_i, \gamma) \cup \gamma') \land \gamma' \in (2^{V \setminus V_i})^\omega\}$, also the set of runs induced by $comp(s_i, \gamma)$. Given a bound $n \in \mathbb{N}_0$, bounded dominance for n, which is also called n-dominance, is then defined utilizing the measure $m_{\mathcal{A}_\varphi}$:

Definition 3.1 (n-Dominant Strategy [DF14]).
Let V be a finite set of variables. Let $I \subseteq V$ and $O \subseteq V$ be finite sets of input and output variables with $I \cap O = \emptyset$. Let $n \in \mathbb{N}_0$. Let φ be an LTL formula over atomic propositions V and let $\mathcal{A}_\varphi$ be a universal co-Büchi automaton with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$. Let $s : (2^{I \cup O})^* \times 2^I \rightarrow 2^O$ be a strategy. A strategy $t : (2^{I \cup O})^* \times 2^I \rightarrow 2^O$ is *n-dominated by s* if, and only if, for all $\gamma \in (2^I)^\omega$ either $m_{\mathcal{A}_\varphi}(comp(s, \gamma)) \leq n$ or $m_{\mathcal{A}_\varphi}(comp(t, \gamma)) > n$ holds. Strategy s is *n-dominant* for $\mathcal{A}_\varphi$ if, and only if, it n-dominates all alternative strategies $t : (2^{I \cup O})^* \times 2^I \rightarrow 2^O$.

Similar to remorsefree dominance, a bounded dominant strategy performs at least as good as every other strategy. While, in remorsefree dominance, "good" refers to satisfying the specifica-

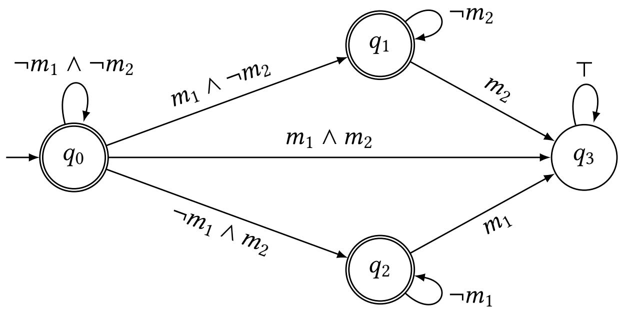


Figure 3.2.: Dominant strategy t_i for system process $p_i \in P^-$ from the running example that waits for receiving m_{3-i} before sending m_i and a co-Büchi automaton $\mathcal{A}_\varphi$ for $\varphi = \diamondsuit m_1 \land \diamondsuit m_2$.

tion, it is defined in terms of satisfying the specification with a small number of visits to rejecting states for bounded dominance. Intuitively, visiting only few rejecting states corresponds to satisfying the specification *fast*. For safety specifications, the notions of remorsefree dominance and bounded dominance coincide. For liveness specifications, however, they differ.

Example 3.3. Consider the message-sending system from the running example. A universal co-Büchi automaton $\mathcal{A}_\varphi$ with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$ is depicted in Figure 3.2. Furthermore, reconsider the strategy t_i for system process $p_i \in P^-$ that waits for receiving m_{3-i} before sending its own message m_i and the finite-state Moore transducer representing t_i depicted in Figure 3.1b. Consider system process p_1 and the corresponding version t_1 of strategy t_i. Let $\gamma \in (2^{\{m_2\}})^\omega$ be an input sequence for process p_1 that models that p_2 sends its message for the first time at point in time $\ell \geq 0$, i.e., we have $\gamma_\ell = \{m_2\}$ and $\gamma_j = \emptyset$ for all $j \geq 0$ with $j < \ell$. Then, $comp(t_1, \gamma)$ neither contains m_1 nor m_2 up to point in time ℓ, contains only m_2 at point in time ℓ, and contains m_1 at point in time $\ell + 1$. Hence, $comp(t_1, \gamma)$ induces a single run r of $\mathcal{A}_\varphi$ that, starting in q_0, stays in q_0 up to point in time $\ell - 1$, then, reading m_2 but not m_1, moves to q_2, and then, reading m_1, moves to q_3 immediately afterward, where it stays forever. Since q_0 and q_2 are rejecting states while q_3 is not, r thus contains $\ell + 2$ visits to rejecting states and thus $m_{\mathcal{A}_\varphi}(comp(t_1, \gamma)) = \ell + 2$ holds.

Consider an alternative strategy s_1 for p_1 that sends m_1 in the first time step, irrespective of whether or not it receives m_2. For input sequence γ described above, $comp(s_1, \gamma)$ then contains m_1 at point in time 0 and m_2 at point in time ℓ. If $\ell = 0$ holds, then $comp(s_1, \gamma)$ induces the single run r' that, starting in q_0, immediately moves to q_3 as it reads both m_1 and m_2 in the very first time step and stays there forever. If $\ell > 0$ holds, then $comp(s_1, \gamma)$ induces the single run r'' that, starting in q_0, moves to q_1 as it reads m_1 but not m_2, then stays in q_1 up to point in time ℓ_1, and then, reading m_2, moves to q_3, where it stays forever. Since q_0 and q_1 are rejecting states while q_3 is not, r' contains a single visit to a rejecting state, while r'' contains $\ell + 1$ visits to rejecting states and thus $m_{\mathcal{A}_\varphi}(comp(s_1, \gamma)) = \ell + 1$ holds. Hence, for bound $n = \ell + 1$, we have $m_{\mathcal{A}_\varphi}(comp(s_1, \gamma)) = n$, while $m_{\mathcal{A}_\varphi}(comp(t_1, \gamma)) > n$. Therefore, t_1 does not n-dominate s_1 on input γ for bound $n = \ell + 1$ and consequently t_1 is not n-dominant, while it is remorsefree dominant as outlined in Example 3.2. △

A crucial disadvantage of bounded dominance is that it does not imply remorsefree dominance [DF14]. There are specifications φ or, more precisely, automata $\mathcal{A}_\varphi$ for φ, with a minimal measure $m \in \mathbb{N}$, i.e., for all strategies, there exists some input sequence such that the resulting computation has a measure of at least m. When choosing a bound $n < m$, every strategy is trivially n-dominant for the automaton $\mathcal{A}_\varphi$, even non-dominant ones. For the message-sending system from the running example and the universal co-Büchi automaton depicted in Figure 3.2, for instance, the minimal measure is $m = 1$. A strategy for process p_i that never sends its message m_i is clearly not dominant; yet, it is trivially 0-dominant.

Therefore, the choice of the bound $n \in \mathbb{N}_0$ is crucial for bounded dominance. However, it is not obvious how to determine a *good* bound. On the one hand, it must be large enough to avoid non-dominant strategies. On the other hand, as the bound has a huge impact on the synthesis time, it cannot be chosen too large as otherwise synthesis becomes infeasible: in order to synthesize bounded dominant strategies, we need to count the number of rejecting states that already occurred during a run up to the bound n. Hence, the larger we choose n, the higher we need to count and therefore the size of an automaton recognizing a bounded dominant strategy grows tremendously for larger bounds. Especially for specifications with several complex dependencies between processes, it is hard to determine a proper bound. Hence, bounded dominance is not practically applicable to compositional synthesis for liveness properties. In the remainder of this chapter, we introduce a different variant of dominance that implies remorsefree dominance and ensures compositionality also for liveness properties.

3.2. Delay-Dominance

In this section, we introduce a new requirement for strategies, *delay-dominance*, which resembles remorsefree dominance but ensures compositionality also for liveness properties. It builds on the idea of bounded dominance to not only consider the satisfaction of the LTL specification φ but to measure progress based on an automaton representing φ. Similar to bounded dominance, we utilize visits of rejecting states in a co-Büchi automaton to measure progress. Yet, we use an *alternating* automaton instead of a universal one. Note that delay-dominance can be equivalently formulated on universal co-Büchi automata, yet, using alternating automata allows for more efficient synthesis algorithms for delay-dominant strategies as we will discuss later in this chapter (see Section 3.4). Moreover, we do not require a fixed bound on the number of visits to rejecting states; rather, we relate visits to rejecting states induced by the possibly delay-dominant strategy to visits to rejecting states induced by the alternative strategy.

Intuitively, delay-dominance requires that, for every input sequence, every visit to a rejecting state in the alternating co-Büchi automaton $\mathcal{A}_\varphi$ caused by the computation of the delay-dominant strategy on this input sequence *is matched* with a visit to a rejecting state in $\mathcal{A}_\varphi$ caused by the computation of the alternative strategy on the same input sequence *eventually*. The visits to rejecting states of $\mathcal{A}_\varphi$ are closely related to the satisfaction of the LTL specification φ: if infinitely many rejecting states are visited, then φ is not satisfied. Thus, delay-dominance allows a strategy to violate the specification if all alternative strategies violate it as well in the same situation. Defining delay-dominance on the rejecting states of $\mathcal{A}_\varphi$ instead of the satisfaction of φ allows for measuring the progress on satisfying the specification. Thus, we can distinguish strategies that wait indefinitely for another process to act – and hence strategies that are critical for compositionality – from those that do not. Intuitively, a strategy s that waits for another process to act first will visit a rejecting state later than a strategy t that does not wait but tries to meet its obligations as soon as possible. This visit to a rejecting state is then not matched eventually with a visit to a rejecting state induced by t, preventing delay-dominance of s.

In the following, we formally define the strategy requirement of delay-dominance. Afterward, we prove that every delay-dominant strategy is also remorsefree dominant.

3.2.1. The Delay-Dominance Game

Delay-dominance is a game-based notion. We thus introduce an infinite two-player game, the so-called *delay-dominance game*, which is loosely inspired by the delayed simulation game for alternating Büchi automata [FW05], to define delay-dominance.

Given an LTL specification φ, an equivalent alternating co-Büchi automaton $\mathcal{A}_\varphi = (Q, q_0, \delta, F)$ with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$, two strategies s_i and t_i for a process $p_i \in \mathbb{P}$, an input sequence $\gamma \in (2^{I_i})^\omega$, and an infinite sequence $\gamma' \in (2^{V \setminus V_i})^\omega$ fixing the valuations of the variables that p_i cannot observe, the delay-dominance game determines whether or not s_i delay-dominates t_i for $\mathcal{A}_\varphi$ on input γ when additionally considering the sequence γ'. Intuitively, the delay-dominance game proceeds in rounds. At the beginning of each round, a pair (p, q) of states $p, q \in Q$ of $\mathcal{A}_\varphi$ and the number of the iteration $j \in \mathbb{N}_0$ is given. Here, p represents a state of $\mathcal{A}_\varphi$ that is visited by a branch of a run tree of $\mathcal{A}_\varphi$ induced by $comp(t_i, \gamma) \cup \gamma'$, while q represents a state of $\mathcal{A}_\varphi$ that is visited by a branch of a run tree of $\mathcal{A}_\varphi$ induced by $comp(s_i, \gamma) \cup \gamma'$. We thus also call p the *alternative state* and q the *dominant state*. For the sake of readability, let $\sigma^{s_i} = comp(s_i, \gamma) \cup \gamma'$ and $\sigma^{t_i} = comp(t_i, \gamma) \cup \gamma'$. The two players Duplicator and Spoiler, where Duplicator takes on the role of Player 0, play as follows:

1. Spoiler chooses a set $c \in \delta(p, \sigma^{t_i}_j)$.
2. Duplicator chooses a set $c' \in \delta(q, \sigma^{s_i}_j)$.
3. Spoiler chooses a state $q' \in c'$.
4. Duplicator chooses a state $p' \in c$.

The starting pair of the next round is then $((p', q'), j+1)$. Beginning with the pair $((q_0, q_0), 0)$, the players construct an infinite play that determines the winner. Duplicator wins for a play if every rejecting dominant state is eventually matched with a rejecting alternative state.

Both the possible delay-dominant strategy s_i and the alternative strategy t_i may control the nondeterministic transitions of $\mathcal{A}_\varphi$, while the universal ones are uncontrollable. Since, intuitively, strategy t_i is controlled by an opponent when proving that s_i delay-dominates t_i, we thus have a change in control for t_i: for process strategy s_i, Duplicator controls the existential transitions of $\mathcal{A}_\varphi$ and Spoiler controls the universal ones. For process strategy t_i, in contrast, Duplicator controls the universal transitions of $\mathcal{A}_\varphi$ and Spoiler controls the existential ones. Note that the order in which the players Spoiler and Duplicator make their moves is crucial to

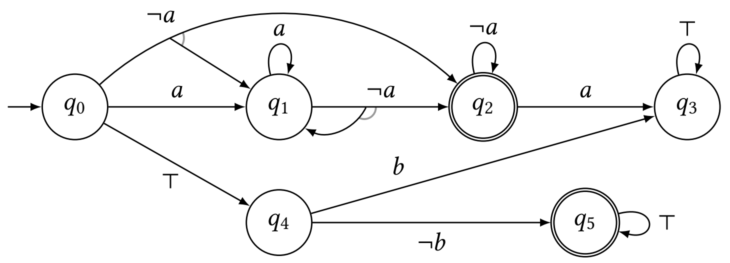


Figure 3.3.: Alternating co-Büchi automaton $\mathcal{A}_\psi$ for $\psi = \square\Diamond a \vee \bigcirc b$.

ensure that Duplicator wins the game when considering the very same process strategies. By letting Spoiler move first, Duplicator is able to mimic – or *duplicate* – Spoiler's moves. Formally, the delay-dominance game is defined as follows:

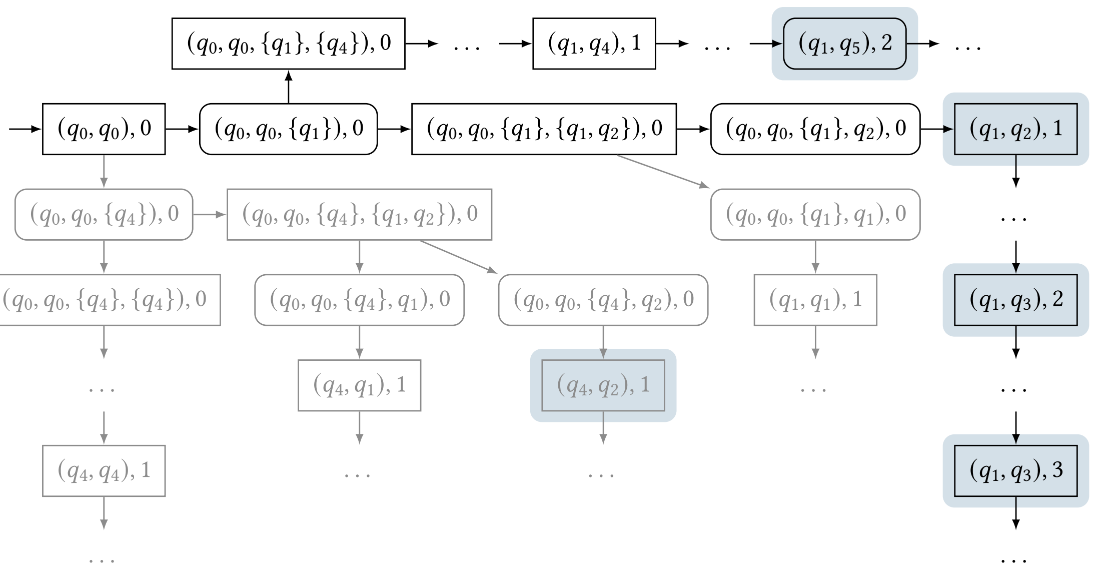
Definition 3.2 (Delay-Dominance Game).
Let φ be an LTL formula over atomic propositions V. Let $\mathcal{A}_\varphi = (Q, q_0, \delta, F)$ be an alternating co-Büchi automaton with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$. Based on $\mathcal{A}_\varphi$, we define the sets $S_\exists = (Q\times Q)\times\mathbb{N}_0$, $D_\exists = (Q \times Q \times 2^Q) \times \mathbb{N}_0$, $S_\forall = (Q \times Q \times 2^Q \times 2^Q) \times \mathbb{N}_0$, and $D_\forall = (Q \times Q \times Q \times 2^Q) \times \mathbb{N}_0$. Let $\sigma, \sigma' \in (2^V)^\omega$ be infinite sequences. The *delay-dominance game* $(\mathcal{A}_\varphi, \sigma, \sigma')$ is the game $\mathbb{G} = (\mathbb{A}, \mathbb{W})$ defined by $\mathbb{A} = (P, P_0, P_1, v_0, E)$ with $P = S_\exists \cup D_\exists \cup S_\forall \cup D_\forall$, $P_0 = D_\exists \cup D_\forall$, and $P_1 = S_\exists \cup S_\forall$ as well as $v_0 = ((q_0, q_0), 0)$ and

$$\begin{aligned} E = &\left\{(((p,q),j),((p,q,c),j)) \mid c \in \delta(p,\sigma_j)\right\} \\ &\cup \left\{(((p,q,c),j),((p,q,c,c'),j)) \mid c' \in \delta(q,\sigma'_j)\right\} \\ &\cup \{(((p,q,c,c'),j),((p,q,c,q'),j)) \mid q' \in c'\} \\ &\cup \{(((p,q,c,q'),j),((p',q'),j+1)) \mid p' \in c\}, \end{aligned}$$

and the winning condition $\mathbb{W} = \{\rho \in P^\omega \mid \forall k \in \mathbb{N}_0.\ f_{dom}(\rho_k) \in F \rightarrow \exists k' \geq k.\ f_{alt}(\rho_{k'}) \in F\}$, where $f_{alt}(v) = \#_1(\#_1(v))$ and $f_{dom}(v) = \#_2(\#_1(v))$, i.e., $f_{alt}(v)$ and $f_{dom}(v)$ map a position $v \in P$ to the alternative state and the dominant state of v, respectively.

The formal definition of a delay-dominance game thus follows thoroughly the intuitive description of the game. The states of the game and in particular their assignment to the players Spoiler and Duplicator match the choices occurring for Spoiler and Duplicator. Furthermore, the set of edges is carefully designed to ensure the desired structure of the game, i.e., the order in which the players make their moves.

Example 3.4. Let $V = \{a, b\}$ and consider the LTL formula $\psi = \square\Diamond a \vee \bigcirc b$ over V. An alternating co-Büchi automaton $\mathcal{A}_\psi$ with $\mathcal{L}(\mathcal{A}_\psi) = \mathcal{L}(\psi)$ is depicted in Figure 3.3. Let p_i be a process with inputs $I_i = \{b\}$ and outputs $O_i = \{a\}$. Let s_i be a strategy for p_i that does not output a in the very first time step but in all time steps afterward, i.e., we have $comp(s_i, \gamma) \cap O_i = \emptyset\{a\}^\omega$ for all input sequences $\gamma \in (2^{\{b\}})^\omega$. Let t_i be a strategy for p_i that outputs a in every step, i.e., we



$(q_0, q_0, \{q_1\}, \{q_4\}), 0$
$(q_1, q_4), 1$
$(q_1, q_5), 2$
$(q_0, q_0), 0$
$(q_0, q_0, \{q_1\}), 0$
$(q_0, q_0, \{q_1\}, \{q_1, q_2\}), 0$
$(q_0, q_0, \{q_1\}, q_2), 0$
$(q_1, q_2), 1$
$(q_0, q_0, \{q_4\}), 0$
$(q_0, q_0, \{q_4\}, \{q_1, q_2\}), 0$
$(q_0, q_0, \{q_1\}, q_1), 0$
$(q_0, q_0, \{q_4\}, \{q_4\}), 0$
$(q_0, q_0, \{q_4\}, q_1), 0$
$(q_0, q_0, \{q_4\}, q_2), 0$
$(q_1, q_1), 1$
$(q_1, q_3), 2$
$(q_4, q_1), 1$
$(q_4, q_2), 1$
$(q_4, q_4), 1$
$(q_1, q_3), 3$

Figure 3.4.: Partial game arena of the delay-dominance game $\mathbb{G} = (\mathcal{A}_\psi, comp(t_i, \gamma), comp(s_i, \gamma))$ from Example 3.4. Positions controlled by Spoiler are depicted as rectangles, positions with rounded edges are controlled by Duplicator. Parts of the game arena that are not consistent with the winning moves of Spoiler are grayed out. Positions of the form $((p, q), j)$ that are critical for Duplicator are highlighted in blue.

have $comp(s_i, \gamma) \cap O_i = \{a\}^\omega$ for all input sequences $\gamma \in (2^{\{b\}})^\omega$. Let $\gamma \in (2^{\{b\}})^\omega$ be some input sequence for p_i that does not contain any b, i.e., we have $\gamma = \emptyset^\omega$. Consider the delay-dominance game $\mathbb{G} = (\mathcal{A}_\varphi, comp(t_i, \gamma), comp(s_i, \gamma))$. Note here that since $I_i \cup O_i = V$ holds, $comp(s_i, \gamma)$ and $comp(t_i, \gamma)$ are indeed infinite sequences over V. The relevant part of the game arena of $\mathbb{G}$ is depicted in Figure 3.4. Positions controlled by Spoiler are depicted as rectangles. Positions with rounded edges, in contrast, are controlled by Duplicator. Positions of the form $((p, q), j)$ that are *critical* for Duplicator are highlighted in blue. A position is critical if it is reachable in a play with a rejecting dominant state in a round $j' \in \mathbb{N}_0$ with $j' < j$ that is not yet matched with a rejecting alternative state up to round j. A strategy μ for Spoiler in $\mathbb{G}$ or, more precisely, all initial plays that are consistent with μ, are depicted in black. All other parts of the game arena are grayed out. In the first step of the game, Spoiler chooses the transition from q_0 to q_1 in the alternative states; resulting in the successor set $\{q_1\}$.

First, suppose that Duplicator chooses the successor set $\{q_4\}$ and thus the transition from q_0 to q_4 in the dominant states. Then, since both successor sets are singletons, both players do not have further choices in the first round of the game and thus we obtain the node $((q_1, q_4), 1)$. Since, in the second time step, t_i outputs a while γ does not contain a b, Spoiler and Duplicator do not have any choices: the alternative states stay in q_1 while the dominant states move from q_4 to q_5, resulting in the node $((q_1, q_5), 2)$. Since t_1 further outputs a in every following time step and since state q_5 only contains a self-loop, Spoiler and Duplicator do not have any choices in the remainder of the game: the alternative states always stay in q_1 and the dominant states always stay in q_5. Thus, all following positions in a play in $\mathbb{G}$ are of the form $((q_1, q_5), j)$. Note here that q_1 is non-rejecting while q_5 is rejecting. Therefore, no play ρ that is consistent with Spoiler's choice in the very first round of the game defined by μ as well as Duplicator's choice afterward considered in this case contains any rejecting alternative state, while it contains infinitely many rejecting dominant state. Hence, we have $\rho \notin \mathbb{W}$ for all such plays ρ.

Second, suppose that Duplicator chooses the successor set $\{q_1, q_2\}$ in the first round of the game. Then, μ defines that Spoiler chooses q_2 as the successor of q_0 in the dominant states. Since the successor set $\{q_1\}$ in the alternative states is a singleton, Duplicator does not have any further choice; resulting in the node $((q_1, q_2), 1)$. As in the case above, t_i ensures that the alternative states always stay in q_1. Since s_i outputs a in the second time step, the only successor set for the dominant states is $\{q_3\}$, resulting in the node $((q_1, q_3), 2)$ since, due to the fact that $\{q_3\}$ is a singleton, Spoiler does not have any choice other than choosing q_3. Due to the structure of $\mathcal{A}_\psi$ and, in particular, the fact that q_3 is a sink state, i.e., it only has a self-loop, the dominant states always stay in q_3. Therefore, since the alternative states always stay in q_1 as outlined above, all subsequent positions in a play in $\mathbb{G}$ are of the form $((q_1, q_3), j)$. Therefore, every play ρ that is consistent with Spoiler's choice in the very first round of the game defined by μ as well as Duplicator's choice afterward considered in this case contains no rejecting alternative state at all since both q_0 and q_1 are non-rejecting, while it contains a rejecting dominant state at the second time step, namely q_2. Hence, we have $\rho \notin \mathbb{W}$ for all such plays ρ.

Thus, there exists a strategy μ for Spoiler in $\mathbb{G}$ such that we have $\rho \notin \mathbb{W}$ for all initial consistent plays $\rho \in \mathit{Plays}(\mathbb{G}, \mu)$. Hence, Duplicator loses the game $\mathbb{G}$, while Spoiler wins it. Moreover, for an input sequence $\gamma \in (2^{\{b\}})^\omega$, Duplicator has a winning strategy in the delay-dominance game $\mathbb{G}' = (\mathcal{A}_\psi, \mathit{comp}(s_i, \gamma), \mathit{comp}(t_i, \gamma))$ due to a similar choice as the one of Spoiler defined by μ in $\mathbb{G}$ in the very first round of the game: in $\mathbb{G}'$, Duplicator controls the existential transitions in $\mathcal{A}_\psi$ when reading $\mathit{comp}(t_i, \gamma)$. Hence, in particular, it is Duplicator's choice to either let the dominant states move from q_0 to q_1 or to q_4. If it chooses q_4, then it follows analogously to the argument for game $\mathbb{G}$ that no initial consistent play will ever encounter any rejecting dominant; resulting in the fact hat every initial consistent play satisfies the winning condition. △

We now define the notion of delay-dominance based on the delay-dominance game. Intuitively, the winner of the game for the computations of two strategies s_i and t_i determines whether or not s_i delay-dominates t_i on a given input sequence. Similar to remorsefree dominance, we then lift this definition to delay-dominant strategies. Formally:

Definition 3.3 (Delay-Dominant Strategy).
Let φ be an LTL formula over atomic propositions V. Let $\mathcal{A}_\varphi$ be an alternating co-Büchi automaton with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$. Let s_i and t_i be strategies for process $p_i \in \mathbb{P}$. Then, s_i *delay-dominates t_i on input sequence $\gamma \in (2^{I_i})^\omega$* for $\mathcal{A}_\varphi$, denoted $t_i \trianglelefteq_{\mathcal{A}_\varphi, \gamma} s_i$, if, and only if, Duplicator wins the delay-dominance game $(\mathcal{A}_\varphi, \mathit{comp}(t_i, \gamma) \cup \gamma', \mathit{comp}(s_i, \gamma) \cup \gamma')$ for all $\gamma' \in (2^{V \setminus V_i})^\omega$. Strategy s_i *delay-dominates* t_i for $\mathcal{A}_\varphi$, denoted $t_i \trianglelefteq_{\mathcal{A}_\varphi} s_i$, if, and only if, $t_i \trianglelefteq_{\mathcal{A}_\varphi, \gamma} s_i$ holds for all $\gamma \in (2^{I_i})^\omega$. Strategy s_i *is delay-dominant* for $\mathcal{A}_\varphi$ if, and only if, $t_i \trianglelefteq_{\mathcal{A}_\varphi} s_i$ holds for every strategy t_i for process p_i.

Similar to the definition of the satisfaction of specifications by computations of strategies, we require a strategy to satisfy the delay-dominance condition, i.e., that Duplicator wins the delay-dominance game, for *all* valuations of variables that the considered process cannot observe. Intuitively, those variables are thus treated similarly to inputs, namely that an delay-dominant strategy needs to delay-dominate every other strategy for all valuations of these variables. However, they cannot be observed by the considered process, and thus, in particular, the strategy cannot react to them. This matches the definition of architectures and, in general, the intuition of unobservable variables. In the following, we illustrate delay-dominance and, in particular, the concept of delay-dominant strategies with the running example.

Example 3.5. Consider the message-sending system from the running example and the co-Büchi automaton $\mathcal{A}_\varphi$ from Figure 3.2, which describes the specification $\varphi = \Diamond m_1 \land \Diamond m_2$. Although $\mathcal{A}_\varphi$ was constructed as a *universal* co-Büchi automaton, every universal automaton can be seen as an alternating automaton without any nondeterministic choices. Furthermore, $\mathcal{A}_\varphi$ is, in fact, the alternating co-Büchi automaton that we obtain from the LTL formula φ when utilizing standard algorithms for automaton construction. Therefore, we consider it to be an alternating co-Büchi automaton in the remainder of this chapter.

Let s_1 be a strategy for system process p_1 that sends message m_1 in the first time step and let t_1 be a strategy that waits for receiving message m_2 before sending m_1. To determine whether s_1 delay-dominates t_1 for $\mathcal{A}_\varphi$ on an input sequence $(2^{\{m_2\}})^\omega$, we consider the delay-dominance game $\mathbb{G} = (\mathcal{A}_\varphi, comp(t_1, \gamma), comp(s_1, \gamma))$. Since $\mathcal{A}_\varphi$ is, in fact, deterministic, the computations $comp(s_1, \gamma)$ and $comp(t_1, \gamma)$ both induce a single run tree with a single branch in $\mathcal{A}_\varphi$ for every input sequence $\gamma \in (2^{\{m_2\}})^\omega)$. Hence, the players Spoiler and Duplicator do not have any choices in any delay-dominance game for $\mathcal{A}_\varphi$. Therefore, we do not provide the (partial) game arena here, but only the unique sequence of state pairs (p, q) of the delay-dominance game, abstracting from all intermediate tuples.

First, consider an input sequence $\gamma \in (2^{\{m_2\}})^\omega$ that contains message m_2 for the first time at point in time $\ell \geq 0$. Then, the single branch of the single run tree of $\mathcal{A}_\varphi$ induced by $comp(s_1, \gamma)$ starts in q_0, moves to q_1 immediately if $\ell > 0$, stays there up to the occurrence of m_2 and then moves to q_3, where it stays forever. If $\ell = 0$, then the branch moves immediately from q_0 to q_3. The single branch of the single run tree of $\mathcal{A}_\varphi$ induced by $comp(t_1, \gamma)$, in contrast, stays in q_0 until m_2 occurs, then moves to q_2 and then immediately to q_3, where it stays forever. Thus, we obtain the unique sequence

$$(q_0, q_0)(q_0, q_1)^{\ell-1}(q_2, q_3)(q_3, q_3)^\omega$$

of state pairs in the delay-dominance game $\mathbb{G} = (\mathcal{A}_\varphi, comp(t_1, \gamma), comp(s_1, \gamma))$. The last rejecting alternative state, i.e., a rejecting state induced by $comp(t_1, \gamma)$ occurs at point in time $\ell + 1$, namely state q_2. In contrast, the last rejecting dominant state i.e., a rejecting state induced by $comp(s_1, \gamma)$, occurs at point in time ℓ, namely state q_1. Thus, $t_1 \trianglelefteq_{\mathcal{A}_\varphi, \gamma} s_1$ holds. In fact, $t_1' \trianglelefteq_{\mathcal{A}_\varphi, \gamma} s_1$ holds for *all* alternative strategies t_1' for p_1 for such an input sequence γ since every strategy t_1' for p_1 induces at least ℓ visits to rejecting states due to the structure of γ.

Second, consider an input sequence $\gamma' \in (2^{\{m_2\}})^\omega$ that does not contain any message m_2, i.e., we have $\gamma' = \emptyset^\omega$. Then, the single branch of the single run tree of $\mathcal{A}_\varphi$ induced a computation of any strategy t_1' for p_1 on γ' never reaches q_3 and thus only visits rejecting states. Hence, every visit to a rejecting state induced by $comp(s_1, \gamma')$ is matched with a visit to a rejecting state induced by $comp(t_1', \gamma')$ for all strategies t_1' for p_1. Thus, $t_1' \trianglelefteq_{\mathcal{A}_\varphi, \gamma'} s_1$ holds for all alternative strategies t_1' as well. We can thus conclude that s_1 is delay-dominant for $\mathcal{A}_\varphi$, meeting our intuition that s_1 should be allowed to violate φ in situations in which it never receives m_2.

Strategy t_1, in contrast, is remorsefree dominant for p_1 and φ but not delay-dominant for $\mathcal{A}_\varphi$: consider again an input sequence $\gamma \in (2^{\{m_2\}})^\omega$ that contains the very first m_2 at point in time ℓ. For the delay-dominance game $\mathbb{G}' = (\mathcal{A}_\varphi, comp(s_1, \gamma), comp(t_1, \gamma))$, we obtain the following sequence of state pairs:

$$(q_0, q_0)(q_1, q_0)^{\ell-1}(q_3, q_2)(q_3, q_3)^\omega.$$

It contains a rejecting dominant state, i.e., a visit to a rejecting state of $\mathcal{A}_\varphi$ induced by $comp(t_1, \gamma)$, at point in time $\ell + 1$, while the last rejecting alternative state, i.e., a visit to a rejecting state of $\mathcal{A}_\varphi$ induced by $comp(s_1, \gamma)$, occurs at point in time ℓ. Hence, t_1 does not delay-dominate s_1 in input sequence γ, preventing that it is delay-dominant to wait indefinitely for the other process to send its message first. △

For the usefulness of the notion of delay-dominance and its applicability in compositional synthesis, it is crucial that every strategy delay-dominates itself. Otherwise, no strategy at all would be delay-dominant. This property is ensured by the design of the delay-dominance game and, in particular, by the order in which the players make their moves:

Lemma 3.1. *Let φ be an LTL formula over atomic propositions V. Let $\mathcal{A}_\varphi$ be an alternating co-Büchi automaton with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$. Let s_i be a strategy for process $p_i \in \mathbb{P}$ and let $\gamma \in (2^{I_i})^\omega$ be some input sequence. Then, $s_i \trianglelefteq_{\mathcal{A}_\varphi, \gamma} s_i$ holds.*

Proof. Let $\gamma' \in (2^{V \setminus V_i})^\omega$ be s sequence of valuations of variables that cannot be observed by p_i. Consider the delay-dominance game $\mathbb{G} = (\mathcal{A}_\varphi, comp(s_i, \gamma) \cup \gamma', comp(s_i, \gamma) \cup \gamma')$. For the sake of readability, let $\sigma = comp(s_i, \gamma) \cup \gamma'$. We construct a winning strategy μ for Duplicator in the game $\mathbb{G}$ by mimicking the respective moves of Spoiler. Since Spoiler moves first by construction of the game, this is always possible. Formally, we construct the strategy μ as follows: let $v \in P^*$ be a finite sequence of positions of $\mathbb{G}$ and let $v \in P_0$ be a position that is assigned to Duplicator, i.e., we have $v \in D_\exists \cup D_\forall$. First, suppose that $v \in D_\exists$ holds, i.e., position v is of the form $v = ((p, q, c), j)$. By construction of $\mathbb{G}$, we have $c \in \delta(p, \sigma_j)$. If $p = q$ holds, we therefore define $\mu(v, v) = ((p, q, c, c), j)$. Otherwise, we define $\mu(v, v) = ((p, q, c, c'), j)$ for some $c' \in \delta(q, \sigma_j)$. Second, suppose that $v \in D_\forall$ holds, i.e., position v is of the form $v = ((p, q, c, q'), j)$. Moreover, by construction of $\mathbb{G}$, the last position of v is of the form $v_{|v|-1} = ((p, q, c, c'), j)$ and we have $q' \in c'$. If $c = c'$ holds, we therefore define $\mu(v, v) = ((q', q'), j + 1)$. Otherwise, we define $\mu(v, v) = ((p, q, p', q'), j + 1)$ for some $p' \in c$.

Let $\rho \in Plays(\mathbb{G}, \mu)$ be some initial play that is consistent with μ. Since $v_0 = ((q_0, q_0), 0)$ holds, where q_0 is the initial state of $\mathcal{A}_\varphi$, it follows inductively from the construction of μ and by definition of the delay-dominance game that we have both $p = q$ for all positions occurring in ρ, irrespective of whether the positions are of the form $((p, q), j)$, $((p, q, c), j)$,

$((p,q,c,c'),j)$, or $((p,q,c,q'),j)$, and $c=c'$ for all positions of the form $((p,q,c,c'),j)$ occurring in ρ. Thus, in particular, $f_{alt}(\rho_k) = f_{dom}(\rho_k)$ holds for all iterations $k \in \mathbb{N}_0$ and therefore $f_{dom}(\rho_k) \in F \rightarrow f_{alt}(\rho_k) \in F$ follows immediately for all $k \in \mathbb{N}_0$. Therefore, $\rho \in \mathbb{W}$ holds and hence, since we chose the play $\rho \in \mathit{Plays}(\mathbb{G},\mu)$ arbitrarily, μ is indeed a winning strategy for Duplicator in the delay-dominance game $\mathbb{G}$. Consequently, $s_i \trianglelefteq_{\mathcal{A}_\varphi,\gamma} s_i$ holds. □

In the remainder of this section, we address one of the other two important properties of a suitable dominance-like notion, namely that, in contrast to bounded dominance, it needs to imply remorsefree dominance. More precisely, we show that every delay-dominant strategy is also remorsefree dominant.

3.2.2. Delay-Dominance implies Remorsefree Dominance

Recall that one of the main weaknesses of bounded dominance is that every strategy, even a non-dominant one, is n-dominant if the bound n is chosen too small [DF14]: let φ be a specification that requires a minimal bound $m \in \mathbb{N}$. Let $\mathcal{A}_\varphi$ be a universal co-Büchi automaton with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$. Then, in particular, we have $m_{\mathcal{A}_\varphi}(comp(s_i,\gamma)) \geq m$ for all strategies s_i and all input sequences $\gamma \in (2^{I_i})^\omega$. Let $n \in \mathbb{N}_0$ be some bound. By definition, a process strategy s_i is n-dominant if, for all input sequences $\gamma \in (2^{I_i})^\omega$, either $m_{\mathcal{A}_\varphi}(comp(s_i,\gamma)) \leq n$ holds, or if we have $m_{\mathcal{A}_\varphi}(comp(t_i,\gamma)) > n$ for all alternative strategies t_i. Since m is the minimal bound of $\mathcal{A}_\varphi$ and thus $m_{\mathcal{A}_\varphi}(comp(s_i,\gamma)) \geq m$ for all strategies s_i, we have, in particular, $m_{\mathcal{A}_\varphi}(comp(s_i,\gamma)) > n$ for all strategies s_i if $n < m$ holds. Hence, for all bounds $n \in \mathbb{N}_0$ with $n < m$, every strategy is trivially n-dominant, even one that is not remorsefree dominant.

The main reason for this undesired result for bounded dominance is the need for an *explicit* bound $n \in \mathbb{N}_0$ and the fact that it is oftentimes challenging to determine a good bound for a synthesis task: it should be high enough to avoid that non-dominant strategies are trivially n-dominant, while it should be as small as possible to reduce the synthesis time. Delay-dominance does not require an explicit bound and thus does not suffer from this problem. In fact, every delay-dominant strategy is also remorsefree dominant. To prove this, we establish a relationship between strategies in the delay-dominance game and run trees in the underlying alternating co-Büchi automaton. To formalize the relationship conveniently, we first define a *projected play* of the delay-dominance game:

Definition 3.4 (Projected Play)**.**
Let φ be an LTL formula over atomic propositions V and let $\mathcal{A}_\varphi$ be an alternating co-Büchi automaton with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$. Let $\sigma,\sigma' \in (2^V)^\omega$. Let ρ be some play in the delay-dominance game $\mathbb{G} = (\mathcal{A}_\varphi,\sigma,\sigma')$. The *projected play* $\hat{\rho} \in (Q \times Q)^\omega$ is defined by $\hat{\rho}_k = \#_1(\hat{\rho}_{4k})$ for all $k \geq 0$. The *projected dominant play* $\hat{\rho}^{dom} \in Q^\omega$ of ρ and the *projected alternative play* $\hat{\rho}^{alt} \in Q^\omega$ of ρ are defined by $\hat{\rho}^{dom}_k = \#_2(\hat{\rho}_k)$ and $\hat{\rho}^{alt}_k = \#_1(\hat{\rho}_k)$ for all $k \geq 0$, respectively.

Intuitively, we obtain the projected play $\hat{\rho}$ from the play ρ by removing all positions that are not of the form $((p,q),j)$ and by projecting to the state tuple; thus removing the index j. The projected dominant play $\hat{\rho}^{dom}$ is then obtained by further projecting to the dominant state of the state tuples of $\hat{\rho}$, i.e., to q for a state tuple (p, q), while we further project to the alternative state in the projected alternative play $\hat{\rho}^{alt}$, i.e., to p for a state tuple (p, q). Utilizing projected plays, we now establish the following correspondence between strategies in the delay-dominance game and run trees: a strategy for player Duplicator in the delay-dominance game $(\mathcal{A}_\varphi, \sigma, \sigma')$ corresponds to a run tree of $\mathcal{A}_\varphi$ induced by σ'. Similarly, a strategy for player Spoiler in the same game corresponds to run tree of $\mathcal{A}_\varphi$ induced by σ. Formally:

Lemma 3.2. *Let φ be an LTL formula over atomic propositions V and let $\mathcal{A}_\varphi$ be an alternating co-Büchi automaton with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$. Let $\sigma, \sigma' \in (2^V)^\omega$. Let $\mathbb{G} = (\mathcal{A}_\varphi, \sigma, \sigma')$ be a delay-dominance game and let μ and μ' be strategies for Duplicator and Spoiler, respectively, in $\mathbb{G}$. Then, there exist run trees $r \in Runs(\mathcal{A}_\varphi, \sigma)$ and $r' \in Runs(\mathcal{A}_\varphi, \sigma')$ of $\mathcal{A}_\varphi$ such that we have both $Branches_{Inf}(r) = \{\hat{\rho}^{dom} \mid \rho \in Plays(\mathbb{G}, \mu)\}$ and $Branches_{Inf}(r') = \{\hat{\rho}^{alt} \mid \rho \in Plays(\mathbb{G}, \mu')\}$.*

Proof. Let $\mathcal{A}_\varphi = (Q, q_0, \delta, F)$. First, we consider Duplicator's strategy μ. We construct a Q-labeled tree $(\mathbb{T}, \ell)$ from the strategy μ as follows by defining the labeling of the root as well as of the successors of all nodes. The labeling of the root ε of $\mathbb{T}$ is defined by $\ell(\varepsilon) = q_0$. Let $x \in \mathbb{T}$ be a node of $\mathbb{T}$ with depth $k = |x|$. Let $\nu = pref(\mathbb{T}, x)$ be x's prefix in $\mathbb{T}$, i.e. the unique finite sequence of nodes in $\mathbb{T}$ that, starting from ε, reaches x. We define the labeling of the successor nodes $children(x)$ of x such that

$$\{\ell(x') \mid x' \in children(x)\} = \{\hat{\rho}^{dom}_{k+1} \mid \rho \in Plays(\mathbb{G}, \mu) \ \wedge\ \forall 0 \leq k' \leq k.\ \hat{\rho}^{dom}_{k'} = \ell(\nu_{k'})\}$$

holds. Next, we show that $(\mathbb{T}, \ell)$ is a run tree of $\mathcal{A}_\varphi$ induced by σ'. Let $x \in \mathbb{T}$ be some node of $\mathbb{T}$. Then, by construction of the delay-dominance game, we know that Duplicator controls the existential transitions of $\mathcal{A}_\varphi$ for σ', while the universal ones are controlled by Spoiler. Hence, since μ is a strategy of Duplicator, μ defines the existential choices in $\mathcal{A}_\varphi$ for σ'. Therefore, for every round of the delay-dominance game and thus for every time step $k \geq 0$, there exists a decision for the existential choices in $\mathcal{A}_\varphi$ for σ', namely the one defined by μ, such that all initial plays that are consistent with μ adhere to it. Moreover, as no strategy for Spoiler is given, for every round of the game the set of initial plays that are consistent with μ, i.e., the set $Plays(\mathbb{G}, \mu)$, defines *all* possible universal choices in $\mathcal{A}_\varphi$ for σ that fit in with the existential choice defined by μ as well as the history. Therefore, it follows that, for every node $x \in \mathbb{T}$ and its depth $k = |x|$, the set $\{\hat{\rho}^{dom}_{k+1} \mid \rho \in Plays(\mathbb{G}, \mu) \ \wedge\ \forall 0 \leq k' \leq k.\ \hat{\rho}^{dom}_{k'} = \ell(\nu_{k'})\}$ satisfies the formula

$$\bigvee_{c' \in \delta(\ell(x), \sigma'_k)} \bigwedge_{q' \in c'} q'.$$

Thus, by construction of the labeling of the successor nodes of node x defined above, the set $\{\ell(x') \mid x' \in children(x)\}$ satisfies this propositional formula as well. Hence, by definition of run trees, the Q-labeled tree $(\mathbb{T}, \ell)$ is indeed a run tree of $\mathcal{A}_\varphi$ induced by σ'. Intuitively, the dominant states of an initial play that is consistent with strategy μ thus evolve according to a branch of a run tree of $\mathcal{A}_\varphi$ induced by σ'. Furthermore, by construction of $(\mathbb{T}, \ell)$, we immediately obtain that $Branches_{Inf}(\mathbb{T}, \ell) = \{\hat{\rho}^{dom} \mid \rho \in Plays(\mathbb{G}, \mu)\}$ holds. Therefore, $(\mathbb{T}, \ell)$ is the desired run tree of $\mathcal{A}_\varphi$ induced by σ'.

Similarly, we can construct a Q-labeled tree $(\mathbb{T}', \ell')$ from Spoiler's strategy μ', which only differs slightly in the definition of the labeling of the successors of a node $x \in \mathbb{T}'$. Instead of utilizing the projected dominant play $\hat{\rho}^{dom}$, we employ the projected alternative play $\hat{\rho}^{dom}$. Due to the change of control for process strategy t_i in the construction of the delay-dominance game, Spoiler controls the existential transitions of $\mathcal{A}_\varphi$ for σ, while Duplicator controls the universal ones. Therefore, it follows completely analogous to the first part of this proof that $(\mathbb{T}', \ell')$ is the desired run tree of $\mathcal{A}_\varphi$ induced by σ. □

Vice versa, we can translate a run tree of an alternating co-Büchi automaton $\mathcal{A}_\varphi$ induced by some sequence $\sigma \in (2^V)^\omega$ of system variable valuations into a strategy for Spoiler in the delay-dominance game $(\mathcal{A}_\varphi, \sigma, \sigma')$, where $\sigma' \in (2^V)^\omega$ is some infinite sequence of valuations of system variables. Similarly, a run tree of $\mathcal{A}_\varphi$ induced by σ' corresponds to a strategy for Duplicator in the same delay-dominance game. Formally:

Lemma 3.3. *Let φ be an LTL formula over atomic propositions V. Let $\mathcal{A}_\varphi$ be an alternating co-Büchi automaton with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$. Let $\sigma, \sigma' \in (2^V)^\omega$. Let $\mathbb{G} = (\mathcal{A}_\varphi, \sigma, \sigma')$ be a delay-dominance game. Let $r \in Runs(\mathcal{A}_\varphi, \sigma')$ and $r' \in Runs(\mathcal{A}_\varphi, \sigma)$ be run trees of $\mathcal{A}_\varphi$. Then, there exist strategies μ and μ' for Duplicator and Spoiler, respectively, in the game $\mathbb{G}$ such that both $Branches_{Inf}(r) = \{\hat{\rho}^{dom} \mid \rho \in Plays(\mathbb{G}, \mu)\}$ and $Branches_{Inf}(r') = \{\hat{\rho}^{alt} \mid \rho \in Plays(\mathbb{G}, \mu')\}$ hold.*

Proof. Let $\mathcal{A}_\varphi = (Q, q_0, \delta, F)$. First, we consider the run tree r' of $\mathcal{A}_\varphi$ induced by σ. We construct a strategy μ' for Spoiler in the delay-dominance game $\mathbb{G}$ from r' as follows. Let $\nu \cdot v$ be a finite sequence of positions of $\mathbb{G}$'s game arena with $\nu \in P^*$ and $v \in P$. We only define μ' explicitly on sequences $\nu \cdot v$ that can occur in the delay-dominance game $\mathbb{G}$ and where v is controlled by Spoiler; on all other sequences, we define $\mu'(\nu, v) = v'$ for some arbitrary position $v' \in P$ that is a valid extension of $\nu \cdot v$. In the following, we thus assume that $\nu \cdot v$ is a prefix of a play that can occur in the game $\mathbb{G}$ and that v is of the form $((p, q), j)$ or $((p, q, c, c'), j)$. We map $\nu \cdot v$ to a prefix of a branch of the run tree r' if there is a compatible one: a *compatible branch* b of r' agrees with the finite projected alternative play $\hat{\nu}^{alt}$ up to point in time $|\nu| - 1$, i.e., we have $b_k = \hat{\nu}^{alt}_k$ for all k with $0 \leq k < |\nu|$. Note here that, slightly misusing notation, we apply the definition of a projected play also to the finite prefix ν of a play. Moreover, no matter whether v is of the form $((p, q, c), j)$ or $((p, q, c, c'), j)$, we have $b_{|\nu|} = p$ in a compatible branch. If there is no compatible branch in r', we again define $\mu(\nu \cdot v) = v'$ for some arbitrary position $v' \in P$ that is a valid extension of $\nu \cdot v$. Otherwise, the successors of p in b define the choice of μ: by definition, the set $\mathcal{S}$ of successors of the node labeled with p in b satisfies $\delta(p, \sigma_{|\nu|})$. Thus, there exists some set $c \in \delta(p, \sigma_{|\nu|})$ such that $p' \in \mathcal{S}$ holds for all $p' \in c$. If v is of the form $v = ((p, q), j)$, we thus define $\mu'(\nu \cdot v) = ((p, q, c), j)$. If v is of the form $v = ((p, q, c', c), j)$, we define $\mu'(\nu \cdot v) = v'$ for some arbitrary position $v' \in P$ that is a valid extension of $\nu \cdot v$. Note here that choosing an arbitrary successor for $\nu \cdot v$ for μ' is possible in this case since the choice defines a successor state for the dominant state q. Hence, the choice does not influence the projected alternative play. Since the existential choices in $\mathcal{A}_\varphi$ define the run tree, it follows immediately from the construction of μ' that $Branches_{Inf}(r') = \{\hat{\rho}^{alt} \mid \rho \in Plays(\mathbb{G}, \mu')\}$ holds.

Similarly, we can construct a strategy μ for Duplicator in the delay-dominance game $\mathbb{G}$ from the run tree r of $\mathcal{A}_\varphi$ induced by σ'. We define the compatibility of a branch analogously, yet, utilizing the projected dominant play $\hat{\nu}^{dom}$ instead of the projected alternative one $\hat{\nu}^{alt}$. If there exists a branch b of r that is compatible with the considered sequence $\nu \cdot v$, then, by definition, the set $\mathcal{S}$ of successors of the node labeled with q in b satisfies $\delta(q, \sigma'_{|\nu|})$. Thus, similar to the case above, there exists some $c' \in \delta(q, \sigma'_{|\nu|})$ such that $q' \in \mathcal{S}$ holds for all $q' \in c'$. If v is of the form $v = ((p, q, c), j)$, we therefore define $\mu(\nu \cdot v) = ((p, q, c, c'), j)$. In all other cases, i.e., if v is of the form $v = ((p, q, c, q'), j)$, if $v \notin P_0$, if there is no compatible branch for $\nu \cdot v$, or if $\nu \cdot v$ cannot occur in the game $\mathbb{G}$, then we choose an arbitrary successor position $v' \in P$ that is a valid extension of $\nu \cdot v$. Since Duplicator controls the existential choices in $\mathcal{A}_\varphi$ for σ' and Spoiler controls the existential choices in $\mathcal{A}_\varphi$ for σ, it follows analogously to the previous case that $\mathit{Branches}_{\mathit{Inf}}(r) = \left\{\hat{\rho}^{alt} \mid \rho \in \mathit{Plays}(\mathbb{G}, \mu)\right\}$ holds. □

Utilizing the observations on the relationship between strategies in a delay-dominance game and run trees in the underlying alternating co-Büchi automaton, we now show that every strategy s_i for a process $p_i \in \mathbb{P}$ that is delay-dominant is also remorsefree dominant. The main idea behind the result is that a winning strategy μ of Duplicator in the delay-dominance game defines a run tree of the alternating co-Büchi automaton induced by a computation of s_i such that all branches either visit only finitely many rejecting states or such that all rejecting states are matched eventually with a rejecting state in some branch of all run trees induced by an alternative strategy. Thus, s_i either satisfies the specification, or every alternative strategy does not satisfy it either. Formally:

Theorem 3.2. *Let φ be an LTL formula over atomic propositions V. Let $\mathcal{A}_\varphi$ be an alternating co-Büchi automaton with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$. Let s_i be a strategy for process $p_i \in \mathbb{P}$. If s_i is delay-dominant for $\mathcal{A}_\varphi$, then s_i is remorsefree dominant for φ.*

Proof. Let $\mathcal{A}_\varphi = (Q, q_0, \delta, F)$. Suppose that strategy s_i is delay-dominant for p_i $\mathcal{A}_\varphi$, while s_i is not remorsefree dominant for φ. Then, there exists an alternative strategy t_i for process p_i and two infinite sequences $\gamma \in (2^{I_i})^\omega$ and $\gamma' \in (2^{V \setminus V_i})^\omega$ such that $\mathit{comp}(s_i, \gamma) \cup \gamma' \not\models \varphi$ holds, while we have $\mathit{comp}(t_i, \gamma) \cup \gamma' \models \varphi$. Furthermore, since strategy s_i is delay-dominant for p_i and $\mathcal{A}_\varphi$ by assumption, there exists, for every infinite sequence $\gamma'' \in (2^{V \setminus V_i})^\omega$, a winning strategy for Duplicator in the delay-dominance game $(\mathcal{A}_\varphi, \mathit{comp}(t_i, \gamma) \cup \gamma'', \mathit{comp}(s_i, \gamma) \cup \gamma'')$. Therefore, in particular, Duplicator has a winning strategy μ in the delay-dominance game $\mathbb{G} = (\mathcal{A}_\varphi, \mathit{comp}(t_i, \gamma) \cup \gamma', \mathit{comp}(s_i, \gamma) \cup \gamma')$.

First, by Lemma 3.2, there exists a run tree $r \in \mathit{Runs}(\mathcal{A}_\varphi, \mathit{comp}(s_i, \gamma) \cup \gamma')$ of $\mathcal{A}_\varphi$ induced by $\mathit{comp}(s_i, \gamma) \cup \gamma'$ that reflects the choices for the existential transitions of $\mathcal{A}_\varphi$ that occur when reading $\mathit{comp}(s_i, \gamma) \cup \gamma'$ defined by Duplicator's winning strategy μ in the game $\mathbb{G}$. Moreover, we have $\mathit{Branches}_{\mathit{Inf}}(r) = \left\{\hat{\rho}^{dom} \mid \rho \in \mathit{Plays}(\mathbb{G}, \mu)\right\}$. Since $\mathit{comp}(s_i, \gamma) \cup \gamma' \not\models \varphi$ holds by assumption, all run trees of $\mathcal{A}_\varphi$ induced by $\mathit{comp}(s_i, \gamma) \cup \gamma'$ contain a branch that contains infinitely many visits to rejecting states. Thus, in particular, the run tree r for which $\mathit{Branches}_{\mathit{Inf}}(r) = \left\{\hat{\rho}^{dom} \mid \rho \in \mathit{Plays}(\mathbb{G}, \mu)\right\}$ holds contains a branch that visits rejecting states infinitely often. Hence, there is an initial play $\rho \in \mathit{Plays}(\mathbb{G}, \mu)$ in $\mathbb{G}$ that is consistent with Duplicator's strategy μ such that $\hat{\rho}^{dom}$ contains infinitely many visits to rejecting states. Therefore, it follows from the definition of projected dominant plays that the play ρ contains infinitely many visits to rejecting *dominant* states.

Next, since we have $comp(t_i, \gamma) \cup \gamma' \models \varphi$, there exists a run tree $r' \in Runs(\mathcal{A}_\varphi, comp(t_i, \gamma) \cup \gamma')$ of $\mathcal{A}_\varphi$ induced by $comp(t_i, \gamma) \cup \gamma'$ whose branches all visit only finitely many rejecting states. Then, by Lemma 3.3, there is a strategy μ' for Spoiler in the delay-dominance game $\mathbb{G}$ that reflects the choices of r' for the existential transitions of $\mathcal{A}_\varphi$ when reading $comp(t_i, \gamma) \cup \gamma'$. Moreover, we have $Branches_{Inf}(r') = \left\{\hat{\rho}^{alt} \mid \rho \in Plays(\mathbb{G}, \mu')\right\}$. Thus, since all branches of r' visit only finitely many rejecting states, it follows immediately from the construction of μ' that, for all initial plays $\rho \in Plays(\mathbb{G}, \mu')$ that are consistent with μ', we have that $\hat{\rho}^{alt}$ visits only finitely many rejecting states. Therefore, particularly for the unique initial play that is consistent with both μ and μ', it holds that $\hat{\rho}^{alt}$ visits only finitely many rejecting states. Hence, by definition of projected alternative plays, ρ visits only finitely many rejecting alternative states. However, as shown above, $\hat{\rho}^{dom}$ visits infinitely many rejecting states. Thus, there is a point in time $k \in \mathbb{N}_0$ such that $f_{dom}(\rho_k) \in F$ holds, while we have $f_{alt}(\rho_{k'}) \notin F$ for all $k' \geq k$. However, then μ is not a winning strategy for Duplicator; yielding a contradiction. □

Thus, every delay-dominant strategy is also remorsefree dominant and hence the notion of delay-dominance overcomes the main weakness of bounded dominance. Furthermore, recall that, given a realizable LTL specification φ, every strategy that is remorsefree dominant for φ is also winning for φ by Proposition 2.5. Together with Theorem 3.2 the same property follows immediately also for delay-dominant strategies:

Corollary 3.1. *Let φ be an LTL formula over atomic propositions V. Let $\mathcal{A}_\varphi$ be an alternating co-Büchi automaton with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$. If φ is realizable, then every strategy that is delay-dominant strategy for $\mathcal{A}_\varphi$ is winning for φ as well.*

Thus, from the point of view of obtaining meaningful strategies for the individual system processes, delay-dominance is a more suitable notion than bounded dominance. In compositional synthesis, however, it is not only crucial to obtain a meaningful process strategy but also to obtain a suitable strategy for the overall system when building the parallel composition of the individual process strategies. In the following section, we thus study the compositionality of delay-dominance for both safety and liveness properties.

3.3. Compositionality of Delay-Dominance

A critical shortcoming of remorsefree dominance is its non-compositionality for liveness properties. This restricts the usage of dominance-based compositional distributed synthesis algorithms to safety specifications, which are, in many cases, not expressive enough to formalize the system requirements. Delay-dominance, in contrast, is specifically designed to be compositional for more properties. This heavily relies on two facts. First, delay-dominance is not defined using the satisfaction of the given specification but on a more involved property on the visits of rejecting states. Second, delay-dominance is defined using a two-player game, and thus we require the existence of a *strategy* for Duplicator, i.e., determining which decisions to make for the existential choices of the delay-dominant strategy and the universal ones for the alternative strategy has to be possible without knowledge about the future input as well as the future decisions for the other choices.

More precisely, compositionality requires that whenever the parallel composition of two strategies s_1 and s_2 for system processes $p_1 \in P^-$ and $p_2 \in P^-$, respectively, does not satisfy the strategy requirment – that is, for instance, remorsefree dominance, bounded dominance, or delay-dominance – we are able to *blame* at least one of the processes for being responsible for violating that strategy requirement. Otherwise, none of the processes ever behaves incorrectly with respect to the strategy requirement, and thus none of the processes violates the strategy requirement. Hence, the parallel composition of two strategies that satisfy the strategy requirement would then not necessarily satisfy it.

Example 3.6. Reconsider the message-sending system from the running example and the strategy property remorsefree dominant. Furthermore, consider the strategies t_1 and t_2 that wait to receive the respective other messages before sending their own one (see Figure 3.1b). Their parallel composition $t_1 \mid\mid t_2$ never sends any message and thus violates the specification $\varphi = \Diamond m_1 \land \Diamond m_2$ on every input sequence. However, none of the processes can be blamed for being responsible for violating the properties of remorsefree dominance: even if process p_i would send its message m_i eventually, the specification is still not satisfied since message m_{3-i} has not been send yet. Thus, as long as p_i did not receive m_{3-i}, it is not required to eventually send m_i. The same, however, also holds for system process p_{3-i}. Note here that it is crucial that, although p_i is required to send m_i eventually if it receives m_{3-i}, process p_{3-i} is not required to send m_{3-i} in the first place; resulting in the deadlock situation where both processes wait on each other indefinitely. Nevertheless, both t_1 and t_2 are remorsefree dominant strategies: the processes are not required to send their message when confronted with the behavior of the other process defined by a computation of the parallel composition $t_1 \mid\mid t_2$ of t_1 and t_2, i.e., when reading an input of the form $comp(t_1 \mid\mid t_2, \gamma) \cap I_i$. △

Let s_1 and s_2 be two strategies for processes p_1 and p_2, respectively, and suppose that their parallel compositions $s_1 \mid\mid s_2$ does not satisfy the process requirement. Intuitively, we can then blame at least one of the processes p_1 and p_2 for violating the strategy requirement if there exists a bad prefix of a computation of their parallel composition $s_1 \mid\mid s_2$ for the strategy requirement, i.e., a prefix of a computation of $s_1 \mid\mid s_2$ such that all of its infinite extensions violate the strategy requirement. For remorsefree dominance, for instance, a bad prefix of a computation $comp(s_1 \mid\mid s_2, \gamma)$ is a finite prefix η of $comp(s_1 \mid\mid s_2, \gamma)$ such that all infinite extensions $\sigma \in (2^{V_1 \cup V_2})^\omega$ of η that agree with γ on the inputs of $p_1 \mid\mid p_2$, i.e., for which $\sigma \cap I_{p_1 \mid\mid p_2} = \gamma \cap I_{p_1 \mid\mid p_2}$ holds, violate the specification while there exists an alternative strategy t for $p_1 \mid\mid p_2$ such that $comp(t, \gamma)$ satisfies the specification. Note here that since remorsefree dominance only considers the satisfaction of a specification, the existence of a bad prefix for remorsefree dominance boils down to the existence of a bad prefix for the considered specification. Since liveness properties do not have bad prefixes by definition (see Section 2.3), there thus clearly do not exist bad prefixes for remorsefree dominance for liveness properties.

Delay-dominance, in contrast, takes an alternating co-Büchi automaton, which describes the system specification, into account and relates the visits of the automaton to rejecting states induced by the computation of a delay-dominant strategy to those induced by a computation of an alternative strategy. Thus, the non-existence of bad prefixes for liveness properties does not necessarily result in the absence of bad prefixes for delay-dominance.

Example 3.7. Reconsider the message-sending system from the running example and its specification $\varphi = \Diamond m_1 \land \Diamond m_2$. Furthermore, consider the alternating co-Büchi automaton $\mathcal{A}_\varphi$ with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$. depicted in Figure 3.2. Although $\mathcal{L}(\varphi)$ is a liveness property and thus does not have a bad prefix by definition, the automaton $\mathcal{A}_\varphi$ ensures, intuitively, the existence of bad prefixes for delay-dominance: let s_i be a strategy for system process $p_i \in P^-$ that is not delay-dominant for $\mathcal{A}_\varphi$. Then, there exists an input sequence $\gamma \in (2^{\{m_{3-i}\}})^\omega$ and an alternative strategy t_i for p_i such that Duplicator does not have a winning strategy in the delay-dominance game $\mathbb{G} = (\mathcal{A}_\varphi, \mathit{comp}(t_i, \gamma), \mathit{comp}(s_i, \gamma))$. As outlined in Example 3.5, for such an input sequence $\gamma \in (2^{\{m_{3-i}\}})^\omega$ it holds that m_{3-i} occurs in $\mathit{comp}(s_i, \gamma)$ before m_i. Let $k \geq 0$ be the earliest point in time at which m_{3-i} occurs in $\mathit{comp}(s_i, \gamma)$ while m_i did not occur so far. The prefix of $\mathit{comp}(s_i, \gamma)$ up to the point in time k, i.e., $\mathit{comp}(s_i, \gamma)_{|k+1}$, is then a bad prefix of $\mathit{comp}(s_i, \gamma)$ for delay-dominance in $\mathcal{A}_\varphi$: since $\mathcal{A}_\varphi$ is deterministic, every play in the delay-dominance game stays in state q_0 in the dominant states up to the point in time k at which m_{3-i} occurs and then moves to either q_1 or q_2, depending on whether $i = 1$ or $i = 2$ holds. It then stays there until m_i occurs and moves to q_3 afterward. An alternative strategy that outputs m_i on input sequence γ at point in time k, i.e., at the exact same point in time at which m_{3-i} occurs in γ, induces a move from q_1 directly to q_3, thus avoiding the visit of a rejecting state q_1 or q_2. Hence, no matter how system process p_1 behaves after entering q_1 or q_2, respectively, there is an alternative strategy that causes Duplicator to lose the delay-dominance game for input sequence γ, namely the one that, intuitively, predicts the point in time at which m_{3-i} is sent, and that sends m_i at the very same point in time. Thus, behaving as s_i on γ up to point in time k always results in Duplicator losing the delay-dominance game for some alternative strategy. Consequently, behaving as s_i on γ up to point in time k always results in not being delay-dominant for $\mathcal{A}_\varphi$. △

In the following, we first establish that delay-dominance is indeed compositional if bad prefixes exist. Afterward, we then study which properties the alternating co-Büchi automaton needs to satisfy to allow for bad prefixes. Thus, we first define a property that formalizes the existence of a bad prefix with respect to delay-dominance.

Definition 3.5 (Bad Prefixes for Delay-Dominance).
Let φ be an LTL formula over atomic propositions V. Let $\mathcal{A}_\varphi$ be an alternating co-Büchi automaton with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$. Then, $\mathcal{A}_\varphi$ *ensures bad prefixes for delay-dominance* if, and only if, for all $p_i \in \mathbb{P}$, $\gamma \in (2^{V \setminus O_i})^\omega$, and $\sigma \in (2^{O_i})^\omega$ for which there exists some $\sigma' \in (2^{V_i})^\omega$ such that Duplicator loses the delay-dominance game $(\mathcal{A}_\varphi, \sigma' \cup \gamma, \sigma \cup \gamma)$, there is some finite prefix $\eta \in (2^{O_i})^*$ of σ such that for all infinite extensions $\hat{\sigma} \in (2^{O_i})^\omega$ of η, there is some $\sigma'' \in (2^{O_i})^\omega$ such that Duplicator loses the delay-dominance game $(\mathcal{A}_\varphi, \sigma'' \cup \gamma, \hat{\sigma} \cup \gamma)$.

If the considered alternating co-Büchi automaton $\mathcal{A}_\varphi$ satisfies the bad prefix property for delay-dominance, then the notion of delay-dominance is indeed compositional, i.e., then the parallel composition of two delay-dominant strategies is guaranteed to be delay-dominant as well. The main idea of the proof is to utilize the bad prefix property to argue that, if the parallel composition $s_1 \mid\mid s_2$ of two strategies is not delay-dominant, then at least one of the processes can be blamed for being responsible for violating the properties of delay-dominance.

No matter which of the processes is responsible – or even both – we can then conclude from the properties of process strategies and the fact that they are represented by Moore transducers that the strategy of the respective process cannot be delay-dominant:

Theorem 3.3. *Let φ be an LTL formula over atomic propositions V. Let $\mathcal{A}_\varphi$ be an alternating co-Büchi automaton with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$. Let $p_1, p_2 \in \mathbb{P}$ be processes and let s_1 and s_2 be strategies for them. If both s_1 and s_2 are delay-dominant for $\mathcal{A}_\varphi$ and p_1 and p_2, respectively, and if $\mathcal{A}_\varphi$ ensures bad prefixes for delay-dominance, then $s_1 \mid\mid s_2$ is delay-dominant for $\mathcal{A}_\varphi$ and $p_1 \mid\mid p_2$.*

Proof. For the sake of readability, let $I_{1,2} = (I_1 \cup I_2) \setminus (O_1 \cup O_2)$ be the set of inputs of $p_1 \mid\mid p_2$, let $O_{1,2} = O_1 \cup O_2$ be the set of outputs of $p_1 \mid\mid p_2$, and let $V_{1,2} = I_{1,2} \cup O_{1,2}$ be the set of variables of $p_1 \mid\mid p_2$. Let $\mathcal{A}_\varphi = (Q, q_0, \delta, F)$. Suppose that $s_1 \mid\mid s_2$ is not delay-dominant for $\mathcal{A}_\varphi$ and $p_1 \mid\mid p_2$. Then, there is some sequence $\gamma' \in (2^{V \setminus V_{1,2}})^\omega$ of variables that cannot be observed by $p_1 \mid\mid p_2$, some alternative strategy t for $p_1 \mid\mid p_2$, and some infinite input sequence $\gamma \in (2^{I_{1,2}})^\omega$ such that there is no winning strategy for Duplicator in the delay-dominance game $\mathbb{G} = (\mathcal{A}_\varphi, \mathit{comp}(t, \gamma) \cup \gamma', \mathit{comp}(s_1 \mid\mid s_2, \gamma) \cup \gamma')$.

By assumption $\mathcal{A}_\varphi$ ensures bad prefixes for delay-dominance and thus there exists a finite prefix $\nu \in (2^V)^*$ of $\mathit{comp}(s_1 \mid\mid s_2, \gamma) \cup \gamma'$ such that for all infinite extensions $\hat{\sigma} \in (2^V)^\omega$ of ν with $\hat{\sigma} \cap (V \setminus O_{1,2}) = (\mathit{comp}(s_1 \mid\mid s_2, \gamma) \cup \gamma') \cap (V \setminus O_{1,2})$, there exists some sequence $\sigma'' \in (2^V)^\omega$ with $\hat{\sigma} \cap (V \setminus O_{1,2}) = \sigma' \cap (V \setminus O_{1,2})$ such that Duplicator loses the delay-dominance game $(\mathcal{A}_\varphi, \sigma'', \hat{\sigma})$. In particular, there thus exists a *smallest*, i.e., shortest, such bad prefix of $\mathit{comp}(s_1 \mid\mid s_2, \gamma) \cup \gamma'$. Let $\eta \cdot \delta \in (2^V)^*$ be this smallest such bad prefix, where $\eta \in (2^V)^*$ and $\delta \in 2^V$ holds. Since $\eta \cdot \delta$ is a bad prefix for delay-dominance by construction, it holds that for all infinite extensions $\hat{\sigma} \in (2^V)^\omega$ of $\eta \cdot \delta$ with $\hat{\sigma} \cap (V \setminus O_{1,2}) = (\mathit{comp}(s_1 \mid\mid s_2, \gamma) \cup \gamma') \cap (V \setminus O_{1,2})$, there exists some infinite sequence $\sigma'' \in (2^V)^\omega$ with $\hat{\sigma} \cap (V \setminus O_{1,2}) = \sigma'' \cap (V \setminus O_{1,2})$ such that Duplicator loses the delay-dominance game $(\mathcal{A}_\varphi, \sigma'', \hat{\sigma})$. Furthermore, since $\eta \cdot \delta$ is the smallest such prefix, there exists an infinite extension $\tilde{\sigma} \in (2^V)^\omega$ of η with $\tilde{\sigma} \cap (V \setminus O_{1,2}) = (\mathit{comp}(s_1 \mid\mid s_2, \gamma) \cup \gamma') \cap (V \setminus O_{1,2})$ such that Duplicator wins the delay-dominance game $(\mathcal{A}_\varphi, \sigma''', \hat{\sigma}')$ for all $\sigma''' \in (2^V)^\omega$ with $\hat{\sigma} \cap (V \setminus O_{1,2}) = \sigma''' \cap (V \setminus O_{1,2})$.

Suppose that $\eta \cdot \delta$ is the empty sequence. In that case, for all infinite sequences $\sigma' \in (2^V)^\omega$ that agree with $\gamma \cup \gamma'$ on the variables in $V \setminus O_{1,2}$, Duplicator does not have a winning strategy in the delay-dominance game $(\mathcal{A}_\varphi, \mathit{comp}(t, \gamma) \cup \gamma', \sigma')$. However, then Duplicator particularly does not have a winning strategy in the delay-dominance game $(\mathcal{A}_\varphi, \mathit{comp}(t, \gamma) \cup \gamma', \mathit{comp}(t, \gamma) \cup \gamma')$; contradicting that every strategy delay-dominates itself by Lemma 3.1. Hence, $\eta \cdot \delta$ cannot be the empty sequence and thus $|\eta \cdot \delta| > 0$. Let $m = |\eta \cdot \delta|$ be the length of $\eta \cdot \delta$. The last position δ of the prefix $\eta \cdot \delta$ – which is guaranteed to exist since $\eta \cdot \delta$ is not the empty sequence – contains decisions of both processes p_1 and p_2 defined by their delay-dominant strategies s_1 and s_2, respectively. We distinguish the following two cases:

1. There exists an infinite extension $\hat{\sigma} \in (2^V)^\omega$ of η with $\sigma_{m-1} \cap (V \setminus O_1) = \delta \cap (V \setminus O_1)$ and $\hat{\sigma} \cap (V \setminus O_{1,2}) = \gamma \cup \gamma'$ such that Duplicator has a winning strategy in the delay-dominance game $(\mathcal{A}_\varphi, \mathit{comp}(t, \gamma) \cup \gamma', \hat{\sigma})$. Hence, intuitively, it is the fault of process p_1 and thus, in particular, of its strategy s_1, that Duplicator loses the game delay-dominance $\mathbb{G}$. Let t' be a strategy for $p_1 \mid\mid p_2$ that produces the sequence $\hat{\sigma} \cap V_i$ on input sequence γ, i.e., a strategy

with $comp(t', \gamma) \cup \sigma = \hat{\sigma}$. For the sake of readability, let $\gamma^{s_2} = comp(s_1 \mid\mid s_2, \gamma) \cap O_2$ and let $\gamma^{t'_2} = comp(t', \gamma) \cap O_2$. Since we have $\delta \cap (V \setminus O_1) = \sigma_{m-1} \cap (V \setminus O_1)$ by assumption, we obtain that $\eta \cdot \delta$ and $\hat{\sigma}_{|m}$ agree on the variables in $V \setminus O_1$ and therefore, in particular, the finite sequences $comp(s_1 \mid\mid s_2, \gamma)_{|m}$ and $comp(t', \gamma)_{|m}$ agree on the variables in $V \setminus O_1$. Hence, it follows from the construction of γ^{s_2} and $\gamma^{t'_2}$ that $(\gamma \cup \gamma^{s_2})_{|m} = (\gamma \cup \gamma^{t'_2})_{|m}$ holds. Since strategies cannot look into the future,strategy s_1 thus cannot behave differently on input sequences $(\gamma \cup \gamma^{s_2}) \cap I_1$ and $(\gamma \cup \gamma^{t'_2}) \cap I_1$ up to point in time $m - 1$. For the sake of readability, let $\rho^s, \rho^{t'} \in (2^{V \setminus O_1})^\omega$ be the infinite sequences of valuations of variables outside the control of p_1 defined by $\rho^s := \gamma \cup \gamma^{s_2} \cup \gamma'$ and $\rho^{t'} := \gamma \cup \gamma^{t'_2} \cup \gamma'$. Then,

$$comp(s_1, \rho^s \cap I_1)_{|m} \cup (\rho^s_{|m} \cap (V \setminus V_1)) = comp(s_1, \rho^{t'} \cap I_1)_{|m} \cup (\rho^{t'}_{|m} \cap (V \setminus V_1))$$

follows. Hence, since $\eta \cdot \delta$ is a finite prefix of $comp(s_1 \mid\mid s_2, \gamma) \cup \gamma'$ by definition and since we have $comp(s_1, \rho^s \cap I_1) \cup (\rho^s \cap (V \setminus V_1)) = comp(s_1 \mid\mid s_2, \gamma) \cup \gamma'$ by construction of γ^{s_2} and ρ^s a well as by definition of computations of strategies, $comp(s_1, \rho^{t'} \cap I_1) \cup (\rho^{t'} \cap (V \setminus V_1))$ is an infinite extension of $\eta \cdot \delta$. Furthermore, since clearly $(comp(s_1 \mid\mid s_2, \gamma) \cup \gamma') \cap (V \setminus O_i) = \gamma \cup \gamma'$ holds by definition of computations of strategies, it follows immediately that we have

$$(comp(s_1, \rho^{t'} \cap I_1) \cup (\rho^{t'} \cap (V \setminus V_1))) \cap (V \setminus O_1) = (\gamma \cup \gamma') \cap (V \setminus O_1),$$

i.e. that $comp(s_1, \rho^{t'} \cap I_1) \cup (\rho^{t'} \cap (V \setminus V_1))$ agrees with $\gamma \cup \gamma'$ on the variables in $V \setminus O_1$. Therefore, by construction of the finite prefix $\eta \cdot \delta$, Duplicator loses the delay-dominance game $\mathbb{G}' = (\mathcal{A}_\varphi, comp(t, \gamma) \cup \gamma', comp(s_1, \rho^{t'} \cap I_1) \cup (\rho^{t'} \cap (V \setminus V_1)))$. However, by construction of the strategy t', Duplicator has a winning strategy μ in the delay-dominance game $(\mathcal{A}_\varphi, comp(t, \gamma) \cup \gamma', comp(t', \gamma) \cup \gamma')$. Let t'_1 be a strategy for process p_1 such that $comp(t', \gamma) \cap V_1 = comp(t'_1, \rho^{t'} \cap I_1)$ holds. Then, since s_1 is delay-dominant for p_1 and $\mathcal{A}_\varphi$ by assumption, s_1 particularly delay-dominates t'_1 on input $\rho^{t'} \cap I_1$ for sequence $\rho^{t'} \cap (V \setminus V_1)$ and therefore Duplicator has a winning strategy μ' in the delay-dominance game $(\mathcal{A}_\varphi, comp(t'_1, \rho^{t'} \cap I_1) \cup (\rho^{t'} \cap (V \setminus V_1)), comp(s_1, \rho^{t'} \cap I_1) \cup (\rho^{t'} \cap (V \setminus V_1))$. Since $comp(t'_1, \rho^{t'} \cap I_1) \cup (\rho^{t'} \cap (V \setminus V_1)) = comp(t', \gamma) \cup \gamma'$ holds by construction of the process strategy t'_1, we can thus combine the strategies μ and μ', i.e., the winning strategies of Duplicator in the two delay-dominance games $(\mathcal{A}_\varphi, comp(t, \gamma) \cup \gamma', comp(t', \gamma) \cup \gamma')$ and $(\mathcal{A}_\varphi, comp(t'_1, \rho^{t'} \cap I_1) \cup (\rho^{t'} \cap (V \setminus V_1)), comp(s_1, \rho^{t'} \cap I_1) \cup (\rho^{t'} \cap (V \setminus V_1))$, to a strategy μ'' for Duplicator in the delay-dominance game $\mathbb{G}'$. Furthermore, since μ and μ' are winning in the respective games, it follows immediately that for all initial plays $\rho \in Plays(\mathbb{G}', \mu'',)$ that are consistent with Duplicator's combined strategy μ'' it holds that whenever $f_{dom}(\rho_k) \in F$ holds for a point in time $k \in \mathbb{N}_0$, then there is a point in time $k' \in \mathbb{N}_0$ with $k' \geq k$ such that $f_{alt}(\rho_{k'}) \in F$ holds. Thus, μ'' is a winning strategy for Duplicator in the delay-dominance game $\mathbb{G}'$; contradicting that Duplicator loses $\mathbb{G}'$.

2. There is no infinite extension $\hat{\sigma} \in (2^{V_{1,2}})^\omega$ of η with $\hat{\sigma}_{m-1} \cap (V \setminus O_1) = \delta \cap (V \setminus O_1)$ and $\hat{\sigma} \cap (V \setminus O_{1,2}) = \gamma \cup \gamma'$ such that Duplicator has a winning strategy in the delay-dominance game $(\mathcal{A}_\varphi, comp(t, \gamma) \cup \gamma', \hat{\sigma})$. Hence, intuitively, it is (at least also) the fault of process p_2 and thus, in particular, of its strategy s_2, that Duplicator loses the game $\mathbb{G}$. By construction of the finite prefix $\eta \cdot \delta$, there exists an infinite extension $\sigma' \in (2^V)^\omega$ of η such that

Duplicator has a winning strategy in the delay-dominance game $(\mathcal{A}_\varphi, comp(t,\gamma) \cup \gamma', \sigma')$. Let t' be a process strategy for $p_1 \,||\, p_2$ that produces $\sigma' \cap (V \setminus V_i)$ on input γ, i.e., a strategy with $comp(t',\gamma) \cup \gamma' = \sigma'$. For the sake of readability, let $\gamma^{s_1} = comp(s_1 \,||\, s_2, \gamma) \cap O_1$ and let $\gamma^{t'_1} = comp(t',\gamma) \cap O_1$. By definition of t', the sequence $comp(t',\gamma) \cup \gamma'$ is an infinite extension of η. Thus, since $\eta \cdot \delta$ is a prefix of $comp(s_1 \,||\, s_2, \gamma) \cup \gamma'$ by definition, we have $comp(t',\gamma)_{|m-1} = comp(s_1 \,||\, s_2, \gamma)_{|m-1}$. Hence, it follows from the construction of γ^{s_1} and $\gamma^{t'_1}$ that $(\gamma \cup \gamma^{s_1})_{|m-1} = (\gamma \cup \gamma^{t'_1})_{|m-1}$ holds. Since strategies cannot look into the future, s_2 thus cannot behave differently on input sequences $(\gamma \cup \gamma^{s_1}) \cap I_2$ and $(\gamma \cup \gamma^{t'_1}) \cap I_2$ up to point in time $m-2$. For the sake of readability, let $\rho^s, \rho^{t'} \in (2^{V \setminus O_2})^\omega$ be the infinite sequences of valuations of variables outside the control of p_2 defined by $\rho^s := \gamma \cup \gamma^{s_2} \cup \sigma$ and $\rho^{t'} := \gamma \cup \gamma^{t'_2} \cup \sigma$. Then,

$$comp(s_2, \rho^s \cap I_2)_{|m-1} \cup (\rho^s_{|m-1} \cap (V \setminus V_2)) = comp(s_2, \rho^{t'} \cap I_2)_{|m-1} \cup (\rho^{t'}_{|m-1} \cap (V \setminus V_2))$$

follows. Hence, $comp(s_2, \rho^{t'} \cap I_2) \cup (\rho^{t'} \cap (V \setminus V_2))$ is an infinite extension of η as well. Since we consider process strategies that are represented by Moore transducers, s_2 cannot react directly to an input. In particular, s_2 can thus, at point in time $m-1$, not behave differently when reading $\rho^s \cap I_2$ and $\rho^{t'} \cap I_2$; even if $\rho^s \cap I_2$ and $\rho^{t'} \cap I_2$ differ at point in time $m-1$. Consequently,

$$comp(s_2, \rho^s \cap I_2)_{m-1} \cap O_2 = comp(s_2, \rho^{t'} \cap I_2)_{m-1} \cap O_2$$

holds. Furthermore, we have $\rho^s \cap (I_{1,2} \cup (V \setminus V_{1,2}) = \rho^{t'} \cap (I_{1,2} \cup (V \setminus V_{1,2})$ by construction of ρ^s and $\rho^{t'}$ and thus, since $I_{1,2} \cup (V \setminus V_{1,2}) = V \setminus O_{1,2}$ holds by definition of $V_{1,2}$ as well as by definition of architectures, ρ^s and $\rho^{t'}$ agree on the variables in $V \setminus O_{1,2}$. Therefore, it follows from the definition of computations of strategies that $comp(s_2, \rho^s \cap I_2)_{m-1} \cup (\rho^s \cap (V \setminus V_2))$ and $comp(s_2, \rho^{t'} \cap I_2)_{m-1} \cup (\rho^{t'} \cap (V \setminus V_2))$ agree on the variables in $V \setminus O_{1,2}$. By definition of $O_{1,2}$, we have $(V \setminus O_{1,2}) \cup O_2 = V \setminus O_1$ and thus $comp(s_2, \rho^s \cap I_2)_{m-1} \cup (\rho^s \cap (V \setminus V_2))$ and $comp(s_2, \rho^{t'} \cap I_2)_{m-1} \cup (\rho^{t'} \cap (V \setminus V_2))$ agree on the variables in $V \setminus O_1$ as well. For the sake of readability, let $\delta' = comp(s_2, \rho^{t'} \cap I_2))_{m-1} \cup (\rho^{t'}_{m-1} \cap (V \setminus V_2))$. Then, since we have $\delta = comp(s_2, \rho^s \cap I_2)_{m-1} \cup (\rho^s_{m-1} \cap (V \setminus V_2))$ holds by definition of the prefix $\eta \cdot \delta$ as well as by construction of ρ^s, $\delta' \cap (V \setminus O_1) = \delta \cap (V \setminus O_1)$ follows. Furthermore, the sequence $comp(s_2, \rho^{t'} \cap I_2)) \cup (\rho^{t'} \cap (V \setminus V_2))$ is an infinite extension of $\eta \cdot \delta'$. By construction of the strategy t', Duplicator has a winning strategy μ in the delay-dominance game $(\mathcal{A}_\varphi, comp(t,\gamma) \cup \gamma', comp(t',\gamma) \cup \gamma')$. Let t'_2 be a strategy for process p_2 such that $comp(t',\gamma) \cap V_2 = comp(t'_2, \rho^{t'} \cap I_1)$ holds. Then, since s_2 is delay-dominant for p_2 and $\mathcal{A}_\varphi$ by assumption, s_2 particularly delay-dominates t'_2 on input $\rho^{t'} \cap I_2$ for sequence $\rho^{t'} \cap (V \setminus V_2)$ and therefore Duplicator has a winning strategy μ' in the delay-dominance game $(\mathcal{A}_\varphi, comp(t'_2, \rho^{t'} \cap I_2) \cup (\rho^{t'} \cap (V \setminus V_2)), comp(s_2, \rho^{t'} \cap I_2) \cup (\rho^{t'} \cap (V \setminus V_2)))$. Similar to the previous case, we can combine μ and μ' to a winning strategy μ'' for Duplicator in the delay-dominance game $(\mathcal{A}_\varphi, comp(t,\gamma) \cup \gamma', comp(s_2, \rho^{t'} \cap I_2) \cup (\rho^{t'} \cap (V \setminus V_1))$. Thus, $comp(s_2, \rho^{t'} \cap I_2) \cup (\rho^{t'} \cap (V \setminus V_1))$ is an infinite extension $\hat{\sigma} \in (2^V)^\omega$ of η with $\hat{\sigma}_{m-1} \cap (V \setminus O_1) = \delta \cap (V \setminus O_1)$ and $\hat{\sigma} \cap (V \setminus O_{1,2}) = \gamma \cup \gamma'$ such that Duplicator wins the delay-dominance game $(\mathcal{A}_\varphi, comp(t,\gamma) \cup \gamma', \hat{\sigma})$; contradicting the assumption that no such infinite extension exists.

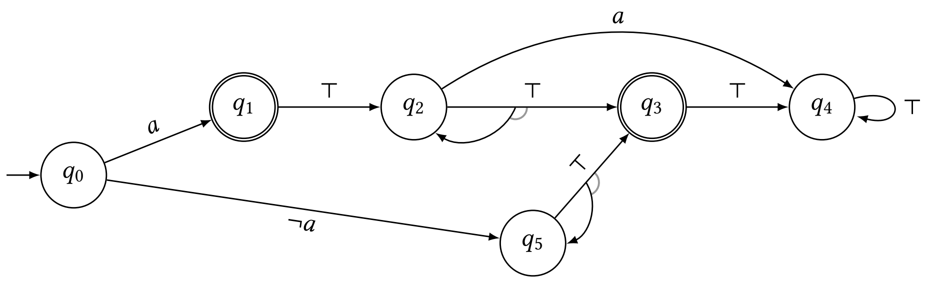


Figure 3.5.: Alternating co-Büchi automaton $\mathcal{A}$ over alphabet $\{a, b\}$ that does not ensure bad prefixes for delay-dominance. Universal choices are depicted with a gray arc.

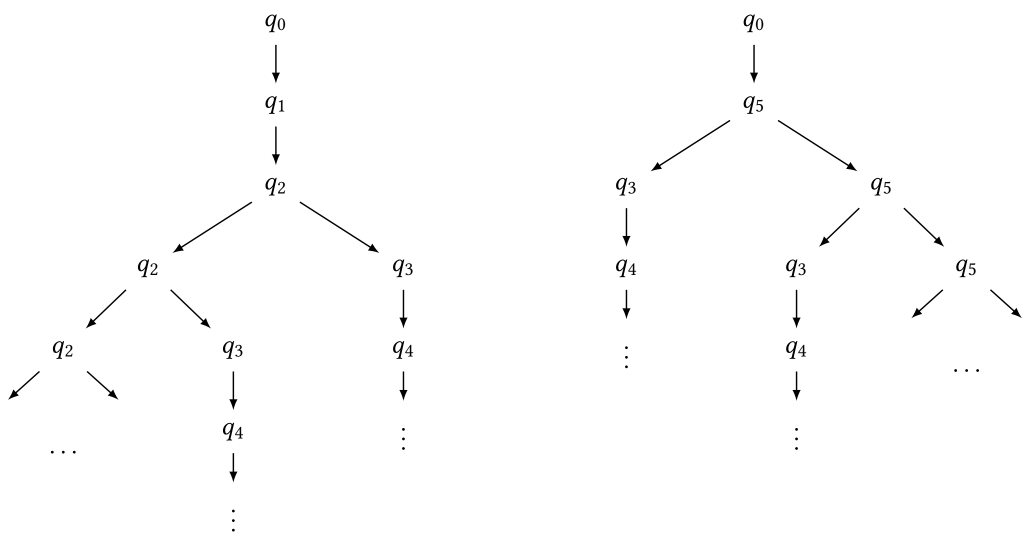
Hence, no matter which of the processes p_1 and p_2 can be blamed for being responsible for Duplicator losing the delay-dominance game $\mathbb{G}$, which determines whether or not $s_1 \mid\mid s_2$ delay-dominates t on input γ when considering γ', we obtain a contradiction. Thus, for all $\gamma \in (2^{I_{1,2}})^\omega$, all $\gamma' \in (2^{V \setminus V_{1,2}})^\omega$, and all strategies t for $p_1 \mid\mid p_2$, Duplicator has a winning strategy in the respective delay-dominance game $(\mathcal{A}_\varphi, \mathit{comp}(t, \gamma) \cup \gamma', \mathit{comp}(s_1 \mid\mid s_2, \gamma) \cup \gamma')$. Therefore, it follows that $s_1 \mid\mid s_2$ is delay-dominant for $\mathcal{A}_\varphi$ and $p_1 \mid\mid p_2$; concluding the proof. □

We have thus shown that delay-dominance is compositional for all specifications given as alternating co-Büchi automata that ensure bad prefixes for delay-dominance according to Definition 3.5. From Theorems 3.2 and 3.3 it now follows immediately that the parallel composition of two delay-dominant strategies is also remorsefree dominant if the considered alternating co-Büchi automaton representing the specification ensures bad prefixes:

Corollary 3.2. *Let φ be an LTL formula over atomic propositions V. Let $\mathcal{A}_\varphi$ be an alternating co-Büchi automaton with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$ that ensures bad prefixes for delay-dominance. Let s_1 and s_2 be delay-dominant strategies for $\mathcal{A}_\varphi$ and processes p_1 and p_2, respectively. Then, $s_1 \mid\mid s_2$ is remorsefree dominant for φ and $p_1 \mid\mid p_2$.*

Moreover, recall that, given a realizable LTL specification φ, every strategy that is delay-dominant for an alternating co-Büchi automaton $\mathcal{A}_\varphi$ with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$ is also winning for φ by Corollary 3.1. Hence, together with Theorem 3.3, it follows that given a specification φ and an alternating co-Büchi automaton $\mathcal{A}_\varphi$ with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$ that ensures bad prefixes for delay-dominance, the parallel composition of delay-dominant strategies for $\mathcal{A}_\varphi$ and all system processes of a distributed system is winning if φ is realizable. Hence, delay-dominance is a notion that can be soundly used for dominance-based compositional synthesis approaches when ensuring the bad prefix criterion.

As already pointed out above, there are many more properties for which there exists an alternating co-Büchi automaton that ensures bad prefixes for delay-dominance than properties that have a "classical" bad prefix. By definition, no liveness property has a classical bad prefix, yet, for many of them, there exist alternating co-Büchi automata that ensure bad prefixes for delay-dominance. In the following, however, we consider an alternating co-Büchi automaton that does not ensure bad prefixes:



(a) Run tree induced by $comp(s_i, \gamma)$.

(b) Run tree induced by $comp(t_i, \gamma)$.

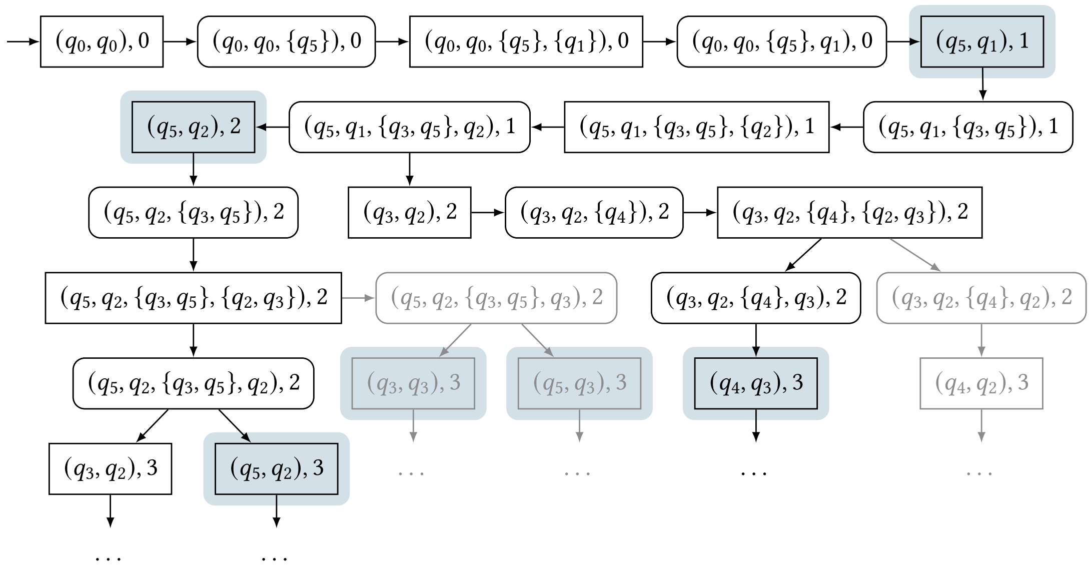
Figure 3.6.: Run trees of the alternating co-Büchi automaton $\mathcal{A}$ depicted in Figure 3.5 induced by $comp(s_i, \gamma)$ and $comp(t_i, \gamma)$ from Example 3.8, respectively.

Example 3.8. Let $V = \{a, b\}$ be a set of variables. Consider the alternating co-Büchi automaton $\mathcal{A}$ over alphabet V depicted in Figure 3.5. Let p_i be some process with inputs $I_i = \{b\}$ and outputs $O_i = \{a\}$. Let s_i be a strategy for p_i that outputs a in the very first time step and never outputs a afterward. That is, irrespective of the input sequence $\gamma \in (2^{I_i})^\omega$, the computation of s_i is given by $comp(s_i, \gamma) = \{a\}\emptyset^\omega$. Let t_i be an alternative strategy for p_i that never outputs a, i.e., $comp(t_i, \gamma) = \emptyset^\omega$ holds for all $\gamma \in (2^{I_i})^\omega$. The run trees of $\mathcal{A}$ induced by the computations of s_i and t_i on any input sequence are depicted in Figure 3.6. Note that both $comp(s_i, \gamma)$ and $comp(t_i, \gamma)$ induce only a single run tree in $\mathcal{A}$. For an arbitrary input sequence $\gamma \in (2^{I_i})^\omega$, consider the delay-dominance game $\mathbb{G} = (\mathcal{A}, comp(t_i, \gamma), comp(s_i, \gamma))$.

In the following, we illustrate that Duplicator does not have a winning strategy in $\mathbb{G}$. We present a strategy μ for Spoiler in $\mathbb{G}$ and show afterward that it is winning, irrespective of Duplicator's moves. Strategy μ is a *memoryless* game strategy, which performs its choices only based on the current node of the game. It does not consider the history of the play. Thus, let $v \in P^*$ be some finite history of a play in $\mathbb{G}$. Due to the structure of $\mathcal{A}$ and the definition of t_i, no existential transitions occur in $\mathcal{A}$ when reading $comp(t_i, \gamma)$. Consequently, Spoiler's choices for positions of the form $((p, q), j) \in S_\exists$, are already fixed by the transition function δ of $\mathcal{A}$. More precisely, given position $((p, q), j) \in S_\exists$, the set $\delta(p, comp(s_i, \gamma)_j)$ is a singleton, i.e., we have $\delta(p, comp(t_i, \gamma)_j) = \{c\}$ for some set $c \subseteq Q$ of states of $\mathcal{A}$, and we define

$$\mu(v, ((p, q), j)) = ((p, q, c), j).$$

For positions of the form $((p, q, c, c'), j) \in S_\forall$, in contrast, Spoiler is, in some situations, required to make choices regarding the successor node $((p, q, c, q'), j)$ since, due to the structure



$(q_0, q_0), 0$
$(q_0, q_0, \{q_5\}), 0$
$(q_0, q_0, \{q_5\}, \{q_1\}), 0$
$(q_0, q_0, \{q_5\}, q_1), 0$
$(q_5, q_1), 1$
$(q_5, q_1, \{q_3, q_5\}), 1$
$(q_5, q_1, \{q_3, q_5\}, \{q_2\}), 1$
$(q_5, q_1, \{q_3, q_5\}, q_2), 1$
$(q_5, q_2), 2$
$(q_5, q_2, \{q_3, q_5\}), 2$
$(q_3, q_2), 2$
$(q_3, q_2, \{q_4\}), 2$
$(q_3, q_2, \{q_4\}, \{q_2, q_3\}), 2$
$(q_5, q_2, \{q_3, q_5\}, \{q_2, q_3\}), 2$
$(q_5, q_2, \{q_3, q_5\}, q_3), 2$
$(q_3, q_2, \{q_4\}, q_3), 2$
$(q_3, q_2, \{q_4\}, q_2), 2$
$(q_5, q_2, \{q_3, q_5\}, q_2), 2$
$(q_3, q_3), 3$
$(q_5, q_3), 3$
$(q_4, q_3), 3$
$(q_4, q_2), 3$
$(q_3, q_2), 3$
$(q_5, q_2), 3$

Figure 3.7.: Partial game arena of the delay-dominance game $\mathbb{G} = (\mathcal{A}, comp(t_i, \gamma), comp(s_i, \gamma))$ from Example 3.8. Positions controlled by Spoiler are depicted as rectangles, positions with rounded edges are controlled by Duplicator. Parts of the game arena that are not consistent with the winning moves of Spoiler are grayed out. Positions of the form $((p, q), j)$ that are critical for Duplicator are highlighted in blue.

of $\mathcal{A}$ and the definition of s_i, universal transitions can occur in $\mathcal{A}$ when reading $comp(s_i, \gamma)$. However, note that only a single universal transition can occur, namely the one from q_2 to both q_2 and q_3. Thus, whenever c' is not a singleton, then we have $c' = \{q_2, q_3\}$. Given position $((p, q, c, c'), j) \in S_\forall$, we then define

$$\mu(v, ((p, q, c, c'), j) = \begin{cases} ((p, q, c, q'), j+1) & \text{if } c' = \{q'\}, \\ ((p, q, c, q_3), j+1) & \text{if } q_3 \in c' \ \wedge \ p = q_3 \\ ((p, q, c, q_2), j+1) & \text{if } q_2 \in c' \ \wedge \ p \neq q_5, \end{cases}$$

The relevant part of the game arena of the delay-dominance game $\mathbb{G}$ is depicted in Figure 3.7. Parts that are not consistent with Spoiler's strategy μ are grayed out. Starting from the initial position $((q_0, q_0), 0)$, neither Duplicator not Spoiler has any choice in the first round of $\mathbb{G}$, resulting in the successor position $((q_1, q_5), 1)$. In the next round, the only choice is by Duplicator and allows for deciding whether the alternative states stay in q_5 or move to q_3.

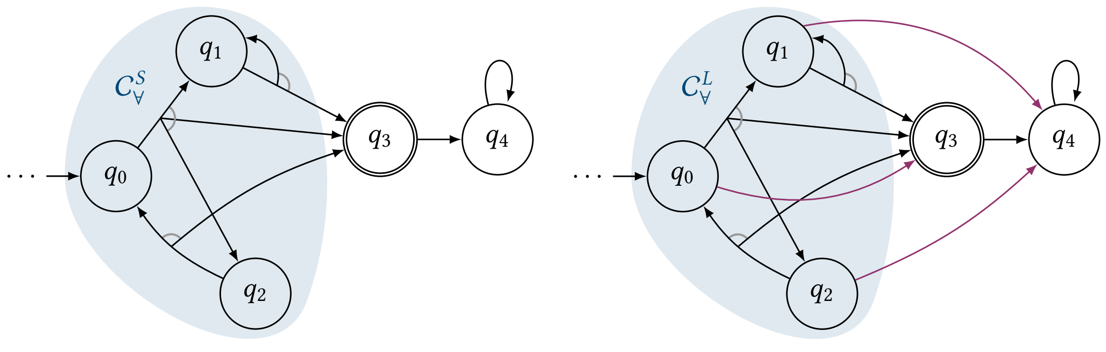
First, suppose that Duplicator chooses to let the alternative states move to q_3. Then, we obtain the successor node $((q_3, q_2), 2)$. In the next round, no existential transitions are possible due to the definition of strategy s_i. Hence, the first decision of the round is by Spoiler, and it can decide whether the dominant states stay in q_2 or move to q_3. Since the alternative states are in q_3, Spoiler chooses q_3 for the dominant states according to its strategy μ. Since there is only a single outgoing transition from q_3, Duplicator does not have any choice other than letting the alternative states move from q_3 to q_4, resulting in the successor node $((q_4, q_3), 3)$. Due to the structure of $\mathcal{A}$, state q_4 is a sink, i.e., it cannot be left again. Hence, the alternative states will always stay in q_4 from this round on. Furthermore, similar to the previous round for the alternative states, Spoiler does not have any choice other than letting the dominant states move from q_3 to q_4. Hence, we obtain the successor node $((q_4, q_4), 4)$ and, in fact, all subsequent positions in an initial play that is consistent with the choices of Duplicator and Spoiler described above are of the from $((q_4, q_4), j)$. Hence, every such initial consistent play in $\mathbb{G}$ contains exactly two visits to rejecting dominant states – namely q_1 in round 1 and q_3 in round 3 – and one visit to a rejecting alternative state – namely q_3 in round 2. The visit to a rejecting dominant state in round 3 is thus not matched with a visit to a rejecting alternative state. Consequently, such an initial consistent play does not satisfy the winning condition of the delay-dominance game. Hence, Duplicator's choice to let the alternative states move from q_5 to q_3 in round 2 results in losing the delay-dominance game $\mathbb{G}$.

Next, suppose that Duplicator chooses to let the alternative states stay in q_5 in round 2, yielding the successor node $((q_5, q_2), 2)$. Due to the definition of s_i and the structure of $\mathcal{A}$, no existential choices occur. Hence, the only choices Duplicator and Spoiler can make in the following concern the universal transitions in the alternative and dominant states, respectively. In each round, Spoiler moves first. According to Spoiler's strategy μ, it chooses to let the dominant states stay in q_2 as long as the alternative states are still in q_5. As soon as the alternative states move to q_3, Spoiler chooses to let the alternative states move to q_3 as well. If Duplicator never chooses to let the alternative states move to q_3, this results in a play in which the alternative states always stay in q_5, while the dominant states always stay in q_2. Then, no rejecting alternative state is visited. In contrast, a single rejecting dominant state is visited – namely q_1 in round 1. This visit is clearly not matched with a visit to a rejecting alternative state. Thus the play does not satisfy the winning condition of the delay-dominance game, resulting in Duplicator losing the game $\mathbb{G}$. If Duplicator chooses to let the alternative states move from q_5 to q_3 in some round $j \geq 3$, then we obtain the ending tuple $((q_3, q_2), j)$ for this round. In the next round, Spoiler can then decide to let the dominant states move from q_2 to q_3, while Duplicator does not have any choice other than letting the alternative states move from q_3 to q_4 due to the structure of $\mathcal{A}$. This results in the successor node $((q_4, q_3), j+1)$ and, similar to the very first case, in nodes of the form $((q_4, q_4), j')$ for all rounds $j' > j+1$. Hence, an initial consistent play again contains exactly two visits to rejecting dominant states – namely q_1 in round 1 and q_3 in round j – and one visit to a rejecting alternative state – namely q_3 in round $j+1$. The visit to a rejecting dominant state in round j is thus not matched with a visit to a rejecting alternative state. Consequently, such an initial consistent play does not satisfy the winning condition of the delay-dominance game. Duplicator thus also loses the game $\mathbb{G}$ for this choice. Consequently, s_i does not delay-dominate t_i and therefore s_i is not delay-dominant.

However, there does not exist a bad prefix of delay-dominance for s_i: let $k \geq 2$ be some point in time and let η be the prefix of $comp(s_i, \gamma)$ up to point in time k, i.e., let $\eta = comp(s_i, \gamma)_{|k+1}$. Let $\sigma \in (2^V)^\omega$ be an infinite extension of η with $a \in \sigma_{k'}$ for some point in time $k' \geq k$ and consider the delay-dominance game $\mathbb{G}' = (\mathcal{A}, comp(t_i, \gamma), \sigma)$. We construct a winning strategy μ' for Duplicator in $\mathbb{G}'$ as follows: for the existential choice in q_2 in round k', i.e., in the round corresponding to the point in time at which a occurs, μ' chooses to let the dominant states move to q_4. For the universal choice in q_5, it chooses to let the alternative states stay in q_5 up to round $k' - 1$ and to let them move to q_3 afterward, i.e., in round k'. Then, μ' ensures the visit to a rejecting alternative state *after* round k', namely in round $k' + 1$. However, we show in the following that the last rejecting dominant state occurs *before* round k'. For an initial play $\rho \in \mathit{Plays}(\mathbb{G}', \mu')$ that is consistent with μ' and in which the dominant states are in q_2 in round k', Duplicator's strategy μ' ensures that no rejecting dominant state is visited after round k'. In fact, no rejecting dominant state is visited after round 1, in which the dominant states visited q_1, since, by construction of $\mathcal{A}$, state q_2 is non-rejecting, it is reached in round 2, and staying in q_2 is the only possibility to be in q_2 in round k'. Since $k \geq 2$ and $k' \geq k$ holds by construction, we clearly have $1 < k'$ and thus the last rejecting dominant state occurs before round k'. For an initial play $\rho \in \mathit{Plays}(\mathbb{G}', \mu')$ that is consistent with μ' and in which the dominant states move from q_2 to q_3 in some round $k'' < k'$, it follows from the construction of $\mathcal{A}$ that the last rejecting dominant state is visited in round $k'' + 1$. Hence, since $k'' < k'$ holds by construction of k'', the last rejecting dominant state occurs before round k'. Thus, in every initial play $\rho \in \mathit{Plays}(\mathbb{G}', \mu')$ that is consistent with μ', every visit to a rejecting dominant state is matched with a visit to a rejecting dominant state and thus $\rho \in \mathbb{W}$ holds. Therefore, μ' is indeed a winning strategy for Duplicator in the delay-dominance game $\mathbb{G}'$. Since we chose $k \geq 2$ arbitrarily, there thus does not exist a bad prefix for delay-dominance in $\mathcal{A}$. $\triangle$

Hence, there indeed exist alternating co-Büchi automata that do not ensure bad prefixes. We identified the universal cycle structures of the automaton $\mathcal{A}$ depicted in Figure 3.5 to be critical in general for the existence of bad prefixes. Let s_i be a strategy for process p_i that is not delay-dominant. Let t_i be the alternative strategy and let $\gamma \in (2^{I_i})^\omega$ be the input sequence such that $s \not\trianglelefteq_{\mathcal{A},\gamma} t$ holds, i.e., such that Duplicator loses the game $\mathbb{G} = (\mathcal{A}, \mathit{comp}(t, \gamma), \mathit{comp}(s, \gamma))$. First of all, note that if postponing the point in time at which a *losing* rejecting dominant state, i.e., a rejecting dominant state that is never matched with a rejecting alternative state, is visited is not possible, then there exists a bad prefix for delay-dominance: in particular, there then exists a point in time $k_{max} \geq 0$ such that for all strategies μ of Duplicator in $\mathbb{G}$, there exists an initial play $\rho \in \mathit{Plays}(\mathbb{G}, \mu)$ that is consistent with μ and a point in time k' with $0 \leq k' \leq k_{max}$ such that $f_{alt}(\rho_{k'}) \in F$ holds, while we have $f_{dom}(\rho_{k''}) \notin F$ for all $k'' \geq k'$. Then, the prefix of length k_{max} of $\mathit{comp}(s_i, \gamma)$ is clearly a bad prefix for delay-dominance.

If, in contrast, Duplicator can postpone the point in time at which a *losing* visit to rejecting dominant state is visited, then either (i) Duplicator can delay making visits to rejecting dominant states losing indefinitely, or (ii) there is a point in time k_{alt} from which on all visits to rejecting dominant states are losing but Duplicator can delay visits to rejecting dominant states indefinitely. If the latter is the case, then Duplicator would not lose the game $\mathbb{G}$. Since the automaton $\mathcal{A}$ has a finite number of states, postponing the visit to losing rejecting dominant states indefinitely requires a cycle in $\mathcal{A}$ that allows Duplicator to choose at an arbitrary point in time to let a play visit a losing rejecting states after point in time k_{alt}. However, then Duplicator is also able to enforce that a rejecting dominant state is *never* visited after k_{alt}, resulting in a winning strategy for Duplicator in $\mathbb{G}$. Hence, (ii) cannot hold, and therefore Duplicator can delay making visits to rejecting states losing indefinitely. Since visits to rejecting dominant states are losing if they are not answered with a visit to a rejecting alternative state eventually, it thus



(a) Universal cycle structure $C^S_\forall$. (b) Universal cycle structure $C^L_\forall$.

Figure 3.8.: Schematic illustration of both types of universal cycle structures, $C^S_\exists$ and $C^L_\forall$. The sets of states corresponding to the structures are highlighted in blue. The transition labels are omitted for readability. The labels of violet transitions differ from those of black transitions.

follows that Duplicator can delay the point in time at which no rejecting alternative states are visited anymore indefinitely. If Duplicator could enforce infinitely many rejecting alternative states, then Duplicator would have a winning strategy in $\mathbb{G}$. Hence, Duplicator can enforce the visit to indefinitely many non-rejecting alternative states before the last rejecting alternative state. Since $\mathcal{A}$ has only a finite number of states, this thus requires a cycle structure $C^S_\forall$ in $\mathcal{A}$ which is entered when reading $comp(t_i, \gamma)$ such that Duplicator can choose to either enforce all plays to stay in the cycle structure in the alternative states or to leave it. Furthermore, all states in the cycle structure are non-rejecting, while a finite number of rejecting states, but at least one, is visited after leaving the cycle structure $C^S_\forall$. Since Duplicator controls the universal transitions for the alternative strategy, the cycle structure is thus universal in the sense that the transitions to either stay in the cycle structure or to leave it are universal. An illustration of such a cycle structure is depicted in Figure 3.8a. In the automaton from Example 3.8 depicted in Figure 3.5, state q_5 represents such a cycle structure.

Furthermore, if there would be a point in time $k \geq 0$ such that Duplicator can enforce that no rejecting dominant state is visited after this point in time k, then Duplicator would clearly win the game $\mathbb{G}$ since it could choose to let the alternative states stay in the cycle structure up to point in time k and to let them then leave the cycle structure, resulting in at least one visit to a rejecting alternative state after point in time k. Thus, Spoiler is able to delay the last visit to a rejecting dominant state indefinitely as well. Therefore, there exists an analogous universal cycle structure $C_\forall$ in the automaton, which is entered when reading $comp(s_i, \gamma)$. Furthermore, there exists a *pending* visit to a rejecting dominant state, i.e., a visit to a rejecting dominant state that has not been matched with a visit to a rejecting alternative state so far. Otherwise, it would be winning for Duplicator to enforce that the alternative states stay in the cycle structure $C^S_\forall$, which is entered when reading $comp(t_i, \gamma)$, forever. In the automaton from Example 3.8 depicted in Figure 3.5, state q_2 represents such a cycle structure. As long as the cycle structure $C_\forall$ that is reached when reading $comp(s_i, \gamma)$ cannot be left irrespective of the moves of Spoiler for some infinite extension of a finite prefix of $comp(s_i, \gamma)$, i.e., as long as $C_\forall$ is of the form as the one depicted in Figure 3.8a, the bad prefix property for delay-dominance is still satisfied. Analogous to the explicit case in Example 3.8, we can show that it is neither winning for Duplicator to choose to let the alternative states stay in the cycle structure $C_\forall$ forever nor to let them leave the cycle structure eventually. If, however, analogous to the transition with a from q_2 to q_5 in the automaton from Example 3.8 depicted in Figure 3.5, there exists the possibility that Duplicator enforces to leave the cycle structure and to visit only a finite number of rejecting states afterward, the bad prefix property for delay-dominance is not guaranteed. Such a cycle structure $C^L_\forall$ is schematically depicted in Figure 3.8b. The violet transitions denote the possibility for Duplicator to enforce to leave the cycle structure as they are labeled differently than the black transitions. Similar to the transition with a from q_2 to q_5 in Example 3.8, they thus allow for infinite extensions of every finite prefix of $comp(s_i, \gamma)$ for which Duplicator has winning strategy in the respective delay-dominance game.

Therefore, the combination of cycle structures as depicted in Figure 3.8 are critical for the bad prefix property for delay-dominance. However, in many cases, such structures do not occur in alternating co-Büchi automata that are constructed from an LTL formula with standard algorithms. For instance, the automaton $\mathcal{A}$ from Figure 3.5 does not feature rejecting states that lie in cycles. Thus, in particular, no branch of a run tree of $\mathcal{A}$ can contain infinitely many visits to rejecting states, and consequently, $\mathcal{A}$ accepts *every* infinite word over $2^{\{a,b\}}$. More precisely, $\mathcal{A}$ describes the LTL formula *true*. However, standard algorithms would never yield such a peculiar automaton as $\mathcal{A}$ when constructing an alternating co-Büchi automaton for the formula *true*. In particular, for every *safety* specification, there exists an alternating co-Büchi automaton that ensures bad prefixes for delay-dominance:

Lemma 3.4. *Let φ be an LTL formula over atomic propositions V. If φ is a safety property, then there exists an alternating co-Büchi automaton $\mathcal{A}_\varphi$ with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$ that ensures bad prefixes for delay-dominance.*

Proof. Suppose that φ is a safety property. Then, there exists an alternating co-Büchi automaton $\mathcal{A}_\varphi = (Q, q_0, \delta, F)$ with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$ and a single rejecting sink, i.e., with a single state $q \in Q$ such that $F = \{q\}$ and such that $(q, \iota, q) \in \delta$ holds for all $\iota \in 2^V$, while we have $(q, \iota, q') \notin \delta$ for all $\iota \in 2^V$ and all $q' \in Q$ with $q \neq q'$. Hence, a branch of a run tree of $\mathcal{A}_\varphi$ induced by an infinite sequence $\sigma \in 2^V$ either visits no rejecting state at all or it visits infinitely many rejecting states. Let $\sigma \in 2^V$ be some infinite sequence such that Duplicator loses the delay-dominance game $\mathbb{G} = (\mathcal{A}_\varphi, \sigma', \sigma)$ for some infinite sequence $\sigma' \in (2^V)^\omega$. Then, it follows from the structure of $\mathcal{A}_\varphi$ that for all strategies μ of Duplicator, there exists some initial consistent play $\rho \in Plays(\mathbb{G}, \mu)$ such that $f_{dom}(\rho_k) = q$ holds for some point in time $k \geq 0$, while we have $f_{alt}(\rho_{k'}) \neq q$ for all $k' \geq 0$. Hence, ρ visits q in its dominant states, while it never reaches q in its rejecting states. Since q is a rejecting sink, ρ thus visits infinitely many rejecting dominant states, while it does not visit any rejecting alternative states. If there is no point in time $k \geq 0$ such that for all strategies μ of Duplicator, there exists some initial consistent play $\rho \in Plays(\mathbb{G}, \mu)$ such that $f_{dom}(\rho_k) = q$ holds, then Duplicator can delay visiting q in the dominant states indefinitely. Yet, since $\mathcal{A}_\varphi$ has a finite number of states, it thus follows that Duplicator can also enforce that q is never entered; contradicting that Duplicator loses the game $\mathbb{G}$. Hence, there is a point in time $k \geq 0$ such that for all strategies μ of Duplicator, there exists some initial consistent play $\rho \in \mathit{Plays}(\mathbb{G}, \mu)$ such that $f_{dom}(\rho_k) = q$ holds. Let $\eta \in (2^V)^*$ be the prefix of σ up to point in time k, i.e., let $\eta = \sigma_{|k+1}$. Since q is a rejecting sink by construction of $\mathcal{A}_\varphi$, it then follows immediately that for all infinite extensions $\sigma'' \in (2^V)^\omega$ of η and all strategies μ of Duplicator in the delay-dominance game $\mathbb{G}' = (\mathcal{A}_\varphi, \sigma', \sigma'')$, there exists some initial consistent play $\rho \in \mathit{Plays}(\mathbb{G}', \mu)$ that visits infinitely many rejecting states. Hence, Duplicator loses the delay-dominance game $\mathbb{G}'$ and therefore, since we chose the infinite extension σ'' of η arbitrarily, $\mathcal{A}_\varphi$ ensures bad prefixes for delay-dominance. □

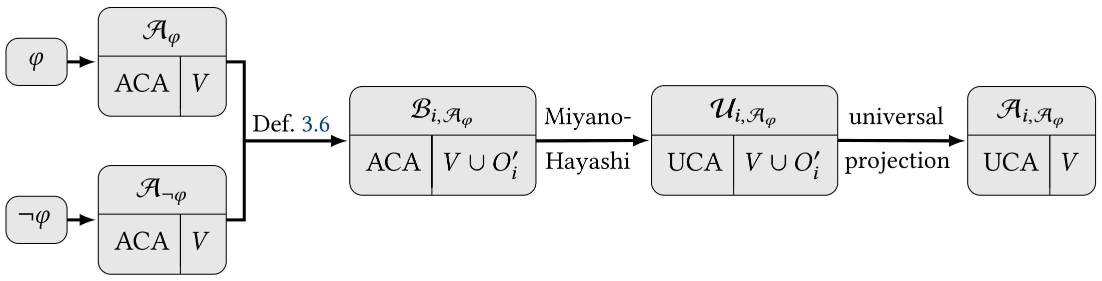
Furthermore, for many liveness specifications φ, there exists an alternating co-Büchi automaton $\mathcal{A}_\varphi$ with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$ that ensures bad prefixes for delay-dominance as well since, in general, the existence of critical automaton structures is quite rare. Nevertheless, we cannot exclude that there exist liveness specification that enforce such critical cycle structures and thus enforce an alternating co-Büchi automaton that does not ensure bad prefixes. We, however, have not encountered such a specification so far. Hence, delay-dominance is a suitable notion of best effort for compositional synthesis: it is compositional for many properties and the bad prefix criterion even allows us to determine whether or not the parallel composition of two delay-dominant strategies will be delay-dominant *before* synthesizing and then composing them by analyzing the specification automaton. Therefore, we introduce an automaton construction for synthesizing delay-dominant strategies for individual processes in the following section, which can then be utilized for compositional distributed synthesis.

3.4. Synthesizing Delay-Dominant Strategies

In this section, we introduce how delay-dominant strategies can be synthesized using existing tools for synthesizing winning strategies. We focus on utilizing *bounded synthesis* [FS13] tools such as BoSy [FFT17]. Mostly, we use bounded synthesis (see Section 2.8.1) as a black box procedure throughout this section. A crucial observation regarding bounded synthesis that we utilize, however, is that it translates the given specification φ into an equivalent universal co-Büchi automaton $\mathcal{A}_\varphi$, i.e., a universal co-Büchi automaton with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$ and then derives a strategy such that, for every input sequence, the runs of $\mathcal{A}_\varphi$ induced by the computation of the strategy on the input sequence visit only finitely many rejecting states.

To synthesize delay-dominant strategies instead of winning ones, we can thus use existing bounded synthesis algorithms by replacing the universal co-Büchi automaton $\mathcal{A}_\varphi$ that represents the specification φ with a universal co-Büchi automaton encoding delay-dominance, i.e., with an automaton $\mathcal{A}_{i,\mathcal{A}_\varphi}$ such that its runs induced by the computations of a delay-dominant strategy on all input sequences visit only finitely many rejecting states. This idea is similar to the approach for synthesizing remorsefree dominant strategies (see Section 2.8.2). The automaton for recognizing delay-dominant strategies, however, differs inherently from the one for recognizing remorsefree dominant strategies.

The automaton construction consists of several steps. An overview is given in Figure 3.9. Since delay-dominance is not defined on the LTL specification φ itself but on an equivalent automaton, we first translate φ into an alternating co-Büchi automaton $\mathcal{A}_\varphi$ with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$.



φ
$\neg\varphi$
$\mathcal{A}_\varphi$
ACA
V
$\mathcal{A}_{\neg\varphi}$
ACA
V
Def. 3.6
$\mathcal{B}_{i,\mathcal{A}_\varphi}$
ACA
$V \cup O'_i$
Miyano-
Hayashi
$\mathcal{U}_{i,\mathcal{A}_\varphi}$
UCA
$V \cup O'_i$
universal
projection
$\mathcal{A}_{i,\mathcal{A}_\varphi}$
UCA
V

Figure 3.9.: Overview of the construction of a universal co-Büchi automaton $\mathcal{A}_{i,\mathcal{A}_\varphi}$ recognizing delay-dominant strategies for the alternating co-Büchi automaton $\mathcal{A}_\varphi$ with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$. The lower parts of the boxes list the automaton type, i.e. alternating or universal, and the alphabet, i.e., with or without primed variables.

For this, we utilize well-known algorithms for translating LTL formulas into equivalent alternating Büchi automata as well as the duality of the Büchi and co-Büchi acceptance condition and of nondeterministic and universal branching (see Section 2.5.2). Similarly, we construct an alternating co-Büchi automaton $\mathcal{A}_{\neg\varphi}$ with $\mathcal{L}(\mathcal{A}_{\neg\varphi}) = \mathcal{L}(\neg\varphi)$ from $\neg\varphi$. The centerpiece of the automaton construction is an alternating co-Büchi automaton $\mathcal{B}_{i,\mathcal{A}_\varphi}$ constructed from $\mathcal{A}_\varphi$ and $\mathcal{A}_{\neg\varphi}$ that recognizes whether a strategy s_i for process p_i delay-dominates some alternative strategy t_i for p_i for $\mathcal{A}_\varphi$. Then, $\mathcal{B}_{i,\mathcal{A}_\varphi}$ is translated into an equivalent universal co-Büchi automaton $\mathcal{U}_{i,\mathcal{A}_\varphi}$ with $\mathcal{L}(\mathcal{U}_{i,\mathcal{A}_\varphi}) = \mathcal{B}_{i,\mathcal{A}_\varphi}$, for example with the Miyano-Hayashi algorithm [MH84]. Lastly, we translate $\mathcal{U}_{i,\mathcal{A}_\varphi}$ into a universal co-Büchi automaton that accounts for requiring a strategy s_i to delay-dominate *all* other strategies t_i for p_i and not only a particular one utilizing universal projection. In the remainder of this section, we describe all steps of the construction in detail and prove their correctness.

3.4.1. Construction of the Basic ACA for Delay-Dominance

From the two alternating co-Büchi automata $\mathcal{A}_\varphi$ and $\mathcal{A}_{\neg\varphi}$, we construct an alternating co-Büchi automaton $\mathcal{B}_{i,\mathcal{A}_\varphi}$ that recognizes whether Duplicator has a winning strategy in the delay-dominance game $(\mathcal{A}_\varphi, \sigma, \sigma')$ and thus, in particular, whether strategy s_i for process p_i delay-dominates an alternative strategy t_i for p_i on some input sequence $\gamma \in (2^{I_i})^\omega$ for $\mathcal{A}_\varphi$. The construction relies on the observation that Duplicator has a winning strategy in the delay-dominance game $\mathbb{G} = (\mathcal{A}_\varphi, \sigma, \sigma')$ if, and only if, either (i) σ violates φ or (ii) there exists a winning strategy μ in the game $\mathbb{G}$ such that every initial play of $\mathbb{G}$ that is consistent with μ visits only finitely many rejecting dominant state.

Lemma 3.5. *Let φ be an LTL formula over atomic propositions V. Let $\mathcal{A}_\varphi = (Q, q_0, \delta, F)$ be an alternating co-Büchi automaton with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$. Let $\sigma, \sigma' \in (2^V)^\omega$ be sequences. Then, Duplicator has a winning strategy in the delay-dominance game $\mathbb{G} = (\mathcal{A}_\varphi, \sigma, \sigma')$ if, and only if, either (i) we have $\sigma \not\models \varphi$, or (ii) there exists a winning strategy μ in the game $\mathbb{G}$ such that for every initial play $\rho \in Plays(\mathbb{G}, \mu)$ that is consistent with μ, there is a point in time $k \geq 0$ such that $f_{dom}(\rho_{k'}) \notin F$ holds for all $k' \geq k$.*

Proof. First, assume that Duplicator has a winning strategy μ for in the delay-dominance game $\mathbb{G} = (\mathcal{A}_\varphi, \sigma, \sigma')$. If every initial play $\rho \in \mathit{Plays}(\mathbb{G}, \mu)$ that is consistent with μ contains only finitely many visits to rejecting dominant states, then (ii) holds and thus the claim follows. Otherwise, we have infinitely many visits to rejecting dominant states for some initial play $\rho \in \mathit{Plays}(\mathbb{G}, \mu)$ that is consistent with μ. Let μ' be some strategy for Spoiler such that ρ is consistent with both μ and μ'. Note that, by construction of the delay-dominance game, only the part of μ' that defines the universal choices in $\mathcal{A}_\varphi$ that occur when reading σ' affects whether or not ρ contains infinitely many visits to rejecting dominant states. Let μ'' be a strategy for only these choices that coincides with the ones defined by μ'. Then, for all full strategies $\hat{\mu}$ for Spoiler that coincide with μ'' on the universal choices in $\mathcal{A}_\varphi$ for σ', the initial play ρ' that is consistent with both μ and $\hat{\mu}$ contains infinitely many visits to rejecting dominant states. Since μ is a winning strategy for Duplicator by assumption and by construction of the delay-dominance game, it follows that every such initial play ρ' contains infinitely many visits to rejecting alternative states. Thus, intuitively, independent of the existential choices in $\mathcal{A}_\varphi$ for σ, strategy μ can enforce infinitely many rejecting alternative states.

By Lemma 3.2, for all such full strategies $\hat{\mu}$ for Spoiler, there exists a run tree r of $\mathcal{A}_\varphi$ induced by σ that reflects the existential choices of $\mathcal{A}_\varphi$ for σ defined by $\hat{\mu}$. Moreover, we have $\mathit{Branches}_{\mathit{Inf}}(r) = \left\{\hat{\rho}^{alt} \mid \rho \in \mathit{Plays}(\mathbb{G}, \hat{\mu})\right\}$. Thus, by definition of the projected alternative play, we obtain that for all strategies $\hat{\mu}$ for Spoiler extending μ'', the initial play ρ' that is consistent with both μ and $\hat{\mu}$ is a branch of r. Since ρ' contains infinitely many rejecting alternative states, it follows that all such run trees r contain a branch with infinitely many visits to rejecting states. Moreover, since μ'' does not fix any decision regarding the choices occurring in $\mathcal{A}_\varphi$ when reading σ, indeed *every* run tree of $\mathcal{A}_\varphi$ induced by σ contains a branch with infinitely many visits to rejecting states. Therefore, by definition of alternating co-Büchi automata, $\mathcal{A}_\varphi$ rejects σ. Since $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$ holds by assumption, $\sigma \not\models \varphi$ follows. Hence, (i) holds.

Second, let (i) or (ii) hold. If (ii) holds, then it follows immediately that Duplicator has a winning strategy in the delay-dominance game $\mathbb{G}$. Thus, let (i) hold, i.e., we have $\sigma \not\models \varphi$. Then, since $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$ holds by assumption, $\mathcal{A}_\varphi$ rejects σ and hence for all run trees of $\mathcal{A}_\varphi$ induced by σ, there exists a branch that visits infinitely many rejecting states. Let $r \in \mathit{Runs}(\mathcal{A}_\varphi, \sigma)$ be some run tree of $\mathcal{A}_\varphi$ induced by σ. By Lemma 3.3, there exists a strategy μ for Spoiler in the delay-dominance game $\mathbb{G}$ that reflects the existential choices in $\mathcal{A}_\varphi$ when reading σ defined by μ. Moreover, we have $\mathit{Branches}_{\mathit{Inf}}(r) = \left\{\hat{\rho}^{alt} \mid \rho \in \mathit{Plays}(\mathbb{G}, \mu)\right\}$. Note that only the part of μ controlling the existential choices of $\mathcal{A}_\varphi$ when reading σ is relevant for this property. Thus, in fact, there are strategies μ^r for all run trees r of $\mathcal{A}_\varphi$ induced by σ that coincide for the other part of a strategy for Spoiler, i.e., the universal choices of $\mathcal{A}_\varphi$ when reading σ'. Let $\mathcal{M}$ be the set of such strategies μ^r of all run trees $r \in \mathit{Runs}(\mathcal{A}_\varphi, \sigma)$. As shown above, the sequences of alternative states in consistent plays of such strategies $\mu^r \in \mathcal{M}$ coincide with branches of r. Thus, since every run r contains a branch b that visits infinitely many rejecting states, there also exists an initial play $\rho^r \in \mathit{Plays}(\mathbb{G}, \mu^r)$ of the delay-dominance game $\mathbb{G}$ that is consistent with μ^r and which contains infinitely many rejecting alternative states. Note that since the number of rejecting alternative states is only affected by the alternative states of the play and since all strategies μ^r coincide on the universal choices of $\mathcal{A}_\varphi$ when reading σ', there are, in particular, such plays ρ^r that all coincide in the dominant states. Moreover, there is a set $\mathcal{P}$ of such plays such that for every two plays $\rho, \rho' \in \mathcal{P}$ that coincide in the alternative states up to point in time $k \geq 0$ as well as in the previous decisions for the alternative states in the current round $k+1$ of the game $\mathbb{G}$, the plays ρ and ρ' coincide on the universal decision for the alternative state in round $k+1$ as well: suppose that this is not the case. Then, there is a finite prefix $v \in P^{\omega}$ of a play that coincides with ρ and ρ' up to point in time $|v|-1$ and that requires a universal choice between options u and u' in the next step. Moreover, suppose that u is the correct extension of v for a play ρ, while u' is the correct one for a play ρ', i.e., the respective other choice does not yield a play with infinitely many rejecting alternative states. But then, there is also the run tree r that, depending on the universal choice u vs. u' makes the existential choices that causes a play with only finitely many rejecting alternative states, i.e., ρ for u' and ρ' for u. Since this is the case for all such situations and since there are run trees for all possible combinations of existential choices, there thus exists a run tree whose branches all visit only finitely many rejecting states; contradicting the assumption. Hence, there indeed exists such a set $\mathcal{P}$ of plays of the delay-dominance game $\mathbb{G}$ such that (i) all plays $\rho \in \mathcal{P}$ contain infinitely many rejecting alternative states, (ii) all plays $\rho \in \mathcal{P}$ coincide on the dominant states, and (iii) where for every two plays $\rho, \rho' \in \mathcal{P}$ that coincide in the alternative states up to point in time $k \geq 0$ as well as in the previous decisions for the alternative states in the current round $k+1$ of the game, ρ and ρ' coincide on the universal decision for the alternative state in round $k+1$ as well. Thus, in particular, for every finite prefix of a play in $\mathcal{P}$, the next universal decision of $\mathcal{A}_\varphi$ when reading σ can always be made solely based on the information about the history. Hence, we construct a *strategy* μ' for the universal choices occurring in $\mathcal{A}_\varphi$ when reading σ from $\mathcal{P}$ by defining the respective choice defined by the plays in $\mathcal{P}$ for every finite prefix. But then, since all plays in $\mathcal{P}$ contain infinitely many rejecting alternative states, every initial play that is consistent with μ' does so as well. Since the existential choices occurring in $\mathcal{A}_\varphi$ when reading σ' do not influence the alternative states of a play, it follows that for all strategies μ of Duplicator that coincides with μ' on the universal choices occurring in $\mathcal{A}_\varphi$ when reading σ, all consistent initial plays contain infinitely many rejecting alternative states. Thus, all such strategies μ are winning strategies for Duplicator in the game $\mathbb{G}$ and therefore Duplicator wins the delay-dominance game $\mathbb{G}$. □

Due to this observation, the alternating co-Büchi automaton $\mathcal{B}_{i,\mathcal{A}_\varphi}$ consists of two parts, one accounting for (i) and one accounting for (ii), and guesses nondeterministically in the initial state which part is entered when reading some sequence $\sigma \in (2^V)^\omega$. The alternating co-Büchi automaton $\mathcal{A}_{\neg\varphi}$ with $\mathcal{L}(\mathcal{A}_{\neg\varphi}) = \mathcal{L}(\neg\varphi)$ clearly accounts for (i). For (ii), we intuitively build the product of two copies of the alternating co-Büchi automaton $\mathcal{A}_\varphi$ with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$, one for each of the considered process strategies s_i and t_i. Note that similar to the change of control for t_i in the delay-dominance game, we consider the *dual* transition function of $\mathcal{A}_\varphi$, i.e., the one where conjunctions and disjunctions are swapped, for the copy of $\mathcal{A}_\varphi$ for t_i. We keep track of whether we encountered a situation in which a rejecting state was visited for s_i while it was not for t_i. This allows for defining the set of rejecting states of $\mathcal{B}_{i,\mathcal{A}_\varphi}$.

Note that we need to allow for differentiating valuations of output variables computed by strategies s_i and t_i on the same input sequence. Therefore, we extend the alphabet of $\mathcal{B}_{i,\mathcal{A}_\varphi}$. In addition to the set V of all variables of the system, we consider the set $O'_i := \{o' \mid o \in O_i\}$

of *primed output variables* of system process $p_i \in P^-$, where every output variable is marked with a prime to obtain a fresh symbol. The set V_i' of *primed variables* of p_i is then given by $V_i' := I_i \cup O_i'$. Intuitively, the output variables O_i depict the behavior of the possibly delay-dominant strategy s_i, while the primed output variables O_i' depict the behavior of the alternative strategy t_i. The alphabet of $\mathcal{B}_{i,\mathcal{A}_\varphi}$ is then given by $V \cup O_i'$. Note that this is equivalent to $V \cup V_i'$ since the input variables are never primed to ensure that we consider the same input sequence for both strategies. In the following, we use the functions $pr : V_i \to V_i'$ and $unpr : V_i' \to V_i$ to switch between primed variables and normal ones: given a valuation $a \in V_i$ of variables, $pr(a)$ replaces every output variable $o \in O_i$ occurring in a with its primed version o'. For a valuation $a \in V_i'$, $unpr(a)$ replaces every primed output variable $o' \in O_i'$ occurring in a with its regular unprimed version o. We extend pr and $unpr$ to finite and infinite sequences as usual. The alternating co-Büchi automaton $\mathcal{B}_{i,\mathcal{A}_\varphi}$ is then constructed as follows:

Definition 3.6.
Let φ be an LTL formula over atomic propositions V. Let $p_i \in P^-$ be some system process. Let $\mathcal{A}_\varphi = (Q, q_0, \delta, F)$ and $\mathcal{A}_{\neg\varphi} = (Q^c, q_0^c, \delta^c, F^c)$ be alternating co-Büchi automata with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$ and $\mathcal{L}(\mathcal{A}_{\neg\varphi}) = \mathcal{L}(\neg\varphi)$, respectively. For p_i, we construct the alternating co-Büchi automaton $\mathcal{B}_{i,\mathcal{A}_\varphi} = (Q^A, Q_0^A, \delta^A, F^A)$ with alphabet $V \cup O_i'$ as follows:

- $Q^A := (Q \times Q \times \{\top, \bot\}) \cup Q^c$
- $Q_0^A = (q_0, q_0, \top)$
- $F^A = (Q \times Q \times \{\bot\}) \cup F^c$
- $\delta^A : ((Q \times Q \times \{\top, \bot\}) \cup Q^c) \times 2^{V \cup O_i'} \to (Q \times Q \times \{\top, \bot\}) \cup Q^c$ with

$$\begin{aligned}
\delta^A(q_c, \tilde{\iota}) &= \delta^c(q_c, \iota') \quad \text{for } q_c \in Q^c \\
\delta^A((q_0, q_0, \top), \tilde{\iota}) &= \delta^c(q_0, \iota') \vee \bigwedge_{c \in \delta(q_0, \iota')} \; \bigvee_{c' \in \delta(q_0, \iota)} \; \bigwedge_{q' \in c'} \; \bigvee_{p' \in c} \vartheta(p', q', \top) \\
\delta^A((p, q, m), \tilde{\iota}) &= \bigwedge_{c \in \delta(p, \iota')} \; \bigvee_{c' \in \delta(q, \iota)} \; \bigwedge_{q' \in c'} \; \bigvee_{p' \in c} \vartheta(p', q', m)
\end{aligned}$$

where we have $\iota = \tilde{\iota} \cap V$ and $\iota' = unpr(\tilde{\iota} \cap V_i) \cup (\tilde{\iota} \cap (V \setminus V_i))$ as well as where we define $\vartheta : (Q \times Q \times \{\top, \bot\}) \to Q \times Q \times \{\top, \bot\}$ by

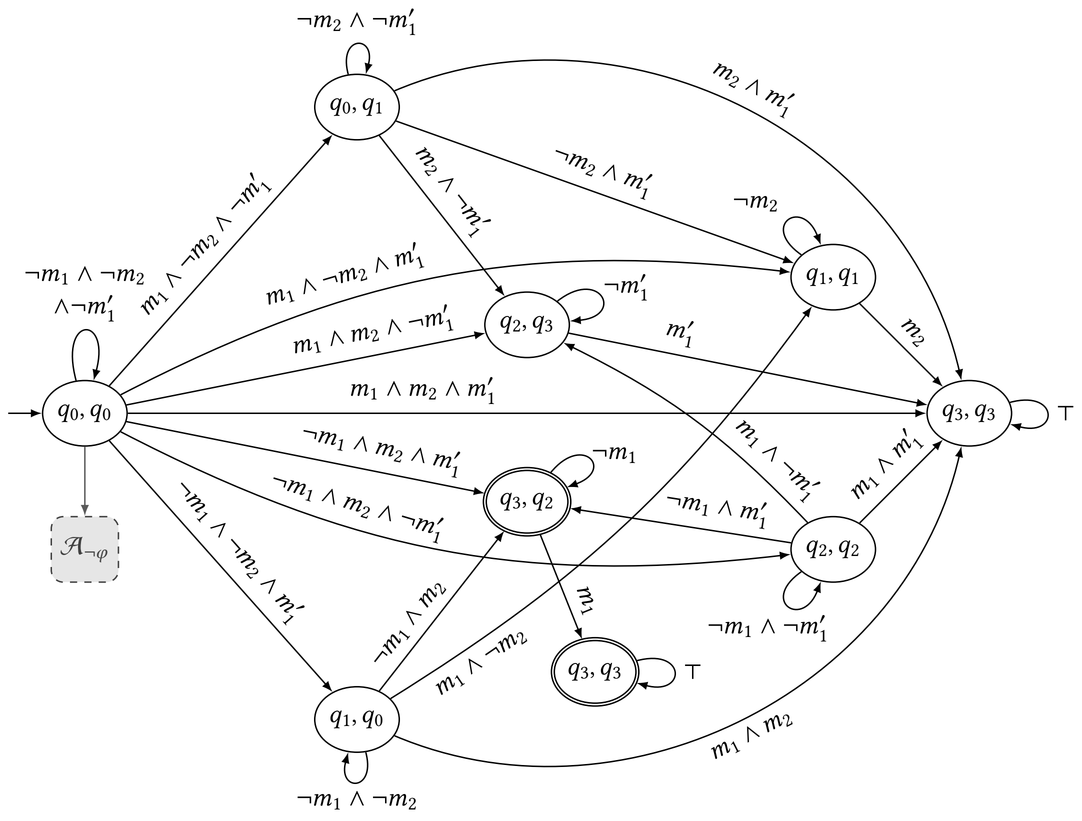
$$\vartheta(p, q, m) = \begin{cases} (p, q, \bot) & \text{if } p \notin F,\ q \in F, \text{ and } m = \top \\ (p, q, \bot) & \text{if } p \notin F \text{ and } m = \bot \\ (p, q, \top) & \text{otherwise} \end{cases}$$

Indeed, the alternating co-Büchi automaton $\mathcal{B}_{i,\mathcal{A}_\varphi}$ constructed as defined above consists of two parts: the one defined by states of the form (p, q, m), and the one defined by the states of $\mathcal{A}_{\neg\varphi}$. By definition of δ^A, these parts are only connected in the initial state of $\mathcal{B}_{i,\mathcal{A}_\varphi}$, where a nondeterministic transition to the respective successors in both parts ensures that choosing nondeterministically whether (i) or (ii) will be satisfied is possible, i.e., whether (i) the alternative strategy t_i violates φ on input sequence $\gamma \in (2^{I_i})$ or (ii) strategy s_i delay-dominates the alternative strategy t_i on input sequence γ and for every initial play that is consistent with Duplicator's winning strategy, there exists a point in time such that no rejecting dominant state is visited from this point in time on. For states of the form (p, q, m), the mark $m \in \{\top, \bot\}$ determines whether there are *pending visits to rejecting states* in the copy of $\mathcal{A}_\varphi$ for the possibly delay-dominant strategy, i.e., the second component q of (p, q, m). A pending visit to a rejecting state is one that is not yet matched with a visit to a rejecting state in the copy of $\mathcal{A}_\varphi$ for the alternative strategy. Therefore, the function ϑ defines that if a visit to a rejecting dominant state that is not immediately matched with a rejecting alternative state is encountered, the mark is set to $\bot$. As long as no rejecting alternative state is visited, the mark stays set to $\bot$. If a matching rejecting alternative state occurs, however, the mark is reset to $\top$, indicating that the pending visit to a rejecting visit has been matched and is thus not pending anymore. States of $\mathcal{B}_{i,\mathcal{A}_\varphi}$ marked with $\bot$ are then defined to be rejecting states, ensuring that a visit to a rejecting dominant state is not pending forever.

Including an alternating co-Büchi automaton $\mathcal{A}_{\neg\varphi}$ that recognizes sequences satisfying the negated specification into $\mathcal{B}_{i,\mathcal{A}_\varphi}$ is necessary to, intuitively, allow a sequence σ to induce visits to rejecting states that are not immediately matched with a rejecting state induced by an alternative sequence σ', but that are matched eventually, infinitely often. In such a case, both σ and σ' cause infinitely many visits to rejecting states in $\mathcal{A}_\varphi$ and, in particular, Duplicator wins the delay-dominance game $(\mathcal{A}_\varphi, \sigma', \sigma)$ due to the definition of the winning condition $\mathbb{W}$. Since the rejecting states induced by σ are not matched immediately, however, the automaton $\mathcal{B}_{i,\mathcal{A}_\varphi}$ contains rejecting states for the visits of rejecting states of $\mathcal{A}_\varphi$ caused by σ due to the definition of ϑ. Since σ causes infinitely many visits to rejecting states in $\mathcal{A}_\varphi$, the part of the automaton $\mathcal{B}_{i,\mathcal{A}_\varphi}$ that does not represent $\mathcal{A}_{\neg\varphi}$ thus also visits infinitely many rejecting states when confronted with σ and σ', resulting in rejection. To accurately capture delay-dominance, we thus add the part corresponding to $\mathcal{A}_{\neg\varphi}$, which recognizes such cases and enforces acceptance, to $\mathcal{B}_{i,\mathcal{A}_\varphi}$.

Example 3.9. Consider the message-sending system from the running example and the alternating co-Büchi automaton $\mathcal{A}_\varphi$ depicted in Figure 3.2, which describes the system specification. An alternating co-Büchi automaton $\mathcal{A}_{\neg\varphi}$ for the negated specification is similar to $\mathcal{A}_\varphi$, yet, rejecting and non-rejecting states are interchanged. That is, states q_0, q_1, and q_2 are non-rejecting in $\mathcal{A}_{\neg\varphi}$ while state q_3 is rejecting. The (partial) alternating co-Büchi automaton $\mathcal{B}_{1,\mathcal{A}_\varphi}$ for system process p_1 constructed from $\mathcal{A}_\varphi$ and $\mathcal{A}_{\neg\varphi}$ according to Definition 3.6 is depicted in Figure 3.10. The part of $\mathcal{B}_{1,\mathcal{A}_\varphi}$ that corresponds to $\mathcal{A}_{\neg\varphi}$ is omitted for the sake of readability. Since the mark $\top$ or $\bot$ of a state of the displayed part of $\mathcal{B}_{1,\mathcal{A}_\varphi}$ can be uniquely inferred from the fact that a state is non-rejecting or rejecting, respectively, we omit it from the state names.

Note that the displayed part of $\mathcal{B}_{1,\mathcal{A}_\varphi}$, i.e., the product automaton part, rejects a word $\sigma \in (2^{\{m_1,m_2,m_1'\}})^\omega$ if, and only if, it contains m_2 at some point in time $k \geq 0$ and m_1' at some (possibly different) point in time $k' \geq 0$, while it does not contain m_1 for all points in time $k'' \geq 0$ with $k'' \leq \max\{k, k'\}$. Furthermore, by construction, the part of $\mathcal{B}_{1,\mathcal{A}_\varphi}$ that corresponds to $\mathcal{A}_{\neg\varphi}$ rejects a word $\sigma \in (2^{\{m_1,m_2,m_1'\}})^\omega$ if, and only if, $\sigma \models \Diamond m_2 \land \Diamond m_1'$ holds. Thus, $\mathcal{A}_{\neg\varphi}$ is more restrictive than the displayed part of $\mathcal{B}_{1,\mathcal{A}_\varphi}$ in the sense that it rejects all words that are



q_0, q_0
q_0, q_1
q_1, q_0
q_1, q_1
q_2, q_3
q_3, q_3
q_3, q_2
q_2, q_2
q_3, q_3
$\mathcal{A}_{\neg\varphi}$
$\neg m_2 \wedge \neg m_1'$
$m_2 \wedge m_1'$
$\neg m_2 \wedge m_1'$
$m_2 \wedge \neg m_1'$
$\neg m_2$
$m_1 \wedge \neg m_2 \wedge \neg m_1'$
$\neg m_1 \wedge \neg m_2$
$\wedge \neg m_1'$
$m_1 \wedge \neg m_2 \wedge m_1'$
$\neg m_1'$
$m_1 \wedge m_2 \wedge \neg m_1'$
m_1'
m_2
$m_1 \wedge m_2 \wedge m_1'$
$\top$
$\neg m_1 \wedge m_2 \wedge m_1'$
$\neg m_1$
$m_1 \wedge \neg m_1'$
$\neg m_1 \wedge m_2 \wedge \neg m_1'$
$\neg m_1 \wedge m_1'$
$m_1 \wedge m_1'$
$\neg m_1 \wedge \neg m_2 \wedge m_1'$
$\neg m_1 \wedge m_2$
m_1
$\neg m_1 \wedge \neg m_1'$
$m_1 \wedge \neg m_2$
$\top$
$m_1 \wedge m_2$
$\neg m_1 \wedge \neg m_2$

Figure 3.10.: Alternating co-Büchi automaton $\mathcal{B}_{1,\mathcal{A}_\varphi}$ for the running example. The part of the automaton that corresponds to $\mathcal{A}_{\neg\varphi}$ is omitted for the sake of readability. Since the mark $\top$ or $\bot$ of a state of the displayed part of $\mathcal{B}_{1,\mathcal{A}_\varphi}$ can uniquely inferred from the fact that a state is non-rejecting or rejecting, respectively, we omit it from the state names.

rejected by the displayed part of $\mathcal{B}_{1,\mathcal{A}_\varphi}$ as well as some additional words, namely those that contain both m_2 and m_1' but also contain m_1 before or at the same time as both m_2 and m_1' have occurred. Since both parts of $\mathcal{B}_{1,\mathcal{A}_\varphi}$ are only connected in the initial state and since we use an existential transition for this connection, it thus follows that the overall automaton $\mathcal{B}_{1,\mathcal{A}_\varphi}$ rejects a word $\sigma \in (2^{\{m_1,m_2,m_1'\}})^\omega$ if, and only if, it contains m_2 at some point in time $k \geq 0$ and m_1' at some (possibly different) point in time $k' \geq 0$, while it does not contain m_1 for all points in time $k'' \geq 0$ with $k'' \leq \max\{k, k'\}$; meeting our intuition that $\mathcal{B}_{1,\mathcal{A}_\varphi}$ accepts words that satisfy the specification $\Diamond m_1 \wedge \Diamond m_2$ as fast as possible. △

The alternating co-Büchi automaton $\mathcal{B}_{i,\mathcal{A}_\varphi}$ constructed from the alternating co-Büchi automata $\mathcal{A}_\varphi$ and $\mathcal{A}_{\neg\varphi}$ according to Definition 3.6 is sound and complete in the sense that it recognizes whether or not Duplicator has a winning strategy in an delay-dominance game. That is, given a process p_i and two infinite words $\sigma, \sigma' \in (2^V)^\omega$ with $\sigma \cap (V \setminus O_i) = \sigma' \cap (V \setminus O_i)$, the automaton $\mathcal{B}_{i,\mathcal{A}_\varphi}$ accepts the infinite word $\sigma' \cup pr(\sigma \cap O_i)$ if, and only if, Duplicator wins the delay-dominance game $\mathbb{G} = (\mathcal{A}_\varphi, \sigma, \sigma')$. The main reason for soundness and completeness is that a run tree of $\mathcal{B}_{i,\mathcal{A}_\varphi}$ induced by σ' can be translated into a strategy for Duplicator in the delay-dominance game $\mathbb{G}$ and vice versa since, by construction, both define the existential choices in $\mathcal{A}_\varphi$ for σ' and the universal choices in $\mathcal{A}_\varphi$ for σ. First, we focus on soundness. From the above observation regarding the connection of run trees of $\mathcal{B}_{i,\mathcal{A}_\varphi}$ and strategies in $\mathbb{G}$, it follows that for a run tree of $\mathcal{B}_{i,\mathcal{A}_\varphi}$ whose branches all visit only finitely many rejecting states, there exists a strategy for Duplicator in the delay-dominance game $\mathbb{G}$ that ensures that for all consistent plays either $\sigma \not\models \varphi$ holds or, by construction of ϑ and δ^A, every rejecting dominant state is matched with a rejecting alternative state eventually. Formally:

Lemma 3.6. *Let φ be an LTL formula over atomic propositions V. Let $p_i \in P^-$ be a system process. Let $\mathcal{A}_\varphi$ and $\mathcal{A}_{\neg\varphi}$ be alternating co-Büchi automata with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$ and $\mathcal{L}(\mathcal{A}_{\neg\varphi}) = \mathcal{L}(\neg\varphi)$. Let $\mathcal{B}_{i,\mathcal{A}_\varphi}$ be the alternating co-Büchi automaton constructed from $\mathcal{A}_\varphi$ and $\mathcal{A}_{\neg\varphi}$ according to Definition 3.6. Let $\sigma, \sigma' \in (2^V)^\omega$ be sequences with $\sigma \cap (V \setminus O_i) = \sigma' \cap (V \setminus O_i)$. If $\mathcal{B}_{i,\mathcal{A}'_\varphi}$ accepts $\sigma' \cup pr(\sigma \cap O_i)$, then Duplicator wins the delay-dominance game $(\mathcal{A}_\varphi, \sigma, \sigma')$.*

Proof. For the sake of readability, let $\hat{\sigma} = \sigma' \cup pr(\sigma \cap O_i)$. Furthermore, let $\hat{\sigma}' \in (2^V)^\omega$ be the sequence obtained from $\hat{\sigma}$ by removing all unprimed outputs of p_i and by then making all primed outputs of p_i unprimed, i.e., $\hat{\sigma}' = (\hat{\sigma} \cap (V \setminus O_i)) \cup unpr(\hat{\sigma} \cap O'_i)$. Since $\mathcal{B}_{i,\mathcal{A}'_\varphi}$ accepts $\hat{\sigma}$ by assumption, there exists a run tree $r \in Runs(\mathcal{B}_{i,\mathcal{A}_\varphi}, \hat{\sigma})$ of $\mathcal{B}_{i,\mathcal{A}_\varphi}$ induced by $\hat{\sigma}$ whose branches all visit only finitely many rejecting states. By definition, r defines the existential choices occurring in $\mathcal{B}_{i,\mathcal{A}_\varphi}$ when reading $\hat{\sigma}$. Thus, in particular, r defines the choice in the initial state $(q_0, q_0, \top)$ for, intuitively, either entering the alternating co-Büchi automaton $\mathcal{A}_{\neg\varphi}$ for the negated specification or for entering the product automaton part of $\mathcal{B}_{i,\mathcal{A}_\varphi}$.

First, suppose that r defines to enter the alternating co-Büchi automaton $\mathcal{A}_{\neg\varphi}$. Then, by construction of $\mathcal{B}_{i,\mathcal{A}_\varphi}$, there is a run tree $\tilde{r}$ of $\mathcal{A}_{\neg\varphi}$ induced by $\hat{\sigma}'$ that only differs from r in the labeling of the root. In $\tilde{r}$, the root is labeled with q_0, while it is labeled with $(q_0, q_0, \top)$ in r. Thus, by definition of the set F^A of rejecting states of $\mathcal{B}_{i,\mathcal{A}_\varphi}$, all branches of the run tree $\tilde{r}$ visit only finitely many rejecting states as well. Hence, $\mathcal{A}_{\neg\varphi}$ accepts $\hat{\sigma}'$ and thus we have $\hat{\sigma}' \in \mathcal{L}(\mathcal{A}_{\neg\varphi})$. Since $\mathcal{L}(\mathcal{A}_{\neg\varphi}) = \mathcal{L}(\neg\varphi)$ holds by assumption, $\hat{\sigma}' \not\models \varphi$ follows. By definition of $\hat{\sigma}$, we have $\hat{\sigma} \cap (V \setminus O_i) = \sigma' \cap (V \setminus O_i)$ and hence $\hat{\sigma} \cap (V \setminus O_i) = \sigma \cap (V \setminus O_i)$ follows since $\sigma \cap (V \setminus O_i) = \sigma' \cap (V \setminus O_i)$ holds by assumption. Furthermore, since $\hat{\sigma} \cap O'_i = pr(\sigma \cap O_i)$ holds by definition of the sequence $\hat{\sigma}$, we have $unpr(\hat{\sigma} \cap O'_i) = \sigma \cap O_i$. Thus, $\hat{\sigma}' = \sigma$ holds and hence we have $\sigma \not\models \varphi$. Therefore, it follows with Lemma 3.5, that Duplicator wins the delay-dominance game $(\mathcal{A}_\varphi, \sigma, \sigma')$.

Second, suppose that r defines to enter the product automaton part of $\mathcal{B}_{i,\mathcal{A}_\varphi}$. Then, we construct a strategy μ for Duplicator in the delay-dominance game $\mathbb{G} = (\mathcal{A}_\varphi, \sigma, \sigma')$ from r as follows. Let $v \cdot v$ be a finite sequence of positions with $v \in P^*$ and $v \in P$. We only define μ explicitly on sequences $v \cdot v$ that can occur in the delay-dominance game $\mathbb{G}$ and where v is controlled by Duplicator; on all other sequences we define $\mu(v, v) = v'$ for some arbitrary $v' \in P$ that is a valid extension of $v \cdot v$. Thus, in the following we assume that $v \cdot v$ is a prefix that can occur in the game and that v is of the form $((p, q, c), j)$ or $((p, q, c, q'), j)$. We map $v \cdot v$ to a prefix of a branch of the run tree r if there is a compatible one: a compatible branch b of r agrees with the finite projected play $\hat{v}$ up to point in time $|v| - 1$. Note her that, slightly misusing notation, we apply the definition of a projected play also to the finite prefix ν of a play. Moreover, no matter whether v is of the form $((p,q,c),j)$ or $((p,q,c,q'),j)$, we have $b_{|\nu|} = (p,q,m)$ for some $m \in \{\top,\bot\}$. If there is no compatible branch in r, we again define $\mu(\nu,v) = v'$ for some arbitrary position $v' \in P$ that is a valid extension of $\nu \cdot v$. Otherwise, the successors of (p,q,m) in b define the choice of μ: by definition, the set $\mathcal{S}$ of successors of (p,q,m) satisfies $\delta^A((p,q,m),\hat{\sigma}_{|\nu|})$. Therefore, for all sets $c \in \delta(p,\hat{\sigma}'_{|\nu|})$, there is some set $c' \in \delta(q,\hat{\sigma}_{|\nu|} \cap V)$ such that for all states $q' \in c'$, there is some state $p' \in c$ such that we have $\vartheta(p',q',m) \in \mathcal{S}$. Note here that we do not distinguish between the initial state $(q_0,q_0,\top)$ and other states (p,q,m) of $\mathcal{B}_{i,\mathcal{A}_\varphi}$ since, by assumption, the run tree r defines the choice of entering the product automaton part of $\mathcal{B}_{i,\mathcal{A}_\varphi}$ and thus the choice of the second disjunct for $(q_0,q_0,\top)$ which coincides with δ^A for other states (p,q,m). If $v = ((p,q,c),j)$ holds, we thus define $\mu(\nu,v) = ((p,q,c'_c),j)$, where the choice of c' is based on c. If $v = ((p,q,c,q'),j)$ holds, then we define $\mu(\nu,v) = ((p',q'),j+1)$, where the choice of p' is based on c, c', and q'. Since $\hat{\sigma} \cap V = \sigma$ and $\hat{\sigma}' = \sigma'$ hold, μ is indeed a strategy for Duplicator in the delay-dominance game $\mathbb{G}$.

It remains to show that μ is winning for Duplicator from the initial position v_0 of $\mathbb{G}$. Let $\rho \in \mathit{Plays}(\mathbb{G},\mu)$ be some initial play in $\mathbb{G}$ that is consistent with μ. Then, by construction of μ, there is a branch b of the run tree r that coincides with the projected play $\hat{\rho}$ in the states p and q, i.e., we have $\hat{\rho} = b'$, where b' is the sequence obtained from b when removing the mark m from all states (p,q,m) of $\mathcal{B}_{i,\mathcal{A}_\varphi}$. By assumption, all branches of r contain only finitely many visits to rejecting states. Thus, in particular the branch b with $b' = \hat{\rho}$ contains only finitely many visits to rejecting states. Hence, by construction of $\mathcal{B}_{i,\mathcal{A}_\varphi}$ and since, by assumption, we only consider the product automaton part of $\mathcal{B}_{i,\mathcal{A}_\varphi}$, the branch b thus contains only finitely many visits to states of the form $(p,q,\bot)$. Furthermore, by definition of ϑ, we only have $m = \bot$ for a state (p,q,m) at point in time $k \geq 0$ in b if either (i) $p \notin F$ and $q \in F$ holds, or if (ii) $p' \notin F$ and $q' \in F$ holds for (p',q',m') at some point in time $k' < k$ in b and $p'' \notin F$ holds for (p'',q'',m'') at all points in time k'' with $k' \leq k'' \leq k$. Therefore, since b visits only finitely many states of the form $(p,q,\bot)$, there are only finitely many points in time, where b visits a rejecting dominant state while it does not visit a rejecting alternative state, and for all these points in time there are only finitely many following steps until a rejecting alternative state is visited. Thus, in particular, $\#_2(b'_k) \in F \rightarrow \exists k' \geq k.\ \#_1(b'_{k'}) \in F$ holds for all points in time $k \geq 0$. Since we have $b' = \hat{\rho}$ by construction, it thus follows that $\rho \in \mathbb{W}$ holds. Therefore, Duplicator wins the delay-dominance game $(\mathcal{A}_\varphi,\sigma,\sigma')$. □

Thus, if the alternating co-Büchi automaton $\mathcal{B}_{i,\mathcal{A}_\varphi}$ constructed according to Definition 3.6 accepts some infinite sequence $\hat{\sigma} \in (2^{V \cup O'_i})^\omega$ constructed from two sequences $\sigma,\sigma' \in (2^V)^\omega$ that only differ on outputs of process p_i such that $\hat{\sigma} = \sigma' \cup pr(\sigma \cap O_i)$ holds, then Duplicator wins the delay-dominance game $\mathbb{G} = (\mathcal{A}_\varphi,\sigma,\sigma')$, where $\mathcal{A}_\varphi$ is the alternating co-Büchi automaton representing the LTL specification φ from which $\mathcal{B}_{i,\mathcal{A}_\varphi}$ is constructed. Next, we consider completeness of $\mathcal{B}_{i,\mathcal{A}_\varphi}$. Similarly to the main idea behind soundness, a winning strategy for Duplicator in $\mathbb{G}$ can be translated into a run tree r of $\mathcal{B}_{i,\mathcal{A}_\varphi}$. If $\sigma \models \varphi$ holds, then r visits only finitely many rejecting states since only finitely many rejecting dominant states are visited. If $\sigma \not\models \varphi$ holds, then there exists a run tree, namely one entering the part of $\mathcal{B}_{i,\mathcal{A}_\varphi}$ that coincides with $\mathcal{A}_{\neg\varphi}$, whose branches all visit only finitely many rejecting states. Formally:

Lemma 3.7. *Let φ be an LTL formula over atomic propositions V. Let $p_i \in P^-$ be a system process. Let $\mathcal{A}_\varphi$ and $\mathcal{A}_{\neg\varphi}$ be alternating co-Büchi automata with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$ and $\mathcal{L}(\mathcal{A}_{\neg\varphi}) = \mathcal{L}(\neg\varphi)$. Let $\mathcal{B}_{i,\mathcal{A}_\varphi}$ be the alternating co-Büchi automaton constructed from $\mathcal{A}_\varphi$ and $\mathcal{A}_{\neg\varphi}$ according to Definition 3.6. Let $\sigma, \sigma' \in (2^V)^\omega$ be sequences with $\sigma \cap (V \setminus O_i) = \sigma' \cap (V \setminus O_i)$. If Duplicator wins the delay-dominance game $(\mathcal{A}_\varphi, \sigma, \sigma')$, then $\mathcal{B}_{i,\mathcal{A}'_\varphi}$ accepts $\sigma' \cup pr(\sigma \cap O_i)$.*

Proof. Let $\mathcal{A}_\varphi = (Q, q_0, \delta, F)$ and let $\mathcal{B}_{i,\mathcal{A}_\varphi} = (Q^A, {q_0}^A, \delta^A, F^A)$. For the sake of readability, let $\hat{\sigma} = \sigma' \cup pr(\sigma \cap O_i)$. Furthermore, let $\hat{\sigma}' \in (2^V)^\omega$ be the infinite sequence obtained from $\hat{\sigma}$ by removing all unprimed outputs of process p_i and by then making all primed outputs of p_i unprimed afterward, i.e., we have $\hat{\sigma}' = (\hat{\sigma} \cap (V \setminus O_i)) \cup unpr(\hat{\sigma} \cap O'_i)$. Since Duplicator wins the delay-dominance game $\mathbb{G} = (\mathcal{A}_\varphi, \sigma, \sigma')$ by assumption, it follows with Lemma 3.5 that either (i) $\sigma \models \varphi$ holds or (ii) Duplicator has a winning strategy μ in the delay-dominance game $\mathbb{G}$ and for every initial play $\rho \in Plays(\mathbb{G}, \mu)$ that is consistent with μ, there is a point in time $k \geq 0$ such that $f_{dom}(\rho_{k'}) \notin F$ holds for all $k' \geq k$, i.e., ρ does not contain any rejecting dominant states from some point in time $k \geq 0$ on and thus it contains only finitely many visits to rejecting dominant states. We distinguish two cases.

First, suppose that (i) holds. Then, we have $\sigma \not\models \varphi$ and hence $\sigma \models \neg\varphi$ holds. Therefore, we have $\sigma \in \mathcal{L}(\neg\varphi)$ and thus, since $\mathcal{L}(\mathcal{A}_{\neg\varphi}) = \mathcal{L}(\neg\varphi)$ holds by assumption, $\sigma \in \mathcal{L}(\mathcal{A}_{\neg\varphi})$ follows. Thus, there exists a run tree r of $\mathcal{A}_{\neg\varphi}$ induced by σ whose branches all contain only finitely many visits to rejecting states. By construction of $\mathcal{B}_{i,\mathcal{A}_\varphi}$, there exists a corresponding run tree $\tilde{r}$ of $\mathcal{B}_{i,\mathcal{A}_\varphi}$ that only differs from r in the labeling of the root. In r, the root is labeled with q_0, while it is labeled with $(q_0, q_0, \top)$ in $\tilde{r}$. Hence, by definition of the rejecting states F^A of $\mathcal{B}_{i,\mathcal{A}_\varphi}$, all branches of the run tree $\tilde{r}$ contain only finitely many visits to rejecting states as well. Moreover, since $\mathcal{A}_{\neg\varphi}$ is an alternating co-Büchi automaton with alphabet V and by construction of the transition function δ^A of $\mathcal{B}_{i,\mathcal{A}_\varphi}$, the successors in $\tilde{r}$ only depend on the valuations of the variables in $(V \setminus O_i) \cup O'_i$ or, more precisely, on their unprimed versions, and thus, in particular, are solely defined by σ. Therefore, all infinite sequences $\sigma'' \in (2^{V \cup O'_i})^\omega$ with $(\sigma'' \cap (V \setminus O_i)) \cup unpr(\sigma'' \cap O'_i) = \sigma$ induce the run tree $\tilde{r}$. Hence, in particular, the sequence $\hat{\sigma}$ induces the run tree $\tilde{r}$: by definition of $\hat{\sigma}$, we have $\hat{\sigma} \cap (V \setminus O_i) = \sigma' \cap (V \setminus O_i)$ and therefore $\hat{\sigma} \cap (V \setminus O_i) = \sigma \cap (V \setminus O_i)$ follows since $\sigma \cap (V \setminus O_i) = \sigma' \cap (V \setminus O_i)$ holds by assumption. Furthermore, $\hat{\sigma} \cap O'_i = pr(\sigma \cap O_i)$ holds by construction of $\hat{\sigma}$ and therefore we obtain that $unpr(\hat{\sigma} \cap O'_i) = \sigma \cap O_i$ holds. Hence, $(\hat{\sigma} \cap (V \setminus O_i)) \cup unpr(\hat{\sigma} \cap O'_i) = \sigma$ follows. Consequently, $\hat{\sigma}$ induces a run tree of $\mathcal{B}_{i,\mathcal{A}_\varphi}$ that contains only finitely many visits to rejecting states, namely $\tilde{r}$. Therefore, $\mathcal{B}_{i,\mathcal{A}_\varphi}$ accepts $\hat{\sigma}$.

Second, suppose that (ii) holds. Then, Duplicator has a winning strategy μ in the delay-dominance game $\mathbb{G} = (\mathcal{A}_\varphi, \sigma, \sigma')$ and for every initial play $\rho \in Plays(\mathbb{G}, \mu)$ that is consistent with μ, there is a point in time $k \geq 0$ such that $f_{dom}(\rho_{k'}) \notin F$ holds for all k' with $k' \geq k$. Let $f : (Q \times Q)^\omega \to (Q \times Q \times \{\top, \bot\})^\omega$ be a function that, given an infinite sequence $\chi \in (Q \times Q)^\omega$ of state tuples (p, q), returns an extended sequence $\chi' \in (Q \times Q \times \{\top, \bot\})^\omega$ that is incrementally defined as follows: for the initial point in time, let $\chi'_0 := (p, q, \top)$ if $\chi_0 = (p, q)$. For a point in time $k > 0$, let $\chi'_k := \vartheta(p', q', m)$ if $\chi'_{k-1} = (p, q, m)$ and $\chi_k = (p', q')$. Here, ϑ denotes the corresponding function used in Definition 3.6. We construct a Q-labeled tree $(\mathbb{T}, \ell)$ from μ as follows by defining the labeling of the root as well as of the successors of all nodes. The labeling of the root ε of the tree $\mathbb{T}$ is defined by $\ell(\varepsilon) = (q_0, q_0, \top)$. Let $x \in \mathbb{T}$ be a node of $\mathbb{T}$ and let $k = |x|$ be its depth. Let $\nu = \mathit{pref}(\mathbb{T}, x)$ be x's prefix in $\mathbb{T}$, i.e. the unique finite sequence of nodes in $\mathbb{T}$ that, starting from the root node ε, reaches node x. We define the labeling of the successor nodes $\mathit{children}(x)$ of x such that

$$\{\ell(x') \mid x' \in \mathit{children}(x)\} = \{f(\hat{\rho}_{k+1}) \mid \rho \in \mathit{Plays}(\mathbb{G}, \mu) \land \forall 0 \le k' \le k.\, f(\hat{\rho}_{k'}) = \ell(\nu_{k'})\}$$

holds. Next, we show that $(\mathbb{T}, \ell)$ is a run tree of $\mathcal{B}_{i,\mathcal{A}_\varphi}$ that is induced by $\hat{\sigma}$. Since it follows immediately from the construction of $(\mathbb{T}, \ell)$ that $\ell(\varepsilon) = (q_0, q_0, \top)$ holds, which is the initial state of $\mathcal{B}_{i,\mathcal{A}_\varphi}$, it only remains to show that $\{\ell(x') \mid x' \in \mathit{children}(x)\} \models \delta(\ell(x), \hat{\sigma}_{|x|})$ holds for every node $x \in \mathbb{T}$. Let $x \in \mathbb{T}$ be a node of $\mathbb{T}$ and let $k = |x|$ be its depth. Let $\nu = \mathit{pref}(\mathbb{T}, x)$ be x's prefix in $\mathbb{T}$, i.e. the unique finite sequence of nodes in $\mathbb{T}$ that, starting from the root node ε, reaches node x. Let $\mathcal{S} = \{\hat{\rho}_{k+1} \mid \rho \in \mathit{Plays}(\mathbb{G}, \mu) \land \forall 0 \le k' \le k.\, f(\hat{\rho}_{k'}) = \ell(\nu_{k'})\}$. By construction of the delay-dominance game, the alternative states of an initial play that is consistent with μ intuitively evolves according to a run of $\mathcal{A}^d_\varphi$ induced by σ, where $\mathcal{A}^d_\varphi$ is the alternating co-Büchi automaton obtained from $\mathcal{A}_\varphi$ by dualizing the transition function δ, i.e., by swapping conjunctions and disjunctions. The dominant states, in contrast, evolve according to a run of $\mathcal{A}_\varphi$ induced by σ'. Hence, formally, we know from the construction of the delay-dominance game that $\mathcal{S}$ satisfies

$$\bigwedge_{c \in \delta(p, \hat{\sigma}'_k)} \; \bigvee_{c' \in \delta(q, \hat{\sigma}_k \cap V)} \; \bigwedge_{q' \in c'} \; \bigvee_{p' \in c} (p', q'),$$

where $p := \#_1(\ell(x))$ and $q := \#_2(\ell(x))$. Furthermore, the function f is defined such that it accurately reflects the marks $\top$ and $\bot$ assigned by the function ϑ in Definition 3.6. Hence, the set $\mathcal{S}' = \{\hat{\rho}_{k+1} \mid \rho \in \mathit{Plays}(\mathbb{G}, \mu) \land \forall 0 \le k' \le k.\, f(\hat{\rho}_{k'}) = \ell(\nu_{k'})\}$ satisfies

$$\bigwedge_{c \in \delta(p, \hat{\sigma}'_k)} \; \bigvee_{c' \in \delta(q, \hat{\sigma}_k \cap V)} \; \bigwedge_{q' \in c'} \; \bigvee_{p' \in c} \vartheta(p', q', m),$$

where $(p, q, m) = \ell(x)$. Therefore, $(\mathbb{T}, \ell)$ is indeed a run tree of $\mathcal{B}_{i,\mathcal{A}_\varphi}$ that is induced by $\hat{\sigma}$.

Lastly, we show that all branches of the run tree $(\mathbb{T}, \ell)$ contain only finitely many visits to rejecting states. Since μ is a winning strategy for Duplicator in the delay-dominance game $\mathbb{G}$ by assumption, we have $\rho \in \mathbb{W}$ for all initial plays $\rho \in \mathit{Plays}(\mathbb{G}, \mu)$ that are consistent with μ. Hence, for all such plays $\rho \in \mathit{Plays}(\mathbb{G}, \mu)$ and all points in time $k \ge 0$, it holds that if we have $f_{dom}(\rho_k) \in F$, then $f_{alt}(\rho_{k'}) \in F$ holds for some point in time k' with $k' \ge k$ as well. Moreover, since (ii) holds by assumption, for every initial play $\rho \in \mathit{Plays}(\mathbb{G}, \mu)$ that is consistent with μ, there exists a point in time $k \ge 0$ such that $f_{dom}(\rho_{k'}) \notin F$ holds for all $k' \ge k$. Thus, there are only finitely many points in time at which ρ visits a rejecting dominant state and for all these points in time it holds that a rejecting alternative state occurs in ρ at the very same point in time or at a later point in time. Therefore, by construction of $(\mathbb{T}, \ell)$ and f, we obtain that there are only finitely many nodes $x \in \mathbb{T}$ with $\#_3(\ell(x)) = \bot$. Hence, since only states of the form (p, q, m) are reached and since for these states the ones with mark $\bot$ are the only rejecting ones of $\mathcal{B}_{i,\mathcal{A}_\varphi}$, all branches of $(\mathbb{T}, \ell)$ visit only finitely many rejecting states. Hence, since $(\mathbb{T}, \ell)$ is a run tree of $\mathcal{B}_{i,\mathcal{A}_\varphi}$ induced by $\hat{\sigma}$, it follows that $\mathcal{B}_{i,\mathcal{A}_\varphi}$ accepts $\hat{\sigma}$. □

Thus, if Duplicator has a winning strategy in the delay-dominance game $(\mathcal{A}_\varphi, \sigma, \sigma')$ for two sequences $\sigma, \sigma' \in (2^V)^\omega$ that only differ on outputs of process p_i, then the alternating co-Büchi automaton $\mathcal{B}_{i,\mathcal{A}_\varphi}$ constructed from $\mathcal{A}_\varphi$ and $\mathcal{A}_{\neg\varphi}$ accepts the infinite sequence $\sigma' \cup pr(\sigma \cap O_i)$. Therefore, it follows immediately from Lemmas 3.6 and 3.7 that the alternating co-Büchi automaton $\mathcal{B}_{i,\mathcal{A}_\varphi}$ constructed as described in Definition 3.6 is sound and complete in the sense that it recognizes whether or not Duplicator wins the delay-dominance game:

Theorem 3.4. *Let φ be an LTL formula over atomic propositions V. Let $p_i \in P^-$ be a system process. Let $\mathcal{A}_\varphi$ and $\mathcal{A}_{\neg\varphi}$ be alternating co-Büchi automata with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$ and $\mathcal{L}(\mathcal{A}_{\neg\varphi}) = \mathcal{L}(\neg\varphi)$. Let $\mathcal{B}_{i,\mathcal{A}_\varphi}$ be the alternating co-Büchi automaton constructed from $\mathcal{A}_\varphi$ and $\mathcal{A}_{\neg\varphi}$ according to Definition 3.6. Let $\sigma, \sigma' \in (2^V)^\omega$ be sequences with $\sigma \cap (V \setminus O_i) = \sigma' \cap (V \setminus O_i)$. Then, $\mathcal{B}_{i,\mathcal{A}'_\varphi}$ accepts $\sigma' \cup pr(\sigma \cap O_i)$ if, and only if, Duplicator wins the delay-dominance game $(\mathcal{A}_\varphi, \sigma, \sigma')$.*

Since $\mathcal{B}_{i,\mathcal{A}_\varphi}$ thus determines whether or not Duplicator has a winning strategy in an delay-dominance game, it follows immediately from the definition of delay-dominance that $\mathcal{B}_{i,\mathcal{A}_\varphi}$ is suitable alternating co-Büchi automaton for determining whether or not a process strategy s_i for p_i delay-dominates another process strategy t_i for p_i on some input sequence:

Corollary 3.3. *Let φ be an LTL formula over atomic propositions V. Let $p_i \in P^-$ be a system process. Let $\mathcal{A}_\varphi$ and $\mathcal{A}_{\neg\varphi}$ be alternating co-Büchi automata with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$ and $\mathcal{L}(\mathcal{A}_{\neg\varphi}) = \mathcal{L}(\neg\varphi)$. Let $\mathcal{B}_{i,\mathcal{A}_\varphi}$ be the alternating co-Büchi automaton constructed from $\mathcal{A}_\varphi$ and $\mathcal{A}_{\neg\varphi}$ according to Definition 3.6. Let s_i and t_i be strategies for p_i. Let $\gamma \in (2^{I_i})^\omega$. Then, $t_i \trianglelefteq_{\mathcal{A}_\varphi,\gamma} s_i$ holds if, and only if, $\mathcal{B}_{i,\mathcal{A}_\varphi}$ accepts $comp(s_i, \gamma) \cup pr(comp(t_i, \gamma) \cap O_i) \cup \gamma'$ for all $\gamma' \in (2^{V \setminus V_i})^\omega$.*

Yet, although $\mathcal{B}_{i,\mathcal{A}_\varphi}$ recognizes whether or not a strategy s_i delay-dominates another strategy t_i on some input sequence, it cannot directly be used for synthesizing delay-dominant strategies. First, $\mathcal{B}_{i,\mathcal{A}_\varphi}$ is an alternating co-Büchi automaton, while we require a universal co-Büchi automaton for bounded synthesis. Second, it considers *one particular* alternative strategy t_i. For recognizing delay-dominance, however, we need to consider *all* alternative strategies. In the remainder of this section, we thus describe how $\mathcal{B}_{i,\mathcal{A}_\varphi}$ can be translated into a universal co-Büchi automaton for bounded synthesis of delay-dominant strategies.

3.4.2. Construction of the UCA for Bounded Synthesis

Next, we translate the alternating co-Büchi automaton $\mathcal{B}_{i,\mathcal{A}_\varphi}$ constructed as described in the previous subsection to a universal co-Büchi automaton $\mathcal{A}_{i,\mathcal{A}_\varphi}$ that can be utilized in existing bounded synthesis frameworks for synthesizing delay-dominant strategies for a system process $p_i \in P^-$. As outlined before, we need to (i) translate $\mathcal{B}_{i,\mathcal{A}_\varphi}$ into a *universal* co-Büchi automaton, and (ii) ensure that the automaton considers *all* alternative strategies for process p_i instead of a particular one. Therefore, we proceed in two steps.

First, we translate the alternating co-Büchi automaton $\mathcal{B}_{i,\mathcal{A}_\varphi}$ into an equivalent universal co-Büchi automaton $\mathcal{U}_{i,\mathcal{A}_\varphi}$. We utilize the well-known Miyano-Hayashi algorithm [MH84] for translating alternating Büchi automata into nondeterministic Büchi automata. It introduces an exponential blowup: the resulting nondeterministic Büchi automaton is of exponential size in the number of states of the initial alternating Büchi automaton. Since we consider co-Büchi rather than Büchi automata, we need a translation from an alternating co-Büchi automaton to an equivalent universal co-Büchi automaton. Recall from Section 2.5 that the Büchi and co-Büchi acceptance conditions as well as nondeterministic and universal branching are dual. Utilizing this duality, we can reuse Miyano and Hayashi's result for co-Büchi automata:

Lemma 3.8. *Let $\mathcal{A}$ be an alternating co-Büchi automaton with m states. There exists a universal co-Büchi automaton $\mathcal{B}$ with $O(2^m)$ states such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$ holds.*

Proof. Let $\mathcal{A} = (Q, Q_0, \delta, F)$. Let $\mathcal{A}_d = (Q^d, Q_0^d, \delta^d, F^d)$ be the dual automaton of $\mathcal{A}$, i.e., the alternating Büchi automaton with $Q^d = Q$, $Q_0^d = Q_0$, $F^d = F$, and $\delta^d(u, \boldsymbol{i}) = \bigwedge_{c \in \delta(u,\boldsymbol{i})} \bigvee_{u' \in c} u'$. Then $\mathcal{L}(\mathcal{A}_d) = \overline{\mathcal{L}(\mathcal{A})}$ holds due to the duality of nondeterministic and universal branching as well as of the Büchi and co-Büchi acceptance condition. As shown by Miyano and Hayashi [MH84], there exists a nondeterministic Büchi automaton $\mathcal{B}'$ with $O(2^{|Q^d|})$ states and with $\mathcal{L}(\mathcal{B}') = \mathcal{L}(\mathcal{A}^d)$. Let $\mathcal{B}$ be the dual automaton of $\mathcal{B}'$, i.e., the universal co-Büchi automaton that is a copy of $\mathcal{B}'$, but where the nondeterministic transitions are interpreted as universal ones and where the accepting states are interpreted as rejecting states. Then, $\mathcal{B}$ has $O(2^{|Q^d|})$ states and we have $\mathcal{L}(\mathcal{B}) = \overline{\mathcal{L}(\mathcal{B}')}$. Since $\mathcal{L}(\mathcal{B}') = \mathcal{L}(\mathcal{A}^d) = \overline{\mathcal{L}(\mathcal{A})}$ holds, we obtain $\mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{A})$. Thus, $\mathcal{B}$ is the desired universal co-Büchi automaton. □

Next, we construct the desired universal co-Büchi automaton $\mathcal{A}_{i,\mathcal{A}_\varphi}$ that recognizes delay-dominant strategies for $\mathcal{A}_\varphi$ and system process $p_i \in P^-$. For this sake, we need to adapt the universal co-Büchi automaton $\mathcal{U}_{i,\mathcal{A}_\varphi}$ to consider *all* alternative strategies for p_i instead of a particular one. Similar to the automaton construction for synthesizing remorsefree dominant strategies [DF14, FP20a], we utilize *universal projection* (see Definition 2.22) as described in Section 2.8.2. Intuitively, the projected automaton $\pi_X(\mathcal{A})$ for a universal co-Büchi automaton $\mathcal{A}$ over alphabet Σ and a set $X \subset \Sigma$ contains the transitions of $\mathcal{A}$ for *all* possible valuations of the variables in $\Sigma \setminus X$. Hence, for a sequence $\sigma \in (2^X)^\omega$, all runs of $\mathcal{A}$ on sequences extending σ with some valuation of the variables in $\Sigma \setminus X$ are also runs of the projected automaton $\pi_X(\mathcal{A})$. Since both $\mathcal{A}$ and $\pi_X(\mathcal{A})$ are universal automata, $\pi_X(\mathcal{A})$ thus accepts a sequence $\sigma \in (2^X)^\omega$ if, and only if, $\mathcal{A}$ accepts all sequences extending σ with some valuation of the variables in $\Sigma \setminus X$ (see Lemma 2.4). We utilize this property to obtain a universal co-Büchi automaton $\mathcal{A}_{i,\mathcal{A}_\varphi}$ from $\mathcal{U}_{i,\mathcal{A}_\varphi}$ that considers *all* possible alternative strategies for p_i instead of only a particular one: we project to the unprimed variables of the alphabet of $\mathcal{U}_{i,\mathcal{A}_\varphi}$, i.e., to V, thereby quantifying universally over the alternative strategies. We thus obtain a universal co-Büchi automaton that recognizes delay-dominant strategies for system process p_i as follows:

Definition 3.7 (Delay-Dominance Automaton).
Let φ be an LTL formula over atomic propositions V. Let $p_i \in P^-$ be a system process. Let $\mathcal{A}_\varphi$ and $\mathcal{A}_{\neg\varphi}$ be alternating co-Büchi automata with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$ and $\mathcal{L}(\mathcal{A}_{\neg\varphi}) = \mathcal{L}(\neg\varphi)$. Let $\mathcal{B}_{i,\mathcal{A}_\varphi}$ be the alternating co-Büchi automaton constructed from $\mathcal{A}_\varphi$ and $\mathcal{A}_{\neg\varphi}$ according to Definition 3.6. Let $\mathcal{U}_{i,\mathcal{A}_\varphi}$ be a universal co-Büchi automaton with $\mathcal{L}(\mathcal{B}_{i,\mathcal{A}_\varphi}) = \mathcal{L}(\mathcal{U}_{i,\mathcal{A}_\varphi})$. The *delay-dominance automaton* $\mathcal{A}_{i,\mathcal{A}_\varphi}$ for $\mathcal{A}_\varphi$ and p_i is defined by $\mathcal{A}_{i,\mathcal{A}_\varphi} = \pi_V(\mathcal{U}_{i,\mathcal{A}_\varphi})$.

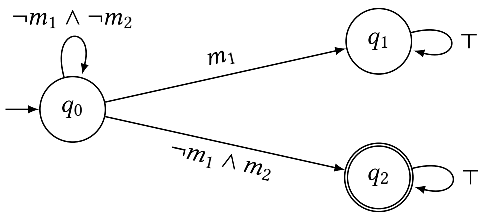


Figure 3.11.: Delay-dominance automaton $\mathcal{A}_{i,\mathcal{A}_\varphi}$ for system process p_1 constructed from $\mathcal{A}_\varphi$ depicted in Figure 3.2 for the running example after simplification.

Hence, the construction of the delay-dominance automaton $\mathcal{A}_{i,\mathcal{A}_\varphi}$ indeed transforms the intermediate alternating co-Büchi automaton $\mathcal{B}_{i,\mathcal{A}_\varphi}$ introduced in Section 3.4.1 into a universal co-Büchi automaton for recognizing delay-dominant strategies in two steps: first, we translate $\mathcal{B}_{i,\mathcal{A}_\varphi}$ into a universal automaton, namely $\mathcal{U}_{i,\mathcal{A}_\varphi}$. Afterward, we consider *all* alternative strategies instead of only a single one by projecting to the unprimed variables.

Example 3.10. Reconsider the message-sending system from the running example and the alternating co-Büchi automaton $\mathcal{A}_\varphi$ from Figure 3.2 describing the specification. Furthermore, consider the intermediate alternating co-Büchi automaton $\mathcal{B}_{1,\mathcal{A}_\varphi}$ for process p_1 constructed from depicted in Figure 3.10. After simplification, the delay-dominance automaton $\mathcal{A}_{i,\mathcal{A}_\varphi}$ for φ and process p_i is given by the universal co-Büchi automaton depicted in Figure 3.11.

It accepts an infinite word $\sigma \in (2^{\{m_1,m_2\}})^\omega$ if, and only if, message m_1 occurs before or at the same time as the first occurrence of message m_2 in σ. In particular, it thus rejects the computation of the strategy t_1, which waits for the other message m_2 before sending its own message (see Figure 3.1b), for all input sequences $\gamma \in (2^{\{m_2\}})^\omega$ in which m_2 occurs at some point in time, i.e., with $\gamma \neq \emptyset^\omega$. Since we model strategies with Moore transducers and since strategies cannot look into the future by definition, a strategy for p_1 cannot wait for m_2 and immediately react with sending its own message. Instead, a strategy that waits for m_2 will always have a delay of at least one time step in sending m_1, such as, for instance, t_1. Therefore, since there is an input sequence that contains m_2 at the very first point in time, only strategies that output their own message in the very first time step ensure that the automaton $\mathcal{A}_{1,\mathcal{A}_\varphi}$ accepts its computation on all input sequences. Thus, in particular, strategy s_1, which sends m_1 in the very first time step and then never afterward (see Figure 3.1a) is recognized by $\mathcal{A}_{1,\mathcal{A}_\varphi}$ as a delay-dominant strategy. This meets our intuition that a delay-dominant strategy needs to satisfy the specification as fast as possible. △

Utilizing the previous results, we can now show soundness and completeness of the delay-dominance universal co-Büchi automaton $\mathcal{A}_{i,\mathcal{A}_\varphi}$. From Theorem 3.4 and, in particular, from Corollary 3.3, we know that $\mathcal{B}_{i,\mathcal{A}_\varphi}$ recognizes whether or not a strategy s_i for process $p_i \in P^-$ delay-dominates another strategy t_i for p_i for $\mathcal{A}_\varphi$ on an input sequence $\gamma \in (2^{I_i})^\omega$. By Lemma 3.8, there exists a universal co-Büchi automaton $\mathcal{U}_{i,\mathcal{A}_\varphi}$ with $\mathcal{L}(\mathcal{U}_{i,\mathcal{A}_\varphi}) = \mathcal{L}(\mathcal{B}_{i,\mathcal{A}_\varphi})$. With the definition of the delay-dominance automaton as well as with Lemma 2.4, it then follows that the universal co-Büchi delay-dominance automaton $\mathcal{A}_{i,\mathcal{A}_\varphi}$ determines whether or not a strategy s_i for process p_i is delay-dominant for $\mathcal{A}_\varphi$. Formally:

Theorem 3.5. *Let φ be an LTL formula over atomic propositions V. Let $p_i \in P^-$ be a system process. Let $\mathcal{A}_\varphi$ be an alternating co-Büchi automaton with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$. Let $\mathcal{A}_{i,\mathcal{A}_\varphi}$ be the delay-dominance automaton for $\mathcal{A}_\varphi$ and p_i as constructed in Definition 3.7. Let s_i be a process strategy for p_i. Then, s_i is delay-dominant for $\mathcal{A}_\varphi$ if, and only if, the universal co-Büchi delay-dominance automaton $\mathcal{A}_{i,\mathcal{A}_\varphi}$ accepts $comp(s_i, \gamma) \cup \sigma$ for all $\gamma \in (2^{I_i})^\omega$ and all $\sigma \in (2^{V \setminus V_i})^\omega$.*

Furthermore, the delay-dominance automaton $\mathcal{A}_{i,\mathcal{A}_\varphi}$ is of convenient size: for an LTL formula φ, there exists an alternating co-Büchi automaton $\mathcal{A}_\varphi$ with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$ such that the delay-dominance automaton $\mathcal{A}_{i,\mathcal{A}_\varphi}$ constructed from $\mathcal{A}_\varphi$ is of exponential size in the squared length of the formula φ. This follows from Lemma 3.8 and from the facts that (i) $\mathcal{A}_\varphi$ and $\mathcal{A}_{\neg\varphi}$ both are of linear size in the length of the LTL formula φ (see Proposition 2.3), and (ii) universal projection preserves the automaton size:

Lemma 3.9. *Let φ be an LTL formula over atomic propositions V. Let $p_i \in P^-$ be a system process. There is an alternating co-Büchi automaton $\mathcal{A}_\varphi$ of size $O(|\varphi|)$ with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$ and a universal co-Büchi automaton $\mathcal{A}_{i,\mathcal{A}_\varphi}$ of size $O(2^{|\varphi|^2})$ such that a strategy s_i for p_i is delay-dominant for $\mathcal{A}_\varphi$ if, and only if, $\mathcal{A}_{i,\mathcal{A}_\varphi}$ accepts $comp(s_i, \gamma) \cup \sigma$ for all $\gamma \in (2^{I_i})^\omega$ and all $\sigma \in (2^{V \setminus V_i})^\omega$.*

Proof. Given an LTL formula φ, there are alternating co-Büchi automat $\mathcal{A}_\varphi = (Q, Q_0, \delta, F)$ and $\mathcal{A}_{\neg\varphi} = (Q^c, Q^c_0, \delta^c, F^c)$, both of size $O(|\varphi|)$, with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$ and $\mathcal{L}(\mathcal{A}_{\neg\varphi}) = \mathcal{L}(\neg\varphi)$ by Proposition 2.3. By Theorem 3.4, the automaton $\mathcal{A}_{i,\mathcal{A}_\varphi}$ constructed according to Definition 3.7 satisfies the property that a strategy s_i for process p_i is delay-dominant for $\mathcal{A}_\varphi$ if, and only if, $\mathcal{A}_{i,\mathcal{A}_\varphi}$ accepts $comp(s_i, \gamma) \cup \gamma'$ for all $\gamma \in (2^{I_i})^\omega$ and all $\gamma' \in (2^{V \setminus V_i})^\omega$. Let $\mathcal{B}_{i,\mathcal{A}_\varphi}$ and $\mathcal{U}_{i,\mathcal{A}_\varphi}$ be the intermediate automata from which $\mathcal{A}_{i,\mathcal{A}_\varphi}$ is constructed. The alternating co-Büchi automaton $\mathcal{B}_{i,\mathcal{A}_\varphi}$ is of size $O(|Q|^2 + |Q^c|)$ by construction. By Lemma 3.8 and by construction, the universal co-Büchi automaton $\mathcal{U}_{i,\mathcal{A}_\varphi}$ is of size $O(2^m)$, where m is the number of states of $\mathcal{B}_{i,\mathcal{A}_\varphi}$. Hence, $\mathcal{U}_{i,\mathcal{A}_\varphi}$ has $O(2^{|Q|^2+|Q^c|})$ states. Since the universal projection does not affect the size of an automaton as it only alters the transition relation, $\mathcal{A}_{i,\mathcal{A}_\varphi}$ has $O(2^{|Q|^2+|Q^c|})$ states as well. Since both $\mathcal{A}_\varphi$ and $\mathcal{A}_{\neg\varphi}$ have $O(|\varphi|)$ states, the claim follows. □

Note here that utilizing an *alternating* co-Büchi automaton $\mathcal{A}_\varphi$ for representing the LTL specification φ, i.e., with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$, as starting point of the construction of the delay-dominance automaton, is crucial for this result. If we would have started with a *universal* co-Büchi automaton describing φ, then the delay-dominance automaton would be of size $O(2^{2^{|\varphi|^2}})$: constructing a universal automaton instead of an alternating automaton from φ introduces an exponential blowup. The construction of the automaton $\mathcal{B}_{i,\mathcal{A}_\varphi}$, however, yields an alternating automaton irrespective of whether $\mathcal{A}_\varphi$ is alternating or universal due to the need of keeping track of two sequences in one copy of $\mathcal{A}_\varphi$ and one copy of the *dual* automaton of $\mathcal{A}_\varphi$. Hence, we need to translate $\mathcal{B}_{i,\mathcal{A}_\varphi}$ to a universal co-Büchi automaton even if we started from a universal automaton $\mathcal{A}_\varphi$, introducing another exponential blowup. Hence, although delay-dominance can be equivalently defined on universal co-Büchi automata instead of alternating co-Büchi automata, utilizing alternating ones allows for avoiding an exponential blowup in the size of the automaton recognizing delay-dominance and thus, as we will show in the following, for more efficient synthesis of delay-dominant strategies.

Since the automaton construction described in this section is sound and complete, the universal co-Büchi automaton $\mathcal{A}_{i,\mathcal{A}_\varphi}$ can be used for synthesizing a delay-dominant strategy for system process $p_i \in P^-$ In fact, the automaton construction immediately enables utilizing existing bounded synthesis tools, which derive winning strategies, for the synthesis of delay-dominant strategies by replacing the universal co-Büchi automaton recognizing winning strategies, i.e., the automaton that accepts the same language as φ with the delay-dominance automaton $\mathcal{A}_{i,\mathcal{A}_\varphi}$.

Similar to the universal co-Büchi automaton recognizing remorsefree dominance [DF14], the delay-dominance automaton $\mathcal{A}_{i,\mathcal{A}_\varphi}$ can be translated into a nondeterministic parity tree automaton with an exponential number of colors and a doubly-exponential number of states in the squared length of the formula. Synthesizing delay-dominant strategies thus reduces to checking tree automata emptiness and, if the automaton is non-empty, to extracting a finite-state transducer, which represents a delay-dominant process strategy, from an accepted tree. This can be done in exponential time in the number of colors and in polynomial time in the number of states [Jur00]. With Lemma 3.9, a doubly-exponential complexity for synthesizing delay-dominant strategies thus follows:

Theorem 3.6. *Let φ be an LTL formula over atomic propositions V. Let $p_i \in P^-$ be a system process. Let $\mathcal{A}_\varphi$ be an alternating co-Büchi automaton of size $O(|\varphi|)$ with $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$. If there exists a delay-dominant strategy for $\mathcal{A}_\varphi$, then it can be computed in* 2EXPTIME.

It is well-known that synthesizing winning strategies is 2EXPTIME-complete [PR89a], see also Theorem 2.1. Since there exists a universal co-Büchi automaton of exponential size in the length of the formula, which recognizes remorsefree dominant strategies, dominant strategies can also be synthesized in 2EXPTIME [DF14]. Synthesizing delay-dominant strategies rather than winning or remorsefree dominant ones thus does not introduce any overhead. At the same time, it allows for a simple compositional synthesis approach for distributed systems for many safety and liveness specifications.

3.5. Compositional Synthesis with Delay-Dominance

In this section, we describe a compositional synthesis approach for distributed systems that utilizes delay-dominant strategies. We extend the algorithm described in [DF14] from safety specifications to general properties by synthesizing delay-dominant strategies instead of remorsefree dominant ones. Hence, given a distributed architecture and an LTL specification φ, the compositional synthesis algorithm proceeds in three steps. First, φ is translated into an equivalent alternating co-Büchi automaton $\mathcal{A}_\varphi$ by constructing an alternating Büchi automaton for $\neg\varphi$ with standard algorithms and by then dualizing the transitions as well as by interpreting accepting states as rejecting states (see Proposition 2.3). Second, for each system process $p_i \in P^-$, we construct the universal co-Büchi automaton $\mathcal{A}_{i,\mathcal{A}_\varphi}$ that recognizes delay-dominant strategies for $\mathcal{A}_\varphi$ and p_i as described in Section 3.4. Note that although the initial automaton $\mathcal{A}_\varphi$ is the same one for every process p_i, the universal co-Büchi automata recognizing delay-dominant strategies differ as, since the processes have different sets of output variables, already the alphabets of the intermediate alternating co-Büchi automaton $\mathcal{B}_{i,\mathcal{A}_\varphi}$ differ for different processes. Third, for each system process $p_i \in P^-$, a delay-dominant strategy s_i is synthesized from the respective universal co-Büchi automaton $\mathcal{A}_{i,\mathcal{A}_\varphi}$ with bounded synthesis [FS13]. By construction of the automata, we can employ standard bounded synthesis algorithms such as, e.g., implemented in the tool BoSy [FFT17], for synthesizing delay-dominant strategies by only exchanging the universal co-Büchi automaton that recognizes winning strategies, i.e., the automaton that accepts the same language as φ, with the delay-dominance automaton $\mathcal{A}_{i,\mathcal{A}_\varphi}$.

If the initial alternating co-Büchi automaton $\mathcal{A}_\varphi$ ensures bad prefixes for delay-dominance, then the parallel composition $s_1 \mathbin{||} \ldots \mathbin{||} s_n$ of the synthesized delay-dominant process strategies is, by Theorem 3.3, delay-dominant again. Thus, the strategies $s_1, \ldots, s_n$ form a correct solution for the distributed synthesis problem if $\mathcal{A}_\varphi$ ensures bad prefixes. Furthermore, if φ is realizable, then $s_1 \mathbin{||} \ldots \mathbin{||} s_n$ is, by Corollary 3.1, winning for φ.

Note that even for realizable LTL formulas φ, there does not necessarily exist a delay-dominant strategy since delay-dominance is not solely defined on the satisfaction of φ but on the *structure* of an equivalent alternating co-Büchi automaton $\mathcal{A}_\varphi$. In certain cases, $\mathcal{A}_\varphi$ can thus "punish" the delay-dominant strategy by introducing rejecting states at clever positions that do not influence acceptance but delay-dominance, preventing the existence of a delay-dominant strategy. However, we experienced that an alternating co-Büchi automaton $\mathcal{A}_\varphi$ constructed with standard algorithms from an LTL formula φ does not punish delay-dominant strategies since $\mathcal{A}_\varphi$ thoroughly follows the structure of φ and thus often does not contain unnecessary rejecting states. In particular, when constructing an alternating Büchi automaton for the negated specification with standard algorithms, accepting states are only induced by negated $\mathcal{U}$-operators. Such operators, however, usually also induce a self-loop in the respective state, thus yielding two runs that can visit the accepting state infinitely often. When dualizing the alternating Büchi automaton to obtain an alternating co-Büchi automaton for the initial specification φ (see Proposition 2.3), we thus obtain an automaton with a run that visits the rejecting state infinitely often; hence possibly influencing the acceptance. Therefore, rejecting states in automata constructed with standard algorithms seem to contain unnecessary rejecting states rarely. Furthermore, we experienced that alternating co-Büchi automata constructed with standard algorithms often ensure bad prefixes for delay-dominance: in Section 3.3, we discussed under which circumstances the bad prefix property is not satisfied and identify critical structures in alternating co-Büchi automata. The critical structures can most likely be encoded into an LTL formula in the sense that we obtain an alternating co-Büchi automaton with a critical structure when constructing it with standard algorithms. However, we experienced that for meaningful specifications, such critical structures rarely – if ever – exist in standard automata.

Simple optimizations such as removing rejecting states that do not lie in a cycle from the set of rejecting states of the alternating co-Büchi automaton $\mathcal{A}_\varphi$ have a positive impact on both the existence of delay-dominant strategies and on ensuring bad prefixes. Such states cannot be visited infinitely often; thus, removing them from the set of rejecting states does not alter the language. Nevertheless, rejecting states can enforce non-delay-dominance, and thus removing unnecessary rejecting states can result in the automaton ensuring bad prefixes and in more strategies being delay-dominant. For instance, it follows immediately with this optimization that, for safety properties, the parallel composition of delay-dominant strategies is delay-dominant; extending the result from Lemma 3.4 that for safety properties there always exists *some* alternating co-Büchi automaton that ensures bad prefixes for safety properties. Thus, we experienced that, for an alternating co-Büchi automaton $\mathcal{A}_\varphi$ constructed from an LTL formula φ with standard algorithms, it holds in many cases that (i) if φ allows for a remorsefree dominant strategy, then $\mathcal{A}_\varphi$ allows for an delay-dominant strategy, and (ii) the parallel composition of delay-dominance strategies for $\mathcal{A}_\varphi$ is delay-dominant as well. Therefore, the compositional synthesis algorithm presented in this section is indeed applicable for many LTL formulas and system architectures.

3.6. Summary

We have presented a new requirement for process strategies, delay-dominance, that allows a strategy to violate a given specification in certain situations. It is thus a notion of best effort. In contrast to the classical requirement of winning, delay-dominance can consequently be used for individually synthesizing strategies for the processes in a distributed system in many cases, enabling a simple compositional synthesis approach. Delay-dominance builds upon the concept of remorsefree dominance, where a strategy is allowed to violate the specification as long as no other strategy would have satisfied it in the same situation. However, remorsefree dominance is only compositional for safety properties. For liveness properties, the parallel composition of dominant strategies is not necessarily dominant. This restricts the use of compositional synthesis algorithms based on remorsefree dominance to safety specifications, which are often not expressive enough to state the system requirements. Delay-dominance, in contrast, is specifically designed to be compositional for more properties while maintaining desirable properties of remorsefree dominance such as that, if the specification is realizable, every remorsefree dominant or delay-dominant strategy is winning.

We have introduced a game-based definition of delay-dominance, which builds upon specifications given as alternating co-Büchi automata. Furthermore, we establish a bad prefix criterion on alternating co-Büchi automata such that, if the criterion is satisfied, compositionality of delay-dominance is guaranteed, both for safety and liveness properties. We have shown that every delay-dominant strategy is remorsefree dominant. Hence, for realizable system specifications, the parallel composition of delay-dominant strategies for all system processes is guaranteed to be winning for the entire system if the specification automaton satisfies the bad prefix criterion. Thus, delay-dominance is a suitable notion for compositional synthesis algorithms. We have, therefore, introduced a three-step automaton construction for recognizing delay-dominant strategies. The resulting universal co-Büchi automaton can immediately be used to synthesize delay-dominant strategies utilizing existing safraless synthesis approaches such as bounded synthesis. The automaton is of single-exponential size in the squared length of the initial LTL specification. Thus, synthesizing delay-dominant strategies is, as synthesis of winning and remorsefree dominant strategies, possible in 2EXPTIME. Synthesizing delay-dominant strategies for the individual system processes thus constitutes an efficient compositional synthesis algorithm for distributed systems.

Chapter 4

Assume-Guarantee Contracts for Distributed Synthesis

In the previous chapter, we have introduced delay-dominance as a best-effort notion for strategies, which weakens the classical strategy requirement of winning. We presented a compositional synthesis algorithm for distributed systems based on delay-dominant strategies. The algorithm utilized the *implicit assumption* induced by delay-dominance that other processes will not maliciously violate the shared goal, i.e., the specification for the entire system. While this allows for compositionally synthesizing strategies more often than with the naïve compositional distributed synthesis approach, which tries to synthesize winning strategies for the processes separately, it fails for systems with complex inter-process dependencies. Such systems often require more explicit assumptions on the process's concrete behavior.

In this chapter, we thus present a compositional synthesis algorithm for distributed systems, called *certifying synthesis*, that considers *explicit assumptions* on the behavior of the other system processes. Every system process provides a guarantee on its own behavior, a so-called *certificate*. The other system processes can then rely on the process to not deviate from its guaranteed behavior. The certificates define an *assume-guarantee contract* between the system processes. A process's strategy is then only required to realize the system specification if the other processes do not deviate from their guaranteed behavior. This allows for considering a system process independent from the other processes' strategies while accounting for the potential need for cooperation between the system processes via explicit assumptions. Certifying synthesis automatically derives both strategies and certificates for all system processes from a formal specification. It is an extension of bounded synthesis [FS13] that incorporates the additional search for certificates into the synthesis task for the process strategies.

In addition to enabling a compositional synthesis algorithm for distributed systems, synthesizing certificates has several benefits. First, observe that the assume-guarantee contract is formed with the processes' certificates rather than their strategies. Thus, while a process may rely on the other processes to not deviate from their certificates, it does not obtain any information about the other processes' strategies apart from their guaranteed behavior. Once the contract has been synthesized, particular process strategies can therefore be exchanged safely with other strategies as long as they still respect the contract, i.e., as long as the certificate still matches the new strategy. This enables *modularity of the system*. If requirements that do not affect the contract but only particular processes change, system strategies can be adapted flexibly without the need for synthesizing a solution for the entire system again.

Furthermore, the certificates accurately capture which information a system process requires about the other processes' behavior to be able to realize the specification. Certificates thus abstract from the behavior of other processes that is irrelevant from the considered process's point of view. Therefore, certifying synthesis allows for *recognizing the system's interconnections* by analyzing the certificates. Moreover, it enables a *local analysis* of the synthesized process strategies. When considering the strategy of an individual process, we do not need to take the other processes' entire strategies, which frequently contain irrelevant behavior, into account, but only their certificates. Both recognizing interconnections between system processes and the possibility of local analysis of process strategies greatly improve the understandability of the system and the derived strategies.

Lastly, bounded synthesis introduces a bound on the size of the desired strategy. This allows for finding size-optimal solutions. Certifying synthesis introduces, in addition to the size bound on the strategies in bounded synthesis, bounds on the sizes of the certificates. Consequently, certifying synthesis bounds the size of the *interface* between the system processes, which is shaped by the assume-guarantee contract. By starting with small certificate bounds and by only increasing them if the specification is unrealizable for the given bounds, our algorithm restricts synthesis to search for solutions with *small interfaces*, which are often preferred in practice. Thus, certifying synthesis guides the synthesis procedure toward desirable solutions.

We introduce two representations of certificates, as LTL formulas and as deterministic and complete finite-state transducers. We prove the soundness and completeness of our certifying synthesis algorithm for both of them. While LTL certificates have the advantage that they allow for nondeterminism, resulting in more compact certificates for certain specifications, certificates modeled with finite-state transducers are easier to integrate into existing synthesis algorithms. For the latter representation, we thus present a reduction of certifying synthesis to a SAT constraint-solving problem, which integrates certificates into the SAT constraint system for classical bounded synthesis [FFRT17]. The constraint system then immediately enables distributed synthesis using certificates.

Furthermore, we extend the representation of certificates with finite-state transducers with nondeterminism, thus taking advantage of the upside of LTL certificates. In particular, for certain specifications for which only knowledge about parts of the other processes' behavior is required, permitting nondeterminism results in significantly smaller certificates than when considering deterministic certificate transducers. We extend the SAT encoding of transducer-based certifying synthesis to allow for nondeterministic certificates. Moreover, we present an optimization of certifying synthesis that reduces the number of considered certificates by determining *relevant processes* for each system process. The certificates of non-relevant processes then do not need to be considered during the synthesis of the process' strategy. Soundness and completeness of certifying synthesis are preserved for all variants of certificates.

We implemented the certifying synthesis algorithm with certificates represented by finite-state transducers, both deterministic and nondeterministic ones, and compared its performance to an extension [Bau17] of the bounded synthesis tool BoSy [FFT17] to distributed systems as well as the compositional synthesis algorithm based on remorsefree dominant strategies [DF14]. The results clearly demonstrate the advantage of synthesizing certificates: if solutions with a small interface between the system processes exist, our algorithm significantly outperforms the other ones. Otherwise, the overhead of synthesizing certificates is small. Permitting nondeterminism can reduce the strategy and certificate sizes notably.

Publications and Structure. This chapter is based on work published in the proceedings of the *19th International Symposium on Automated Technology for Verification and Analysis* [FP21a] and in the *Innovations in Systems and Software Engineering Journal* [FP22a] as well as on the extended version [FP21b] of the former publication. The author of this thesis is the lead author of all three publications.

This chapter is structured as follows. After introducing a running example, which we use throughout the chapter, we present the certifying synthesis algorithm with certificates represented by LTL formulas in Section 4.2 and prove its soundness and completeness. In Section 4.3, we introduce certifying synthesis with certificates represented by deterministic and complete finite-state transducers. We show that incremental synthesis is also sound and complete for this type of certificate. In Section 4.4, we present how certificates represented by finite-state transducers can practically be synthesized alongside strategies for the processes. In particular, we introduce a SAT encoding of certifying synthesis. We present an optimization of certifying synthesis in Section 4.5 that identifies processes whose certificates are relevant to the considered process. Afterward, we extend certifying synthesis to certificates represented by nondeterministic and complete finite-state transducers in Section 4.6. We prove that soundness and completeness are preserved and present the necessary changes in the SAT constraints system with respect to certifying synthesis with deterministic transducers. Lastly, we provide an experimental evaluation of certifying synthesis in general and a comparison of the performance of certifying synthesis with deterministic and nondeterministic transducers.

4.1. Running Example

In this section, we illustrate the main concept of certifying synthesis with an example, which we will use throughout this chapter. Autonomous robots are a crucial component in the production line of many modern factories. The correctness of their implementation is essential; therefore, they are a natural target for synthesis. A factory with several robots can be inherently seen as a distributed system: each robot constitutes a process of the overall system.

We consider a factory with two autonomous robots that carry production parts from one machine to another. In the factory, there is a crossing that is used by both robots. The robots are required to prevent crashing into each other at the crossing. We formalize this with the following LTL formula:

$$\varphi_{no\_crash} = \square \neg \left(\left(atCrossing_1 \land \bigcirc go_1 \right) \land \left(atCrossing_2 \land \bigcirc go_2 \right) \right),$$

where, for $i \in \{1, 2\}$, variable $atCrossing_i$ is an input variable denoting that robot r_i arrived at the crossing, and where go_i is an output variable of robot r_i denoting that r_i moves one step ahead. Intuitively, φ_{no_crash} thus states that it should never be the case that both robots enter the crossing at the very same point in time. Furthermore, to prevent that both robots wait at the crossing forever and thus never arrive at the designated machines, both robots need to cross the intersection at some point in time after arriving there. This requirement is formalized in LTL for each robot r_i with $i \in \{1, 2\}$ as follows:

$$\varphi_{cross_i} = \square \left(atCrossing_i \rightarrow \bigcirc \diamondsuit go_i \right).$$

In addition to these crossing-related requirements, both robots have further individual objectives φ_{add_i}, which are specific to their area of application. For instance, they may capture which machines have to be approached by the robot r_i in which order.

None of the robots can realize $\varphi_{no\_crash} \wedge \varphi_{cross_1} \wedge \varphi_{cross_2}$ alone: since whether or not the other robot moves forward cannot be controlled, $\varphi_{cross_{3-i}}$ is not realizable for robot r_i. Furthermore, even if we restrict the specification to the parts concerning the output variables of r_i, i.e., if we only consider $\varphi_{no\_crash} \wedge \varphi_{cross_i}$ for r_i, no solution for individual synthesis tasks can be found. No matter when r_i enters the crossing after arriving there to ensure that φ_{cross_i} holds, the other robot r_{3-i} may enter the crossing at the exact same point in time, yielding a crash and thus violating φ_{no_crash}. While it is easy for humans to pinpoint this problem when only considering the requirements concerning the crossing, the additional objectives φ_{add_1} and φ_{add_2} of the robots may add much complexity to the specification, making it challenging to understand why the overall specification is not met.

However, if the robots commit to their behavior at crossings, individual solutions can be found. If, for instance, r_2 guarantees to give always priority to r_1 at crossings, a strategy for r_1 that enters crossings regardless of r_2 realizes $\varphi_{no\_crash} \wedge \varphi_{cross_1}$: since r_1 may assume that r_2 will not deviate from its certificate, it can rely on the fact that r_2 will not move forward if both robots are at the crossing. This ensures that φ_{no_crash} is satisfied no matter how r_1 behaves. However, a strategy for r_1 that *always* enters the crossing if it arrives there might prevent the existence of a strategy for robot r_2 that realizes φ_{cross_2}: in the, in fact, quite unrealistic, scenario that r_1 is, from some point in time on, always at the intersection again directly after crossing it – thus attempting to cross it immediately again in the other direction – r_2 would always give priority to r_1 and would never be able to cross the intersection itself. Therefore, robot r_1 needs to guarantee not to block the crossing in this manner. For instance, r_1 can ensure giving priority to r_2 at the crossing if it already waited there in the previous step. A strategy for r_2 that enters the crossing after it gave priority to r_1, for instance, then realizes both $\varphi_{no\_crash} \wedge \varphi_{cross_2}$. Note, however, that r_2 then cannot guarantee to *always* give priority to r_1 at the intersection as it does not in the step immediately after letting r_1 enter the crossing. Thus, we need to slightly adapt r_2's guarantee to this extent. Nevertheless, a strategy for r_1 that gives priority to r_2 if it already waited at the crossing in the previous step still realizes $\varphi_{no\_crash} \wedge \varphi_{cross_1}$ as long as r_2 then actually crosses the intersection as outlined above. The parallel composition of these strategies for the robots then indeed realizes the whole specification $\varphi_{no\_crash} \wedge \varphi_{cross_1} \wedge \varphi_{cross_2} \wedge \varphi_{add_1} \wedge \varphi_{add_2}$ as long as the strategies satisfy the additional requirements φ_{add_i} as well.

Furthermore, we then know that the robots solely interfere at crossings since the assumptions that the robots need to pose on the other robot's behavior to be able to realize the specification only concern the behavior at the crossing. Thus, the certificates, i.e., the guaranteed behavior of the robots, provide insight in the required communication of the robots and abstract away the irrelevant behavior, i.e., the behavior aside from crossings, of the other robot. Especially for large additional objectives φ_{add_i}, this significantly increases the understandability of why r_i's strategy realizes the specification. Moreover, the certificates form a contract of safe behavior at crossings: If φ_{add_i} changes since, e.g., the order in which the machines should be approached changes, it suffices to synthesize a new strategy for robot r_i. As long as r_i does not change its behavior at crossings, r_{3-i}'s strategy can be left unchanged.

4.2. Compositional Synthesis with Certificates

In this section, we present a sound and complete compositional synthesis algorithm for distributed systems. The main idea is, as in the naïve compositional synthesis algorithm from Algorithm 3.1, to synthesize strategies for the system processes separately. Note here that since we are considering arbitrary system architectures, processes are allowed to observe and, in particular, to react to the output variables of other processes. Therefore, similar to the previous chapter, we need to consider process strategies that can be represented by Moore transducers in compositional synthesis, as otherwise, it is not guaranteed that the parallel composition of the process strategies is complete (see Section 2.6.1).

Furthermore, we simplify the specification φ for the entire system when considering the individual processes. Intuitively, the simplified specification φ_i considered for an individual process $p_i \in P^-$ captures the parts of φ that affect p_i. Note that simplifying the specification is not necessary for our compositional synthesis algorithm. However, it can reduce the complexity of individual synthesis tasks. Simplifying specifications is not the main focus of this chapter; in fact, our algorithm can be used with any simplification fulfilling the above requirement. While there is work on obtaining small subspecifications – see, e.g. our specification decomposition algorithm for monolithic systems presented in Chapter 5 – we use an easy specification simplification in this chapter for simplicity:

Definition 4.1 (Specification Decomposition)**.**
Let $\varphi = \xi_1 \wedge \ldots \wedge \xi_k$ be an LTL formula over atomic propositions V with k conjuncts. The *specification decomposition of* φ is a vector $\langle \varphi_1, \ldots, \varphi_n \rangle$ of LTL formulas for the system processes $p_1, \ldots, p_n \in P^-$ such that $\varphi_i = \left\{ \xi_j \in \varphi \mid \mathit{prop}(\xi_j) \cap O_i \neq \emptyset \ \vee \ \mathit{prop}(\xi_j) \cap O^- = \emptyset \right\}$.

A specification decomposition $\langle \varphi_1, \ldots, \varphi_n \rangle$ of an LTL formula φ thus splits φ into n subformulas, one for each system process $p_i \in P^-$. Intuitively, a subformula φ_i contains all conjuncts of φ that contain output variables of the process p_i as well as all input-only conjuncts. Note that if a conjunct of φ contains output variables of two system processes $p_i, p_j \in P^-$ with $i \neq j$, then, intuitively, both the behavior of p_i and p_j may affect the satisfaction of the conjunct. Therefore, it is contained in both subformulas φ_i and φ_j. The satisfaction of input-only conjuncts, i.e., conjuncts that do not contain any output variables of system processes but only environment outputs, cannot be affected by any system process. Nevertheless, input-only conjuncts can prevent the realizability of the full LTL formula φ. Thus we need to take them into account when only considering the subformulas φ_i and not the full formula φ in compositional synthesis. While it suffices to add input-only conjuncts to a single subformula, we add them to *all* subformulas for simplicity of the definition of specification decomposition.

Example 4.1. Consider the two robots from the running example introduced in Section 4.1. Assume for simplicity that none of them has additional requirements φ_{add_i}, i.e., both robots are only required to safely cross the intersection when they arrive there. Hence, the full system specification is given by $\varphi = \varphi_{no\_crash} \land \varphi_{cross_1} \land \varphi_{cross_2}$. Recall that the only output variable of robot r_i is go_i, i.e., $O_i = \{go_i\}$. Thus, both φ_{no_crash} and φ_{cross_i} contain output variables of robot r_i, while $\varphi_{cross_{3-i}}$ does not. Therefore, we obtain the specification decomposition $\langle\varphi_1, \varphi_2\rangle$ for φ with $\varphi_i = \varphi_{no\_crash} \land \varphi_{cross_i}$ for $i \in \{1,2\}$. △

Although we decompose the specification, a system process p_i usually cannot guarantee the satisfaction of φ_i alone; rather, it depends on the cooperation of the other processes. For instance, robot r_1 from the running example from Section 4.1 cannot guarantee that no crash will occur when entering the crossing since r_2 can enter it at the very same point in time. Thus, the compositional synthesis approach presented in this chapter, called *certifying synthesis*, additionally derives a *guarantee on the behavior* of each system process, the so-called *certificate*. The certificate then provides essential information to the other system processes: if system process $p_i \in P^-$ commits to a certificate, the other processes can rely on p_i's strategy to not deviate from this behavior. In particular, the other processes' strategies only need to realize the specification as long as p_i sticks to the behavior formalized in its certificate. Hence, the certificates constitute an *assume-guarantee contract* (see, e.g., [CH07]) between the system processes. Therefore, in certifying synthesis, a system process is not required to react to *all* behaviors of the other processes but only to those that truly occur when the processes interact.

In this section, we represent the certificate of a system process $p_i \in P^-$ by an LTL formula ψ_i over atomic propositions V_i. The requirements on a strategy s_i for p_i are twofold: the strategy (i) may not deviate from p_i's certificate, and (ii) needs to realize the subformula φ_i if the other system processes stick to their certificates. Therefore, to ensure that (i) holds, we require s_i to realize the LTL formula ψ_i representing p_i's certificate. To establish (ii), we require s_i to realize the LTL formula $\Psi_i \rightarrow \varphi_i$, where $\Psi_i = \{\psi_j \mid p_j \in P^- \setminus \{p_i\}\}$, i.e., Ψ_i denotes the conjunction of the certificates of the other system processes.

Example 4.2. Consider the two robots from Section 4.1 and assume for simplicity that none of the robots has additional requirements φ_{add_i}. Thus, the full system specification is given by $\varphi = \varphi_{no\_crash} \land \varphi_{cross_1} \land \varphi_{cross_2}$ and, as outlined in Example 4.1, we obtain the specification decomposition $\langle\varphi_1, \varphi_2\rangle$ with $\varphi_i = \varphi_{no\_crash} \land \varphi_{cross_i}$ for all $i \in \{1,2\}$. An LTL certificate for robot r_2 could, for instance, be given by

$$\begin{aligned}\psi_2 = &\Box((\neg atCrossing_1 \lor \neg atCrossing_2) \rightarrow \bigcirc go_2)\\ &\land \Box((\neg(atCrossing_1 \land atCrossing_2) \land \bigcirc(atCrossing_1 \land atCrossing_2)) \rightarrow \bigcirc\bigcirc \neg go_2)\\ &\land \Box((atCrossing_1 \land atCrossing_2 \land \bigcirc(atCrossing_1 \land atCrossing_2)) \rightarrow \bigcirc\bigcirc go_2)\end{aligned}$$

Intuitively, it formalizes that r_2 always moves forward if one of the robots is not at the crossing. If however, both robots are at the crossing, and not both of them have been there in the previous step, then r_2 waits, thus giving priority to r_1. If both robots are at the contrast and both have been at the crossing in the previous step as well, then r_2 moves forward, ensuring to "take its turn" in crossing the intersection. While a strategy for robot r_1 that enters the crossing regardless of r_2 whenever not both robots have been at the crossing in the previous time step clearly does not realize φ_1, it realizes $\psi_2 \rightarrow \varphi_1$. △

Whether a strategy for a system process $p_i \in P^-$ is valid for the subformula φ_i thus does not only depend on φ_i but also on the certificates of the other system processes. Since the other processes' certificates range over their variables, the LTL formula $\Psi_i \rightarrow \varphi_i$ ranges over *all* variables V of the system and not only over p_i's variables V_i. Note that a strategy for p_i, however, is still defined for p_i's inputs and outputs, i.e., a computation of s_i lies in $(2^{V_i})^\omega$. Formally, we can now define certifying synthesis as follows:

Definition 4.2 (Certifying Synthesis with LTL certificates).
Let φ be an LTL formula over atomic propositions V with decomposition $\langle \varphi_1, \dots, \varphi_n \rangle$. Let $\mathcal{S} = \langle s_1, \dots, s_n \rangle$ and $\Psi = \langle \psi_1, \dots, \psi_n \rangle$ be vectors of strategies and LTL certificates, respectively, for the system processes $p_1 \dots p_n \in P^-$. Let $\Psi_i = \{\psi_j \mid p_j \in P^- \setminus \{p_i\}\}$. If $s_i \models \psi_i \wedge (\Psi_i \rightarrow \varphi_i)$ holds for all $p_i \in P^-$, then we say that $(\mathcal{S}, \Psi)$ realizes φ. *Certifying synthesis* for φ derives vectors $\mathcal{S}$ and Ψ such that $(\mathcal{S}, \Psi)$ realizes φ.

Classical algorithms for distributed synthesis directly search for strategies $s_1, \dots, s_n$ for the system processes such that $s_1 \mid\mid \dots \mid\mid s_n \models \varphi$ holds. Hence, they reason *globally* about the realization of the specification by the parallel composition of the synthesized strategies. Certifying synthesis, in contrast, reasons *locally* about the realization of the subformulas for the individual processes, i.e., without considering the composition of the strategies. Hence the strategies can be considered separately. This greatly improves the understandability of the synthesized solutions since it is possible to focus on a single process and its behavior.

Moreover, local reasoning as employed in certifying synthesis is sound and complete. Thus, if certifying synthesis derives a pair $(\mathcal{S}, \Psi)$ for an LTL specification φ, then the parallel composition of the strategies in $\mathcal{S}$ realizes φ. Furthermore, certifying synthesis derives a pair $(\mathcal{S}, \Psi)$ for all LTL specifications that are realizable in the considered architecture. Intuitively, soundness follows from the fact that every system process is required to realize its own certificate. Completeness is obtained since every strategy can serve as its own certificate. Formally:

Theorem 4.1. *Let φ be an LTL formula over atomic propositions V with decomposition $\langle \varphi_1, \dots, \varphi_n \rangle$. Let $\mathcal{S} = \langle s_1, \dots, s_n \rangle$ be a vector of strategies for the system processes. Then, there exists a vector $\Psi = \langle \psi_1, \dots, \psi_n \rangle$ of LTL certificates for the system processes such that $(\mathcal{S}, \Psi)$ realizes φ if, and only if, $s_1 \mid\mid \dots \mid\mid s_n \models \varphi$ holds.*

Proof. Let $\mathcal{T}_1, \dots \mathcal{T}_n$ be the deterministic and complete finite-state Moore transducers representing the strategies $s_1, \dots, s_n$. Let $\mathcal{T} = \mathcal{T}_1 \mid\mid \dots \mid\mid \mathcal{T}_n$ be their parallel compositions. Since all $\mathcal{T}_i$ are deterministic and complete Moore transducers and since the sets of output variables of different components are disjoint by definition of architectures, $\mathcal{T}$ is deterministic and complete by Lemma 2.1 as well. Hence, all traces of $\mathcal{T}$ are infinite and therefore $\mathit{Traces}(\mathcal{T}) = \{\sigma \in (2^V)^\omega \mid \forall p_i \in P^-.\ \sigma \cap V_i \in \mathit{Traces}(\mathcal{T}_i)\}$ follows with Lemma 2.2.

First, suppose that there exists a vector $\Psi = \langle \psi_1, \dots, \psi_n \rangle$ of LTL certificates for the system processes such that $(\mathcal{S}, \Psi)$ realizes φ. For process $p_j \in P^-$, let $\Psi_j = \{\psi_i \mid p_i \in P^- \setminus \{p_j\}\}$. Let $\sigma \in \mathit{Traces}(\mathcal{T})$ be a trace of $\mathcal{T}$. Then, as shown above, $\sigma \cap V_i \in \mathit{Traces}(\mathcal{T}_i)$ holds for all system processes $p_i \in P^-$. By assumption, $(\mathcal{S}, \Psi)$ realizes φ. Therefore, for all $p_i \in P^-$, we have $s_i \models \psi_i \land (\Psi_i \rightarrow \varphi_i)$ and hence $\mathit{Traces}(\mathcal{T}_i) \cup (V \setminus V_i) \subseteq \mathcal{L}(\psi_i \land (\Psi_i \rightarrow \varphi_i))$ holds. Thus, $(\sigma \cap V_i) \cup \sigma' \in \mathcal{L}(\psi_i \land (\Psi_i \rightarrow \varphi_i))$ follows for all $\sigma' \in V \setminus V_i$. Hence, in particular, $\sigma \in \mathcal{L}(\psi_i \land (\Psi_i \rightarrow \varphi_i))$ holds for all $p_i \in P^-$ and therefore, by definition of conjunction, we have $\sigma \in \mathcal{L}(\bigwedge_{i=1}^{n}(\psi_i \land (\Psi_i \rightarrow \varphi_i)))$. Thus both $\sigma \in \mathcal{L}(\bigwedge_{i=1}^{n} \psi_i)$ and $\sigma \in \mathcal{L}(\bigwedge_{i=1}^{n} \Psi_i \rightarrow \varphi_i)$ hold and hence $\sigma \in \mathcal{L}(\bigwedge_{i=1}^{n} \varphi_i)$ follows with the definition of Ψ_i and the semantics of implication. By definition of specification decomposition, we have $\bigwedge_{i=1}^{n} \varphi_i = \varphi$ and therefore we obtain $\sigma \in \mathcal{L}(\varphi)$. Since we chose the trace $\sigma \in \mathit{Traces}(\mathcal{T})$ of $\mathcal{T}$ arbitrarily, $\mathit{Traces}(\mathcal{T}) \subseteq \mathcal{L}(\varphi)$ follows. Thus, by definition of $\mathcal{T}$, we have $s_1 \mid\mid \dots \mid\mid s_n \models \varphi$.

Second, suppose that $s_1 \mid\mid \dots \mid\mid s_n \models \varphi$ holds. We construct LTL certificates $\psi_1, \dots, \psi_n$ as follows: ψ_i describes exactly the behavior of s_i for the variables in V_i, i.e., $\mathcal{L}(\psi_i) = \mathit{Traces}(\mathcal{T}_i)$ holds. Since $\mathcal{T}_i$ has, by construction, only finitely many states, such an LTL formula ψ_i can always be constructed by encoding the transducer. Let $\Psi = \langle \psi_1, \dots, \psi_n \rangle$ and let $\Psi_i = \{\psi_j \mid p_j \in P^- \setminus \{p_i\}\}$. It remains to show that $(\mathcal{S}, \Psi)$ realizes φ, i.e., that $s_i \models \psi_i \land (\Psi_i \rightarrow \varphi_i)$ holds and thus that we have $\mathit{Traces}(\mathcal{T}_i) \cup (V \setminus V_i) \subseteq \mathcal{L}(\psi_i \land (\Psi_i \rightarrow \varphi_i))$ for all $p_i \in P^-$. Let $p_i \in P^-$ be some system process. Let $\sigma \in \mathit{Traces}(\mathcal{T}_i) \cup (V \setminus V_i)$. By construction of the LTL certificates, we clearly have $\sigma \in \{\rho \in (2^V)^\omega \mid \rho \cap V_i \in \mathcal{L}(\psi_i)\}$. If $\sigma \models \neg\Psi_i$ holds, then $\sigma \in \mathcal{L}(\Psi_i \rightarrow \varphi_i)$ follows immediately with the semantics of implication and since $\Psi_i \rightarrow \varphi_i$ is an LTL formula over atomic propositions V. Thus, since $\sigma \in \{\rho \in (2^V)^\omega \mid \rho \cap V_i \in \mathcal{L}(\psi_i)\}$ holds as shown above, we obtain $\sigma \in \mathcal{L}(\psi_i \land (\Psi_i \rightarrow \varphi_i))$ with the semantics of conjunction. Otherwise, i.e., if $\sigma \models \Psi_i$ holds, then $\sigma \in \mathcal{L}(\bigwedge_{i=1}^{n} \psi_i)$ follows with the definition of Ψ_i and since we have $\sigma \in \{\rho \in (2^V)^\omega \mid \rho \cap V_i \in \mathcal{L}(\psi_i)\}$. Thus, $\sigma \cap V_i \in \mathit{Traces}(\mathcal{T}_i)$ holds for all system processes $p_i \in P^-$ by construction of the LTL certificates. Therefore, $\sigma \in \mathit{Traces}(\mathcal{T})$ follows since we have $\mathit{Traces}(\mathcal{T}) = \{\sigma \in (2^V)^\omega \mid \forall p_i \in P^-.\ \sigma \cap V_i \in \mathit{Traces}(\mathcal{T}_i)\}$ as shown above. By assumption, $s_1 \mid\mid \dots \mid\mid s_n \models \varphi$ holds and thus, by definition of $\mathcal{T}$, we have $\mathit{Traces}(\mathcal{T}) \subseteq \mathcal{L}(\varphi)$. Therefore, by definition of specification decomposition and by the semantics of conjunction, $\mathit{Traces}(\mathcal{T}) \subseteq \mathcal{L}(\varphi_i)$ holds as well. Thus, $\sigma \in \mathcal{L}(\varphi_i)$ follows and hence, by the semantics of implication, $\sigma \in \mathcal{L}(\Psi_i \rightarrow \varphi_i)$ holds as well. Since $\sigma \in \{\rho \in (2^V)^\omega \mid \rho \cap V_i \in \mathcal{L}(\psi_i)\}$ holds as shown above, we obtain $\sigma \in \mathcal{L}(\psi_i \land (\Psi_i \rightarrow \varphi_i))$ with the semantics of conjunction. Since we chose $\sigma \in \mathit{Traces}(\mathcal{T}_i) \cup (V \setminus V_i)$ arbitrarily, $\mathit{Traces}(\mathcal{T}_i) \cup (V \setminus V_i) \subseteq \mathcal{L}(\psi_i \land (\Psi_i \rightarrow \varphi_i))$ follows. Hence, $s_i \models \psi_i \land (\Psi_i \rightarrow \varphi_i)$ holds for all $p_i \in P^-$ and therefore $(\mathcal{S}, \Psi)$ realizes φ. □

Certifying synthesis thus enables modularity and increases the understandability of the system due to local reasoning while ensuring finding solutions for all specifications that are realizable in the architecture. Moreover, the parallel composition of the synthesized strategies serves as a correct solution for the entire system.

There are several quality measures for certificates, for instance, their size. We focus on certificates that are *easy to synthesize* in the sense that certifying synthesis can be integrated into existing synthesis algorithms first. Therefore, in the subsequent section, we study how to model certificates with finite-state transducers instead of LTL formulas.

4.3. Synthesis with Deterministic Certificates

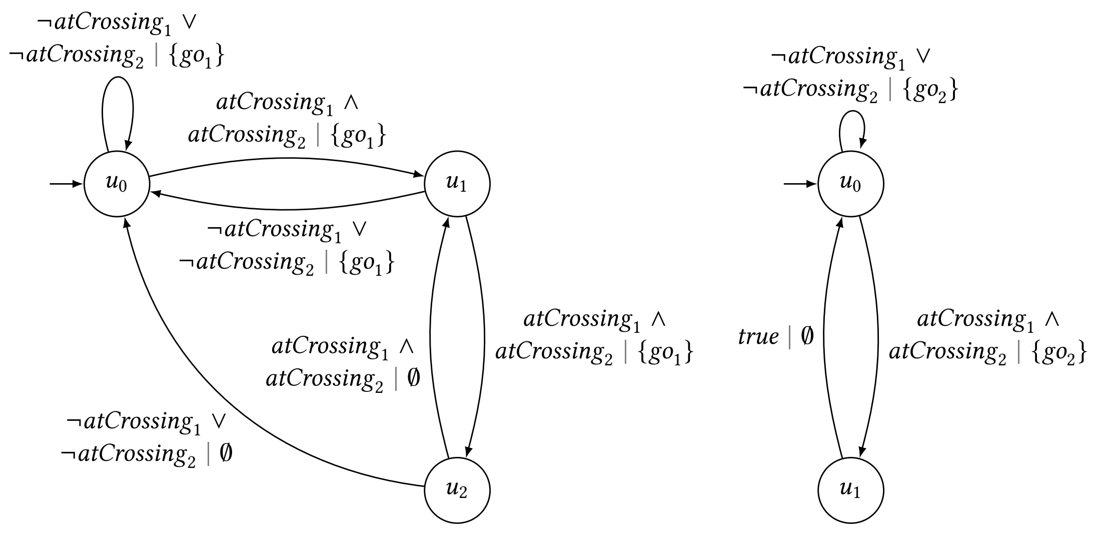
In the previous section, we considered certificates in certifying synthesis to be LTL formulas that describe the guaranteed behavior of the individual system processes. In the following, in contrast, we focus on certificates that allow for simple integration of certifying synthesis into existing synthesis algorithms and frameworks. Here, we focus on *constraint-based bounded synthesis* [FS13, FFRT17] as, for instance, implemented in the tool BoSy [FFT17]. Therefore, in this section, we introduce certifying synthesis with certificates represented by deterministic finite-state transducers. First, we present how certificates can be modeled with transducers. Afterward, we formulate certifying synthesis and, in particular, the satisfaction of a specification in the presence of certificates represented by finite-state transducers. Lastly, we show soundness and completeness of this variant of certifying synthesis.

4.3.1. Modeling Certificates

We model the certificate of a system process $p_i \in P^-$ as a deterministic and complete finite-state Moore transducer $\mathcal{T}_i^G$, called *guarantee transducer* (GT), over input variables I_i and *guarantee output variables* $O_i^G \subseteq O_i$. Only considering a subset of O_i as output variables of the guarantee transducers allows the certificate to abstract from outputs of p_i whose valuation is irrelevant for all other processes. In the following, we assume the guarantee output variables of p_i to be both an output of p_i and an input of some other process, i.e., we define $O_i^G := O_i \cap I^-$. Intuitively, a variable $v \in O_i \setminus O_i^G$, which is an output of p_i but not a guarantee output, cannot be observed by any other system process. Thus, a guarantee on its behavior does not influence any process and hence it can be omitted from the outputs of the guarantee transduce $\mathcal{T}_i^G$. Since a guarantee transducer $\mathcal{T}_i^G$ is both deterministic and complete by construction, it produces exactly one infinite trace for every input sequence $\gamma \in (2^{I_i})^\omega$, i.e., $|\mathit{Traces}(\mathcal{T}_i^G, \gamma)| = 1$ holds. Slightly overloading notation, we call this single trace produced by $\mathcal{T}_i^G$ on input γ the *computation* of $\mathcal{T}_i^G$ on γ, also denoted $\mathit{comp}(\mathcal{T}_i^G, \gamma)$.

Example 4.3. Consider the robots from the running example introduced in Section 4.1. Guarantee transducers $\mathcal{T}_1^G$ and $\mathcal{T}_2^G$ for the robots r_1 and r_2 are depicted in Figure 4.1.

Intuitively, $\mathcal{T}_1^G$ stays in state u_0 until both robots arrive at the crossing, always outputting go_1. If both robots arrive at the crossing, $\mathcal{T}_1^G$ moves to u_1, ensuring that r_1 can make use of its priority in the next step and move forward. If at most one robot arrives at the crossing afterward, $\mathcal{T}_1^G$ transitions back to u_0. Otherwise, it moves to u_2, ensuring that r_1 does not block the crossing but gives r_2 the possibility of crossing the intersection. Thus, $\mathcal{T}_1^G$ keeps track in its state whether at most one robot arrives at the crossing (state u_0), both robots arrive together at the crossing, while not both of them have been at the crossing in the previous step (state u_1), or whether


$\neg atCrossing_1 \lor \neg atCrossing_2 \mid \{go_1\}$
$atCrossing_1 \land atCrossing_2 \mid \{go_1\}$
u_0
u_1
$\neg atCrossing_1 \lor \neg atCrossing_2 \mid \{go_1\}$
$atCrossing_1 \land atCrossing_2 \mid \{go_1\}$
$atCrossing_1 \land atCrossing_2 \mid \emptyset$
$\neg atCrossing_1 \lor \neg atCrossing_2 \mid \emptyset$
u_2
$\neg atCrossing_1 \lor \neg atCrossing_2 \mid \{go_2\}$
u_0
$true \mid \emptyset$
$atCrossing_1 \land atCrossing_2 \mid \{go_2\}$
u_1

(a) Guarantee transducer $\mathcal{T}_1^G$ (b) Guarantee transducer $\mathcal{T}_2^G$

Figure 4.1.: Guarantee transducers for the robots r_1 and r_2 from the running example.

both robots arrive at the crossing in (at least) two consecutive time steps and we are currently in an even one of these time steps (state u_2).

Similar to $\mathcal{T}_1^G$, the guarantee transducer $\mathcal{T}_2^G$ intuitively stays in state u_0 until both robots arrive at the crossing, always outputting go_2. If both robots arrive at the crossing, $\mathcal{T}_2^G$ moves to u_1, ensuring that r_2 does not move forward in the next step to grant r_1 priority. Afterward, irrespective of the position of the robots, $\mathcal{T}_1^G$ moves back to state u_0, ensuring that r_2 outputs go_2 in the next step to take its turn in crossing the intersection. Thus, $\mathcal{T}_2^G$ keeps track of whether at most one robot arrives at the crossing (state u_0) or both robots arrive together at the crossing, while not both of them have been at the crossing in the previous step (state u_2). △

Since both strategy transducers and guarantee transducers – and thus all transducers we are considering in this chapter – have Moore semantics, we omit the input from the labeling relation in the remainder of this chapter. That is, slightly overloading notation, we assume the labeling relation ℓ of a $(2^I, 2^O)$-transducer $\mathcal{T} = (T, T_0, \tau, \ell)$ to be of type $\ell : T \times 2^O$.

In the next section, we present how guarantee transducers can be utilized instead of LTL certificates in certifying synthesis while still ensuring soundness and completeness.

4.3.2. Certifying Synthesis with Guarantee Transducers

In certifying synthesis, it is crucial that a strategy only needs to realize the specification if the other processes do not deviate from their certificates. For certificates modeled as LTL formulas, we use an implication in the local objective, which is again an LTL formula, to model this (see Section 4.2). When representing certificates as finite-state transducers, however, it is no longer possible to easily integrate the satisfaction of the other processes' certificates into the local LTL specification without encoding the certificate as LTL formula and thus losing the benefit of simple integrability into existing synthesis frameworks. Instead, we formalize that a strategy only needs to realize the specification if the other processes do not deviate from their certificates by slightly altering the notion of satisfaction of an LTL formula. Intuitively, a strategy realizes an LTL formula φ if each of its computations either satisfies φ or could not occur if the other processes stick to their certificates. Consequently, we need to identify whether or not a sequence matches the other processes' certificates in the sense that it could occur when the processes interact if none of the processes deviates from its certificate. We formalize this with so-called *valid computations*:

Definition 4.3 (Valid Computation and Valid History)**.**
Let $\mathcal{P} \subseteq P^-$ be a finite set of system processes. Let $\mathcal{G}$ be a finite set of guarantee transducers, one for each of the processes in $\mathcal{P}$. An infinite sequence $\sigma \in (2^V)^\omega$ is called *valid computation* for $\mathcal{G}$ if, and only if $\sigma \cap O_i^G = comp(\mathcal{T}_i^G, \sigma \cap I_i) \cap O_i^G$ holds for all $\mathcal{T}_i^G \in \mathcal{G}$. The set of valid computations for $\mathcal{G}$ is denoted with $\mathcal{V}_\mathcal{G}$. A finite prefix $\rho \in (2^V)^*$ of length $k \geq 0$ of some valid computation $\sigma \in \mathcal{V}_\mathcal{G}$ is called *valid history* of length k for $\mathcal{G}$. The set of all valid histories of length k for $\mathcal{G}$ is denoted with $\mathcal{H}_\mathcal{G}^k$.

Intuitively, a valid computation for a set $\mathcal{G}$ of guarantee transducers is an infinite sequence that is a computation of all guarantee transducers in $\mathcal{G}$. Thus, a valid computation can be produced by all guarantee transducers in $\mathcal{G}$, and therefore it can be produced by their parallel composition. Consequently, a valid computation is a sequence that can occur in the interplay of all processes whose guarantee transducers are contained in $\mathcal{G}$ as long as these processes do not deviate from their guaranteed behavior. A valid history for $\mathcal{G}$ is then a finite prefix of a computation of the parallel composition of the guarantee transducers in $\mathcal{G}$.

Example 4.4. Consider the robots from the running example introduced in Section 4.1 and the guarantee transducers depicted in Figure 4.1. As an example for valid computations, consider robot r_2 with its guarantee transducer $\mathcal{T}_2^G$ from Figure 4.1b. Let $\gamma \in (2^{I_2})^\omega$ be some infinite input sequence or r_2 with $\gamma_k \neq \{atCrossing_1, atCrossing_2\}$ for some point in time $k \geq 0$, denoting that at most one of the robots arrives at the crossing at point in time k. Irrespective of the nature of the valuations of the input variables of p_2 at the remaining points in time $k' \geq 0$ with $k' \neq k$, the path $\pi \in Paths(\mathcal{T}_2^G, \gamma)$ of $\mathcal{T}_2^G$ on input sequence γ visits state u_0 at point in time $k+1$, i.e., we have $\pi_{k+1} = (u_0, \{go_2\})$. Therefore, by definition of traces, $\sigma_{k+1} \cap O_2 = \{go_2\}$ holds for all $\sigma \in Traces(\mathcal{T}_2^G, \gamma)$. Furthermore, since $O_i \cap I^-\{go_2\}$ holds, go_2 is not only an output but also a guarantee output of robot r_2. Thus, we have $\sigma_{k+1} \cap O_2^G = \{go_2\}$ for all $\sigma \in Traces(\mathcal{T}_2^G, \gamma)$ as well. Therefore, every infinite sequence $\rho \in (2^V)^\omega$ with either $atCrossing_1 \notin \rho_k$ or $atCrossing_2 \notin \rho_k$ but $go_2 \notin \rho_{k+1}$ for some point in time $k \geq 0$ is no valid computation for $\mathcal{G} = \{\mathcal{T}_2^G\}$. △

Since the notion of valid computations determines whether or not a particular infinite sequence can occur during the interaction of the processes whose guarantee transducers are contained in the considered set $\mathcal{G}$ as long as these processes do not deviate from their certificates, we use valid computations to define the slightly altered version of satisfaction which is required for certifying synthesis with guarantee transducers: a strategy *locally realizes* an LTL formula for a finite set $\mathcal{G}$ of guarantee transducers if, for all input sequences, its computation either classically satisfies the LTL formula or its computation is not valid and thus does not match the guarantee transducers in $\mathcal{G}$. Formally:

Definition 4.4 (Local Satisfaction and Local Realization)**.**
Let $p_i \in P^-$ be a system process and let $\mathcal{P} \subseteq P^- \setminus \{p_i\}$ be a set of other system processes. Let $\mathcal{G}$ be a set of guarantee transducers, one for each of the processes in $\mathcal{P}$. Let φ_i be an LTL formula over atomic propositions V_i. An infinite sequence $\sigma \in (2^V)^\omega$ *locally satisfies* φ_i with respect to $\mathcal{G}$, denoted $\sigma \models_{\mathcal{G}} \varphi_i$, if, and only if, either $\sigma \models \varphi_i$ or $\sigma \notin \mathcal{V}_{\mathcal{G}}$ holds. A strategy s_i for p_i then *locally realizes* φ_i with respect to $\mathcal{G}$, denoted $s_i \models_{\mathcal{G}} \varphi_i$, if, and only if, $comp(s_i, \gamma) \cup \gamma' \models_{\mathcal{G}} \varphi_i$ holds for all $\gamma \in (2^{I_i})^\omega$ and all $\gamma' \in (2^{V \setminus V_i})^\omega$.

Intuitively, requiring a strategy to locally realize an LTL formula φ_i with respect to a set $\mathcal{G}$ of guarantee transducers thus encodes the classical satisfaction of the local objective $\Psi \rightarrow \varphi_i$, where Ψ is an LTL formula encoding the computations of all guarantee transducers in $\mathcal{G}$, as used in certifying synthesis with LTL certificates. A sequence $\sigma \in (2^V)^\omega$ satisfies $\Psi \rightarrow \varphi_i$ if it either satisfies φ_i or violates Ψ. If the former is the case, then σ clearly also locally satisfies φ_i. If the latter is the case, then σ does not match the guaranteed behavior defined in Ψ, and thus, by construction of Ψ, it does not match the computations of all guarantee transducers in $\mathcal{G}$. Thus, σ is then no valid computation for $\mathcal{G}$, and therefore it locally satisfies φ_i as well.

Example 4.5. Consider the robots from the running example presented in Section 4.1. Furthermore, consider the guarantee transducer $\mathcal{T}_2^G$ for robot r_2 depicted in Figure 4.1b. If r_2 does not deviate from its guaranteed behavior defined by $\mathcal{T}_2^G$, then r_1 can enter the crossing regardless of r_2 without risking a crash whenever both robots arrive at the crossing while at least one of them was not at the crossing in the previous time step. Such a strategy s_1 for r_1 can, for instance, be given by the same transducer as r_1's guarantee transducer depicted in Figure 4.1a. In the following, we call this transducer $\mathcal{T}_1$.

A computation of s_1 on some input sequence $\gamma \in (2^{I_i})^\omega$ only contains go_1 at some point in time $k \geq 0$ if the corresponding path $\pi \in Paths(\mathcal{T}_1, \gamma)$ of $\mathcal{T}_1$ is in state u_0 or u_1 at point in time k. The path π is only in state u_0 at point in time k if either $k = 0$ holds or if at most one of the robots arrives at the crossing at point in time $k - 1$, i.e., if we have either $atCrossing_1 \notin \gamma_{k-1}$ or $atCrossing_2 \notin \gamma_{k-1}$. Furthermore, π is only in state u_1 at point in time k if both robots arrive at the crossing at point in time $k - 1$ and if $k - 1$ is an odd position in the current sequence of consecutive time steps at which both robots arrive at the crossing. Clearly, no crash can happen whenever $\mathcal{T}_1$ is in state u_0: it neither violates φ_{no_crash} if both robots move forward in the first time step nor if at most one of the robots arrived at the crossing at the previous point in time. Whenever $\mathcal{T}_1$ is in state u_1, in contrast, a crash can potentially happen if r_2 moves forward in the very same time step, violating φ_{no_crash}. However, a trace of r_2's guarantee transducer $\mathcal{T}_2^G$ does not contain go_2 at a point in time $k > 0$ if both robots arrive at the crossing at point in time $k - 1$ and if $k - 1$ is an odd position in the current sequence of consecutive time steps at which both robots arrive at the crossing. Hence, an infinite sequence that contains both go_1 and go_2 at some point in time $k \geq 0$ is no valid computation with respect to $\mathcal{G} = \{\mathcal{T}_2^G\}$. and therefore s_1 locally realizes φ_{no_crash} with respect to $\mathcal{G} = \{\mathcal{T}_2^G\}$, i.e., we have $s_1 \models_{\mathcal{G}} \varphi_{no\_crash}$.

Furthermore, every computation of s_1 classically satisfies φ_{cross_1}, i.e., $s_1 \models \varphi_{cross_1}$ holds: the transducer $\mathcal{T}_1$ leaves the only state in which it does not output go_1, i.e., state u_2, immediately after arriving there, irrespective of the input sequence. That is, no path of $\mathcal{T}_1$ can loop indefinitely in u_2, and therefore the states in which $\mathcal{T}_1$ outputs go_1 are visited infinitely often for every input sequence. Hence, s_1 outputs go_1 infinitely often for every input sequence and, thus, in particular, for every input sequence that contains $atCrossing_1$ at some point in time. △

Since local satisfaction allows for formalizing that a strategy only needs to realize the specification if the other system processes do not deviate from their certificates, we employ local satisfaction and local realization for defining certifying synthesis with certificates modeled with guarantee transducers. However, recall that the requirements for strategies in certifying synthesis are twofold. Additionally, strategies are not allowed to deviate from their own certificate. When representing certificates with LTL formulas, we achieved this by requiring the strategy to *realize* the LTL certificate (see Section 4.2). When considering guarantee transducers, in contrast, we utilize transducer *simulation* for Moore transducers instead:

Definition 4.5 (Transducer Simulation)**.**
Let I, O_1, and O_2 be finite sets of input and output variables with $I \cap O_1 = \emptyset$, $I \cap O_2 = \emptyset$, and $O_1 \subseteq O_2$. Let $\mathcal{T}_1 = (T_1, T_{1,0}, \tau_1, \ell_1)$ and $\mathcal{T}_2 = (T_2, T_{2,0}, \tau_2, \ell_2)$ be a finite-state $(2^I, 2^{O_1})$-transducer and a finite-state $(2^I, 2^{O_2})$-transducer, respectively. Then, $\mathcal{T}_1$ *simulates* $\mathcal{T}_2$, denoted $\mathcal{T}_2 \preceq \mathcal{T}_1$, if, and only if, there exists a simulation relation $R : T_2 \times T_1$ with

- $(t_{2,0}, t_{1,0}) \in R$ for all $t_{2,0} \in T_{2,0}$ and all $t_{1,0} \in T_{1,0}$,
- for all $(t_1, t_2) \in R$, we have $\{o \mid (t_1, o) \in \ell_1\} = \{o \cap O_1 \mid (t_2, o) \in \ell_2\}$ and, for all $\iota \in 2^I$ and all $t_2' \in T_2$, if $(t_2, \iota, t_2') \in \tau_2$ holds, then there exists some $t_1' \in T_1$ such that both $(t_1, \iota, t_1') \in \tau_1$ and $(t_2', t_1') \in R$ hold.

Intuitively, a finite-state transducer $\mathcal{T}_1$ thus simulates a finite-state transducer $\mathcal{T}_2$ if all traces of $\mathcal{T}_2$ are, restricted to the variables $I \cup O_1$ of $\mathcal{T}_1$, also traces of $\mathcal{T}_1$. In the following, we show that this intuition indeed holds:

Proposition 4.1. *Let I, O_1, and O_2 be finite sets of input and output variables with $I \cap O_1 = \emptyset$, $I \cap O_2 = \emptyset$, and $O_1 \subseteq O_2$. Let $\mathcal{T}_1$ be a finite-state $(2^I, 2^{O_1})$-transducer and let $\mathcal{T}_2$ be a finite-state $(2^I, 2^{O_2})$-transducer. If $\mathcal{T}_2 \preceq \mathcal{T}_1$ holds, then $\sigma \cap (I \cup O_1) \in Traces(\mathcal{T}_1)$ holds for all $\sigma \in Traces(\mathcal{T}_2)$.*

Proof. Let $\mathcal{T}_1 = (T_1, t_{1,0}, \tau_1, \ell_1)$ and let $\mathcal{T}_2 = (T_2, t_{2,0}, \tau_2, \ell_2)$. Assume that $\mathcal{T}_2 \preceq \mathcal{T}_1$ holds. Let $\sigma \in Traces(\mathcal{T}_2)$ be a trace of $\mathcal{T}_2$ and let $\pi \in Paths(\mathcal{T}_2, \sigma \cap I)$ be the corresponding path. Since $\mathcal{T}_2 \preceq \mathcal{T}_1$ holds by assumption, there exists a simulation relation R that satisfies the properties of transducer simulation defined in Definition 4.5.

By definition of paths, we have $(\#_1(\pi_k), \sigma_k \cap I, \#_1(\pi_{k+1})) \in \tau_2$ and $(\#_1(\pi_k), \sigma_k \cap I, \#_2(\pi_k)) \in \ell_2$ for every point in time $k \geq 0$. Hence, by definition of the simulation relation R, there exists an infinite sequence $\rho \in (T_1 \times 2^I \times 2^{O_1} \times T_1)^\omega$ such that $(\#_1(\pi_k), \#_1(\rho_k)) \in R$, $\#_2(\rho_k) = \sigma_k \cap I$, $\#_3(\rho_k) = (\sigma_k \cap O_2) \cap O_1$, and $\#_4(\rho_k) = \#_1(\rho_{k+1})$ as well as $\rho_k \in \tau_1$ holds for all points in time $k \geq 0$. Let $\pi' \in (T_1 \times 2^{O_1})^\omega$ be the infinite sequence such that $\pi' = (\#_1(\rho_k), \#_3(\rho_k))$ for all

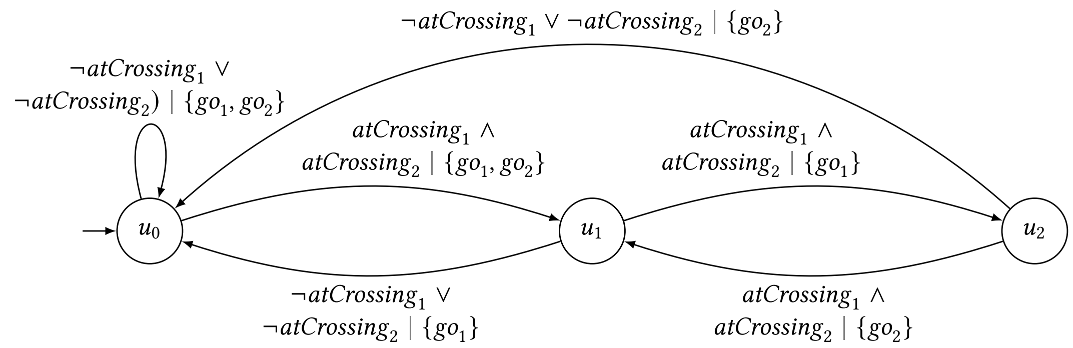

$\neg atCrossing_1 \vee \neg atCrossing_2 \mid \{go_2\}$
$\neg atCrossing_1 \vee \neg atCrossing_2) \mid \{go_1, go_2\}$
$atCrossing_1 \wedge atCrossing_2 \mid \{go_1, go_2\}$
$atCrossing_1 \wedge atCrossing_2 \mid \{go_1\}$
u_0
u_1
u_2
$\neg atCrossing_1 \vee \neg atCrossing_2 \mid \{go_1\}$
$atCrossing_1 \wedge atCrossing_2 \mid \{go_2\}$

Figure 4.2.: Parallel composition of the strategies of the robots from the running example.

points in time $k \geq 0$. By construction of ρ, we clearly have $\pi' \in \mathit{Paths}(\mathcal{T}_1, \sigma \cap I)$. Furthermore, there exists a trace $\sigma' \in \mathit{Traces}(\mathcal{T}_1, \sigma \cap I)$ such that $\sigma' = (\sigma \cap I) \cup ((\sigma \cap O_2) \cap O_1)$ holds. Since $O_1 \subseteq O_2$ holds by assumption, we have $(\sigma \cap O_2) \cap O_1 = \sigma \cap O_1$ and thus $\sigma' = \sigma \cap (I \cup O_1)$ follows. Therefore, $\sigma \cap (I \cup O_1) \in \mathit{Traces}(\mathcal{T}_1)$ holds. □

Hence, requiring that a strategy transducer $\mathcal{T}_i$ for system process $p_i \in P^-$ is simulated by the guarantee transducer $\mathcal{T}_i^G$ of p_i ensures that every trace produced by the strategy of p_i is also produced by p_i's certificate. That is, intuitively, p_i's strategy cannot perform actions that are not captured by the certificate of p_i and therefore the strategy of a process cannot deviate from the process's own certificate.

With local realization and transducer simulation, we have laid the foundations for utilizing deterministic finite-state Moore transducers for representing certificates. In the following, we thus lift certifying synthesis from certificates given as LTL formulas to certificates represented by deterministic finite-state guarantee transducers with Moore semantics. First, we formally define certifying synthesis with guarantee transducers:

Definition 4.6 (Certifying Synthesis with Guarantee Transducers)**.**
Let φ be an LTL formula over atomic propositions V with decomposition $\langle \varphi_1, \ldots, \varphi_n \rangle$. Let $\mathcal{S} = \langle s_1, \ldots, s_n \rangle$ and $\mathcal{G} = \langle \mathcal{T}_1^G, \ldots, \mathcal{T}_n^G \rangle$ be vectors of strategies and guarantee transducers, respectively, for the system processes. For $p_j \in P^-$, let $\mathcal{G}_j = \{\mathcal{T}_i^G \mid p_i \in P^- \setminus \{p_j\}\}$. If $s_i \models_{\mathcal{G}_i} \varphi_i$ and $\mathcal{T}_i \preceq \mathcal{T}_i^G$, where $\mathcal{T}_i$ is the deterministic and complete finite-state Moore transducer representing s_i, hold for all $p_i \in P^-$, then we say that $(\mathcal{S}, \mathcal{G})$ realizes φ. *Certifying synthesis* for φ derives vectors $\mathcal{S}$ and $\mathcal{G}$ such that $(\mathcal{S}, \mathcal{G})$ realizes φ.

Certifying synthesis with guarantee transducers is thus, in general, similar to certifying synthesis with LTL certificates. However, it seeks strategies and guarantee transducers instead of strategies and LTL certificates. Moreover, it models the requirements that a strategy must not deviate from its certificate and that a strategy only needs to satisfy the specification if the other processes stick to their certificates with transducer simulation and local satisfaction rather than with incorporating them into the LTL formula defining the processes objective.

Example 4.6. Consider the robots r_1 and r_2 from the running example from Section 4.1 and their guarantee transducers $\mathcal{T}_1^G$ and $\mathcal{T}_2^G$ depicted in Figure 4.1. Recall that the guarantee transducers can also be interpreted as strategy transducers $\mathcal{T}_1$ and $\mathcal{T}_2$ for the two robots. Clearly, $\mathcal{T}_i \preceq \mathcal{T}_i^G$ holds for all $i \in \{1,2\}$. As outlined in Example 4.5, we also have $\mathcal{T}_1 \models_{\{\mathcal{T}_2^G\}} \varphi_{no\_crash} \land \varphi_{cross_1}$. Similarly, $\mathcal{T}_2 \models_{\{\mathcal{T}_1^G\}} \varphi_{no\_crash} \land \varphi_{cross_2}$ follows. Therefore, the pair $(\langle s_1, s_2\rangle, \langle \mathcal{T}_1^G, \mathcal{T}_2^G\rangle)$, where s_i is the strategy represented by $\mathcal{T}_i$ for $i \in \{1,2\}$, realizes $\varphi_{no\_crash} \land \varphi_{cross_1} \land \varphi_{cross_2}$. The parallel composition of $\mathcal{T}_1$ and $\mathcal{T}_2$ is depicted in Figure 4.2. It is a strategy that allows both robots to move forward as long as at most one of them arrived at the crossing. Furthermore, starting with r_1, both robots take turns in crossing the intersection when both of them are at the crossing at several consecutive time steps. Hence, $\mathcal{T}_1 \mid\mid \mathcal{T}_2$ realizes $\varphi_{no\_crash} \land \varphi_{cross_1} \land \varphi_{cross_2}$ as well. △

In the following, we prove soundness and completeness of certifying synthesis with guarantee transducers by reducing the existence of LTL certificates to the existence of guarantee transducers and vice versa. Given vectors $\mathcal{S}$ and $\mathcal{G}$ of strategies and guarantee transducers for the system processes, respectively, such that $(\mathcal{S}, \mathcal{G})$ realizes an LTL specification φ, we intuitively construct a vector Ψ of LTL certificates that capture the exact behavior of the guarantee transducers. Then, $(\mathcal{S}, \Psi)$ realizes φ as well.

Lemma 4.1. *Let φ be an LTL formula over atomic propositions V with decomposition $\langle \varphi_1, \dots, \varphi_n\rangle$. Let $\mathcal{S}$ and $\mathcal{G}$ be vectors of strategies and guarantee transducers for the system processes, respectively. If $(\mathcal{S}, \mathcal{G})$ realizes φ, then there is a vector Ψ of LTL certificates such that $(\mathcal{S}, \Psi)$ realizes φ.*

Proof. Let $\mathcal{S} = \langle s_1, \dots, s_n\rangle$, $\mathcal{G} = \langle \mathcal{T}_1^G, \dots, \mathcal{T}_n^G\rangle$. For $p_j \in P^-$, let $\mathcal{G}_j := \left\{\mathcal{T}_i^G \mid p_i \in P^- \setminus \{p_j\}\right\}$ and let $\tilde{V}_j = \bigcup_{p_i \in P^- \setminus \{p_j\}} V_i^G$. Let $\tilde{V} = \bigcup_{p_j \in P^-} V_j^G$ Suppose that $(\mathcal{S}, \mathcal{G})$ realizes φ. We construct LTL certificates as follows: for system process $p_i \in P^-$, let ψ_i be an LTL formula over atomic propositions V_i^G describing the exact behavior of p_i's guarantee transducer $\mathcal{T}_i^G$, i.e., ψ_i is an LTL formula over atomic propositions V_i^G with $\mathcal{L}(\psi_i) = \mathit{Traces}(\mathcal{T}_i^G)$. Recall that a guarantee transducer has finitely many states. Therefore, such an LTL formula ψ_i can always be constructed by encoding the guarantee transducer $\mathcal{T}_i^G$. Since $V_i^G \subseteq V_i$ holds, ψ_i then indeed matches the form of an LTL certificate. Let $\Psi = \langle \psi_1, \dots, \psi_n\rangle$ and, for $p_j \in P^-$, let $\Psi_j = \left\{\psi_i \mid p_i \in P^- \setminus \{p_j\}\right\}$. We claim that $(\mathcal{S}, \Psi)$ realizes φ. Hence, we show in the following that $s_i \models \psi_i \land (\Psi_i \rightarrow \varphi_i)$ holds for all $p_i \in P^-$. More precisely, we show that $\sigma \in \mathcal{L}(\psi_i \land (\Psi_i \rightarrow \varphi_i))$ holds for all $\sigma \in \mathit{Traces}(\mathcal{T}_i) \cup (2^{V \setminus V_i})^\omega$, where $\mathcal{T}_i$ is the deterministic and complete finite-state Moore transducer representing s_i. Let $p_i \in P^-$ and let $\sigma \in \mathit{Traces}(\mathcal{T}_i) \cup (2^{V \setminus V_i})^\omega$.

First, we prove that strategy s_i does not deviate from p_i's certificate ψ_i, i.e., we show that $\sigma \cap V_i^G \in \mathcal{L}(\psi_i)$ holds. Since $(\mathcal{S}, \mathcal{G})$ realizes φ by assumption, $\mathcal{T}_i \preceq \mathcal{T}_i^G$ holds. By definition, $\mathcal{T}_i$ is a $(2^{I_i}, 2^{O_i})$-transducer, while $\mathcal{T}_i^G$ is a $(2^{I_i}, 2^{O_i^G})$-transducer. Moreover, by definition of guarantee outputs, we have $O_i^G \subseteq O_i$. Thus, by Proposition 4.1, we have $\sigma' \cap V_i^G \in \mathit{Traces}(\mathcal{T}_i^G)$ for all $\sigma' \in \mathit{Traces}(\mathcal{T}_1)$ since $V_i^G = I_i \cup O_i^G$ holds by definition. Clearly, we have $\sigma \cap V_i \in \mathit{Traces}(\mathcal{T}_i)$ by construction of σ. Therefore, $\sigma \cap V_i^G \in \mathcal{L}(\psi_i)$ follows with the construction of ψ_i.

Next, we prove that s_i realizes φ_i as long as all other system processes do not deviate from their certificates, i.e., we show that $\sigma \in \left\{\rho \in (2^V)^\omega \mid \rho \cap (\tilde{V}_i \cup V_i) \in \mathcal{L}(\Psi_i \rightarrow \varphi_i)\right\}$ holds. Since $(\mathcal{S}, \mathcal{G})$ realizes φ_i by assumption, in particular $s_i \models_{\mathcal{G}_i} \varphi_i$ holds. We have $\sigma \cap V_i \in \mathit{Traces}(\mathcal{T}_i)$ by construction of σ and thus, in particular, $\sigma \models_{\mathcal{G}_i} \varphi_i$ holds. If $\sigma \in \mathcal{V}_{\mathcal{G}_i}$ holds, then $\sigma \models \varphi_i$ follows with the definition of local satisfaction. Thus, by the semantics of implication, we also have $\sigma \models \Psi_i \rightarrow \varphi_i$ and therefore $\sigma \in \{\rho \in (2^V)^\omega \mid \rho \cap (\tilde{V}_i \cup V_i) \in \mathcal{L}(\Psi_i \rightarrow \varphi_i)\}$ holds. Otherwise, i.e., if σ is no valid computation for $\mathcal{G}_i$, then there exists a point in time $k \geq 0$ such that $\sigma_k \cap O^G_j \neq \mathit{comp}(\mathcal{T}^G_j, \sigma \cap I_j)$ holds for some guarantee transducer $\mathcal{T}^G_j \in \mathcal{G}_i$. By definition, $\mathcal{T}^G_j$ is a deterministic and complete finite-state transducer and thus $\mathit{comp}(\mathcal{T}^G_j, \sigma \cap I_j)$ is the single trace of $\mathcal{T}^G_j$ induced by $\sigma \cap I_j$. Thus, we have $\sigma \cap O^G_j \notin \mathit{Traces}(\mathcal{T}^G_j, \sigma \cap I_j)$. Since $I_j \cap O^G_j = \emptyset$ holds by definition of architectures and of guarantee outputs, $\sigma \cap O^G_j \notin \mathit{Traces}(\mathcal{T}^G_j)$ follows with the definition of traces. By construction of the LTL formula ψ_j, we have $\mathcal{L}(\psi_j) = \mathit{Traces}(\mathcal{T}^G_j)$ and therefore $\sigma \cap O^G_j \notin \mathcal{L}(\psi_j)$. Since $\mathcal{T}^G_j \in \mathcal{G}_i$ holds, we have $p_j \in P^- \setminus \{p_i\}$ and thus $\psi_j \in \Psi_i$ holds as well. Hence, $\sigma \cap \tilde{V}_i \notin \mathcal{L}(\Psi_i)$ follows with the semantics of conjunction. Thus, by the semantics of implication, $\sigma \in \{\rho \in (2^V)^\omega \mid \rho \cap (\tilde{V}_i \cup V_i) \in \mathcal{L}(\Psi_i \rightarrow \varphi_i)\}$ holds.

Therefore, for all processes $p_i \in P^-$ and all traces $\sigma \in \mathit{Traces}(\mathcal{T}_i) \cup (2^{V \setminus V_i})^\omega$, we have both $\sigma \cap V^G_i \in \{\sigma \in (2^V)^\omega \mid \sigma \cap V^G_i \in \mathcal{L}(\psi_i)\}$ and $\sigma \in \{\sigma \in (2^V)^\omega \mid \sigma \cap (\tilde{V}_i \cup V_i) \in \mathcal{L}(\Psi_i \rightarrow \varphi_i)\}$. Clearly, $V^G_i \subseteq \tilde{V} \cup V_i$ and $\tilde{V}_i \subseteq \tilde{V} \cup V_i$ hold. Thus, it follows with the semantics of conjunction that $\sigma \in \{\sigma \in (2^V)^\omega \mid \sigma \cap (\tilde{V} \cup V_i) \in \mathcal{L}(\psi_i \wedge (\Psi_i \rightarrow \varphi_i))\}$ holds for all system processes $p_i \in P^-$ and all $\sigma \in \mathit{Traces}(\mathcal{T}_i) \cup (2^{V \setminus V_i})^\omega$ as well. Hence, $(\mathcal{S}, \Psi)$ indeed realizes φ. □

Vice versa, we can construct a vector $\mathcal{G}$ of guarantee transducers from vectors $\mathcal{S}$ and Ψ of strategies and LTL certificates for the system processes, respectively. If $(\mathcal{S}, \Psi)$ realizes an LTL specification φ, then $(\mathcal{S}, \mathcal{G})$ realizes φ as well. Intuitively, we construct the guarantee transducers from the strategies by restricting the strategies to the guarantee variables.

Lemma 4.2. *Let φ be an LTL formula over atomic propositions V with decomposition $\langle \varphi_1, \dots, \varphi_n \rangle$. Let $\mathcal{S}$ and Ψ be vectors of strategies and LTL certificates for the system processes, respectively. If $(\mathcal{S}, \Psi)$ realizes φ, then there is a vector $\mathcal{G}$ of guarantee transducers such that $(\mathcal{S}, \mathcal{G})$ realizes φ.*

Proof. Let $\mathcal{S} = \langle s_1, \dots, s_n \rangle$ and $\Psi = \langle \psi_1, \dots, \psi_n \rangle$. For $p_i \in P^-$, let $\Psi_i := \{\psi_j \mid p_j \in P^- \setminus \{p_i\}\}$. Suppose that $(\mathcal{S}, \Psi)$ realizes φ. We construct guarantee transducers for the system processes as follows: for $p_i \in P^-$, let $\mathcal{T}_i = (T_i, T_{i,0}, \tau_i, \ell_i)$ be the $(2^{I_i}, 2^{O_i})$-transducer with Moore semantics representing s_i. The guarantee transducer $\mathcal{T}^G_i = (T^G_i, T^G_{i,0}, \tau^G_i, \ell^G_i)$ is then defined by

- $T^G_i = T_i$,
- $T^G_{i,0} = T_{i,0}$,
- $(t, \iota, t') \in \tau^G_i$ if, and only if, $(t, \iota, t') \in \tau_i$, and
- $(t, o) \in \ell^G_i$ if, and only if, there exists some $o' \in 2^{O_i}$ with $o' \cap O^G_i = o$ and $(t, o') \in \ell_i$.

Intuitively, $\mathcal{T}^G_i$ is thus a copy of $\mathcal{T}_i$, where the output of each state is restricted to the guarantee outputs O^G_i. Since $\mathcal{T}_i$ represents s_i, it has a finite number of states, is both deterministic and complete, and has Moore semantics. Thus, by construction, these attributes hold for $\mathcal{T}^G_i$ as well. Let $\mathcal{G} = \langle \mathcal{T}^G_1, \dots, \mathcal{T}^G_n \rangle$ and, for $p_j \in P^-$, let $\mathcal{G}_j = \{\mathcal{T}^G_i \mid p_i \in P^- \setminus \{p_j\}\}$. We claim that $(\mathcal{S}, \mathcal{G})$ realizes φ. Hence, we show in the following that both $s_i \models_{\mathcal{G}_i} \varphi_i$ and $\mathcal{T}_i \preceq \mathcal{T}^G_i$ hold for all system processes $p_i \in P^-$. Let $p_i \in P^-$ be some system process.

First, we prove that $\mathcal{T}_i \preceq \mathcal{T}_i^G$ holds. By construction of $\mathcal{T}_i^G$, the two transducers $\mathcal{T}_i$ and $\mathcal{T}_i^G$ only differ in the outputs of the states. The outputs of the states nevertheless agree on p_i's guarantee outputs, i.e., on the variables in O_i^G. By definition, the guarantee outputs are the only output variables that are shared between $\mathcal{T}_i$ and $\mathcal{T}_i^G$ and, in particular, $\mathcal{T}_i^G \subseteq \mathcal{T}_i$ holds. Hence, it follows immediately with the definition of transducer simulation that we have $\mathcal{T}_i \preceq \mathcal{T}_i^G$.

Next, we show that $s_i \models_{\mathcal{G}_i} \varphi_i$ holds. Thus, we prove that for all input sequences $\gamma \in (2^{I_i})^\omega$ and all sequences $\gamma' \in (2^{V \setminus V_i})^\omega$ of valuations of variables that p_i cannot observe, we have $comp(s_i, \gamma) \cup \gamma' \models_{\mathcal{G}_i} \varphi_i$. Let $\gamma \in (2^{I_i})^\omega$ and let $\gamma' \in (2^{V \setminus V_i})^\omega$. By assumption, $(\mathcal{S}, \Psi)$ realizes φ and therefore $s_i \models \psi_i \land (\Psi_i \rightarrow \varphi_i)$ holds. By the semantics of conjunction and by the definition of specification realization, $comp(s_i, \gamma) \cup \gamma' \models \Psi_i \rightarrow \varphi_i$ thus holds. For the sake of readability, let $\sigma := comp(s_i, \gamma) \cup \gamma'$. If $\sigma \models \Psi_i$ holds, then since $\sigma \models \Psi_i \rightarrow \varphi_i$ holds, $\sigma \models \varphi'$ follows immediately. Hence, $\sigma \models_{\mathcal{G}_i} \varphi_i$ follows with the definition of local satisfaction. Otherwise, i.e., if $\sigma \not\models \Psi_i$ holds, then there exists some $p_j \in P^- \setminus \{p_i\}$ such that $\sigma \not\models \psi_j$ holds. Since $(\mathcal{S}, \Psi)$ realizes φ_i by assumption, we also have $s_j \models \psi_j \land (\Psi_j \rightarrow \varphi_j)$. Thus, $s_j \models \psi_j$ and hence $Traces(\mathcal{T}_j) \cap V_j^G \subseteq \mathcal{L}(\psi_j)$ hold, where $\mathcal{T}_j$ is the finite-state transducer representing s_j. Since $\sigma \not\models \psi_j$ and thus $\sigma \cap V_j^G \notin \mathcal{L}(\psi_j)$ holds by assumption, $\sigma \notin Traces(\mathcal{T}_j)$ follows. By definition of traces, we therefore have $\sigma \notin Traces(\mathcal{T}_j, \sigma \cap I_j)$. By construction of the guarantee transducers, $\mathcal{T}_j^G$ is a copy of $\mathcal{T}_j$, which is the transducer representing s_j, where the outputs of each state are restricted to O_j^G. Therefore, we have $Traces(\mathcal{T}_j, \sigma \cap I_j) \cap V_j^G = Traces(\mathcal{T}_j^G, \sigma \cap I_j)$ and thus $\sigma \cap V_j^G \notin Traces(\mathcal{T}_j^G, \sigma \cap I_j)$ holds. Since guarantee transducers are both deterministic and complete, $\mathcal{T}_j^G$ produces exactly one trace on input sequence $\sigma \cap I_j$, namely $comp(\mathcal{T}_j^G, \sigma \cap I_j)$. Hence, $\sigma \cap V_j^G \neq comp(\mathcal{T}_j^G, \sigma \cap I_j)$ follows. Since $O_j^G \subseteq V_j^G$ holds by definition, we thus have $\sigma \cap O_j^G \neq comp(\mathcal{T}_j^G, \sigma \cap I_j) \cap O_j^G$. Therefore, σ is no valid computation for $\mathcal{G}_i$, i.e., $\sigma \notin \mathcal{V}_{\mathcal{G}_i}$ holds. Consequently, $\sigma \models_{\mathcal{G}_i} \varphi_i$ follows with the definition of local satisfaction. Since we chose $\gamma \in (2^{I_i})^\omega$ and $\gamma' \in (2^{V \setminus V_i})^\omega$ arbitrarily, $\sigma \models_{\mathcal{G}_i} \varphi_i$ follows. □

Note that the construction of the guarantee transducers in the proof of Lemma 4.2 only depends on the strategies, not on the LTL certificates. Hence, intuitively, we provide the entire strategy as guaranteed behavior and do not make use of the possibly more concise LTL certificates. However, this does not mean that guarantee transducers always represent the entire strategy in general. There might exist more concise guarantee transducers that represent the very same guaranteed behavior as the LTL certificates and still satisfy the requirements of certifying synthesis with guarantee transducers. Yet, theoretically, an LTL certificate could be more general than the strategy: consider two processes p_1 and p_2 with $I_1 = \{a\}$, $O_1 = \{b\}$, $I_2 = \{b\}$, and $O_2 = \{a\}$ as well as specification $\square(a \lor b)$. Simple strategies for p_1 and p_2 are, for instance, strategies that output b and a, respectively, in every step. While LTL certificates $\psi_1 = \square a$ and $\psi_2 = \square b$ suffice in this case, also the more general LTL certificates $\psi_1' = \psi_2' = \mathit{true}$ yield a valid solution of certifying synthesis. Every transducer that captures exactly the guaranteed behavior modeled by these LTL certificates is nondeterministic and thus, by definition, no guarantee transducer. Lemma 4.2, however, only considers the existence of *some* guarantee transducers such that the requirements of certifying synthesis are satisfied. Hence, we can utilize the deterministic transducers representing the strategies.

Since for every solution of certifying synthesis with LTL certificates, there exists one with guarantee transducers and vice versa, we can utilize the results from Section 4.2 to conclude that certifying synthesis with certificates represented by deterministic finite-state transducers is sound and complete. It follows immediately from Theorem 4.1 together with Lemmas 4.1 and 4.2 that there exist vectors of strategies and guarantee transducers realizing an LTL specification if, and only if, the parallel composition of the strategies realizes the specification:

Theorem 4.2. *Let φ be an LTL formula over atomic propositions V with decomposition $\langle \varphi_1, \ldots, \varphi_n \rangle$. Let $\mathcal{S} = \langle s_1, \ldots, s_n \rangle$ be a vector of strategies for the system processes. Then, there exists a vector $\mathcal{G}$ of guarantee transducers for the system processes such that $(\mathcal{S}, \mathcal{G})$ realizes φ if, and only if, $s_1 \mid\mid \ldots \mid\mid s_n \models \varphi$ holds.*

Proof. First, let there be a vector $\mathcal{G}$ of guarantee transducers such that $(\mathcal{S}, \mathcal{G})$ realizes φ. Then, by Lemma 4.1, there exists a vector Ψ of LTL certificates such that $(\mathcal{S}, \Psi)$ realizes φ. Therefore, by Theorem 4.1, $s_1 \mid\mid \ldots \mid\mid s_n \models \varphi$ holds.

Second, suppose that $s_1 \mid\mid \ldots \mid\mid s_n \models \varphi$ holds. Then, by Theorem 4.1, there exists a vector Ψ of LTL certificates such that $(\mathcal{S}, \Psi)$ realizes φ. Thus, by Lemma 4.2, there also exists a vector $\mathcal{G}$ of guarantee transducers such that $(\mathcal{S}, \mathcal{G})$ realizes φ. □

Hence, similar to LTL certificates, certifying synthesis with guarantee transducer allows for local reasoning and thus enables modularity of the system. At the same time, it still ensures that correct solutions are found for all realizable specifications. In particular, enforcing certificates to be deterministic does not rule out strategies that can be obtained with certifying synthesis with possibly nondeterministic LTL certificates. Nevertheless, nondeterministic certificates can generally be more concise than deterministic ones. Therefore, we also study the advantages and disadvantages of permitting nondeterminism in guarantee transducers in Section 4.6.

Since certifying synthesis with guarantee transducers is sound and complete, it is suitable for compositional synthesis of distributed systems. In the following section, we thus describe how strategies and certificates represented by guarantee transducers can be synthesized for the system processes and hence how the distributed synthesis problem can be solved practically with certifying synthesis.

4.4. Synthesizing Certificates

In this section, we present an algorithm for practically synthesizing strategies and certificates represented by guarantee transducers. Our approach is based on *bounded synthesis* [FS13] and incorporates the search for certificates and the local objectives formalized by certifying synthesis into the existing framework.

In monolithic bounded synthesis, the size of the strategy is bounded and, starting from one state, is only increased if no solution with this size is found (see Section 2.8.1). Thus, bounded synthesis produces size-optimal solutions. Since we additionally synthesize certificates represented by finite-state transducers, we bound the sizes of the certificates as well, allowing for size-optimal solutions in either terms of strategies or certificates.

In the following, we first present which formalisms of certifying synthesis with guarantee transducers presented in Section 4.3 need to be slightly adapted to incorporate certifying synthesis into existing bounded synthesis frameworks easily. We prove soundness and completeness of certifying synthesis with these adaptions. Afterward, we introduce a SAT constraint system that encodes the search for strategies and *deterministic* guarantee transducers that satisfy the requirements of certifying synthesis.

4.4.1. Local Strategies

Like for classical bounded synthesis [FS13, FFRT17] for monolithic systems, we reduce the search for a solution of certifying synthesis of a certain size to a constraint-solving problem. We employ parts of the existing bounded synthesis algorithm, particularly the concept of *valid annotations* of run graphs to determine whether or not a strategy realizes the given specification (see Definition 2.21). Therefore, we need to slightly adapt the formalisms for certifying synthesis with guarantee transducers presented in Section 4.3 in order to incorporate the local objectives of certifying synthesis into the concept of valid annotations.

In Section 4.3, we utilized *local satisfaction* to formalize that, in certifying synthesis with guarantee transducers, a strategy only needs to realize its specification if the other processes do not deviate from their guaranteed behavior formalized in their certificates. Hence, we changed the satisfaction condition with respect to classical notions. However, determining whether or not a strategy classically realizes an LTL formula is *the* crucial part of existing bounded synthesis frameworks and, in particular, the SAT constraint system [FFRT17] that encodes the bounded synthesis problem. Therefore, we present a different formalization of certifying synthesis with guarantee transducers in this section. It relies on classical satisfaction, thus allowing for reusing parts of the SAT constraint system for monolithic bounded synthesis, particularly valid annotations of run graphs, while still ensuring that a strategy only needs to satisfy the specification if the other processes do not deviate from their certificates.

Recall that to determine whether or not a strategy s_i for a system process $p_i \in P^-$ realizes an LTL formula φ_i in bounded synthesis, we first construct a universal co-Büchi automaton $\mathcal{A}_i$ that accepts the language of φ_i, i.e., an automaton with $\mathcal{L}(\mathcal{A}_i) = \mathcal{L}(\varphi_i)$. Then, s_i realizes φ_i if, and only if, $\mathcal{A}_i$ accepts $comp(s_i, \gamma) \cup \gamma'$ for all $\gamma \in (2^{I_i})^\omega$ and all $\gamma' \in (2^{V \setminus V_i})^\omega$. Hence, since the acceptance of a universal co-Büchi automaton is determined by the number of visits to rejecting states during a run, we check whether or not all runs of $\mathcal{A}_i$ induced by some computation of s_i contain only finitely many visits to rejecting states (see Section 2.8.1). Observe that a *finite* run of a co-Büchi automaton can never visit rejecting states infinitely often. Consequently, all finite runs are trivially accepting. Hence, by ensuring that $\mathcal{A}_i$ produces finite runs on all sequences that deviate from the certificate of some other system process, we can check local satisfaction with the very same concept for determining classical satisfaction, namely by using valid annotations for checking whether the runs of $\mathcal{A}_i$ induced by the computations of s_i visit rejecting states only finitely often.

There are two possibilities for a universal co-Büchi automaton $\mathcal{A}_i$ to produce finite runs on the computation $comp(s_i, \gamma)$ of the strategy s_i on some input sequence $\gamma \in (2^{I_i})^\omega$. First, $\mathcal{A}_i$ can be incomplete, i.e., there can be a state q of $\mathcal{A}_i$ that occurs in a run of $\mathcal{A}_i$ induced by the respective computation $comp(s_i, \gamma)$ of s_i at point in time k and that does not have any outgoing edge for input valuation $comp(s_i, \gamma)_{k+1}$, i.e., we have $(q, comp(s_i, \gamma)_{k+1}, q') \notin \delta$ for all $q' \in Q$, where Q is the set of states of $\mathcal{A}_i$ and δ is its transition relation. Second, the respective computation $comp(s_i, \gamma)$ of s_i can be finite. In classical bounded synthesis, $\mathcal{A}_i$ accepts the language of φ_i, i.e., we have $\mathcal{L}(\mathcal{A}_i) = \mathcal{L}(\varphi_i)$. Thus, the former possibility requires altering the universal co-Büchi automaton $\mathcal{A}_i$ to incorporate the other system processes' certificates, for instance, when using LTL certificates. Consequently, $\mathcal{A}_i$ depends on the certificates and is not fixed, which is a major change with respect to classical bounded synthesis algorithms. The latter possibility, in contrast, only requires altering the strategies. To ensure that the computation of s_i is finite on input sequences that do not match the other processes' certificates, we can model strategies with *transition-incomplete transducers* instead of complete ones. Since we synthesize the strategies or, more precisely, the finite-state transducers representing them, in bounded synthesis anyhow, we thus only need to slightly alter the encoding of the strategies we are searching for, which is much less invasive than altering $\mathcal{A}_i$. The transducers representing strategies, however, are still deterministic and labeling-complete.

Therefore, we focus on this possibility and model strategies with deterministic and labeling-complete but *transition-incomplete* finite-state Moore transducers in the following. Intuitively, their transition relation is defined such that the computation of a strategy is infinite if, and only if, the other processes do not deviate from the behavior formalized in their certificates. Note that system strategies defined according to Definition 2.13 cannot produce finite computations as they are modeled with functions. Thus, representing system strategies with transition-incomplete finite-state Moore transducers is, strictly speaking, not possible. Therefore, we define *local strategies*, a variant of system strategies that can be modeled with transition-incomplete finite-state Moore transducers, as follows:

Definition 4.7 (Local Strategy)**.**
Let $p_i \in P^-$ be some system process. Let $\mathcal{P} \subseteq P^- \setminus \{p_i\}$ be a set of other system processes and let $\mathcal{G}$ be a set of guarantee transducers, one for each process in $\mathcal{P}$. A *local strategy* $\hat{s}_i : (2^{V_i})^* \times 2^{I_i} \rightharpoonup 2^{O_i}$ for p_i with respect to $\mathcal{G}$ is represented by a deterministic and labeling-complete finite-state $(2^{I_i}, 2^{O_i})$-transducer $\hat{\mathcal{T}_i}$ with Moore semantics. For all $\gamma \in (2^{I_i})^\omega$ and all $\sigma \in Traces(\hat{\mathcal{T}_i}, \gamma)$ it holds that (i) if σ is infinite, then there exists some $\gamma' \in (2^{V \setminus V_i})^\omega$ such that $\sigma \cup \gamma' \in \mathcal{V}_\mathcal{G}$ holds, and (ii) if σ is finite, then $\sigma \cdot (\gamma_{|\sigma|} \cup o) \cup \gamma' \notin \mathcal{H}^{\mathcal{G}}_{|\sigma|+1}$ holds for all $o \in 2^{O_i}$ and all $\gamma' \in (2^{V \setminus V_i})^*$ with $|\gamma'| = |\sigma| + 1$.

Intuitively, a finite-state Moore transducer representing a local strategy thus omits all transitions that are invoked by an input that may only occur if the other processes deviate from their certificates, possibly resulting in labeling-incompleteness. For an input sequence $\gamma \in (2^{I_i})^\omega$ that does not match the other processes' guaranteed behavior, a local strategy s_i thus encounters, at some point in time k, the situation that in the current state of the transducer $\mathcal{T}_i$ representing s_i there does not exist an outgoing transition that matches γ_{k+1}. Since a local strategy is represented by a deterministic finite-state transducer, the current state at point in time k is unique. Therefore every run of the transducer representing s_i ends at point in time k, resulting in a *finite* computation of the local strategy.

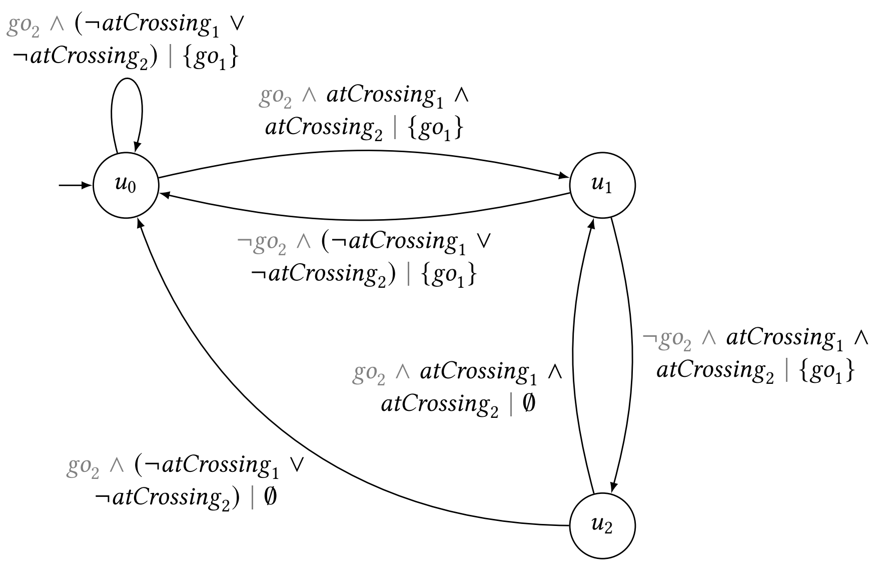

$go_2 \land (\neg atCrossing_1 \lor \neg atCrossing_2) \mid \{go_1\}$
$go_2 \land atCrossing_1 \land atCrossing_2 \mid \{go_1\}$
u_0
u_1
$\neg go_2 \land (\neg atCrossing_1 \lor \neg atCrossing_2) \mid \{go_1\}$
$\neg go_2 \land atCrossing_1 \land atCrossing_2 \mid \{go_1\}$
$go_2 \land atCrossing_1 \land atCrossing_2 \mid \emptyset$
$go_2 \land (\neg atCrossing_1 \lor \neg atCrossing_2) \mid \emptyset$
u_2

Figure 4.3.: Local strategy $\hat{\mathcal{T}}_1$ for robot r_1 from the running example. Atomic propositions denoting output variables of the other robot are highlighted in gray.

Example 4.7. Consider the robots r_1 and r_2 from the running example introduced in Section 4.1. Furthermore, consider the guarantee transducers $\mathcal{T}_1^G$ and $\mathcal{T}_2^G$ for r_1 and r_2, respectively, depicted in Figure 4.1. Recall that the guarantee transducers $\mathcal{T}_1^G$ and $\mathcal{T}_2^G$ can be interpreted as strategy transducers $\mathcal{T}_1$ and $\mathcal{T}_2$ for the robots r_1 and r_2, respectively, as well. Local strategies $\hat{\mathcal{T}}_1$ and $\hat{\mathcal{T}}_2$ for the robots r_1 and r_2 with respect to the sets $\{\mathcal{T}_2^G\}$ and $\{\mathcal{T}_1^G\}$, which are based on the strategy transducers $\mathcal{T}_1$ and $\mathcal{T}_2$, are depicted in Figures 4.3 and 4.4, respectively.

Robot r_1's local strategy $\hat{\mathcal{T}}_1$, depicted in Figure 4.3, looks similar to $\mathcal{T}_1$, depicted in Figure 4.1a. However, the transition labels contain restrictions on the output variable go_2 of robot r_2, which match r_2's guaranteed behavior formalized in $\mathcal{T}_2^G$, depicted in Figure 4.1b. Thus, in particular, $\hat{\mathcal{T}}_1$ does not define transitions that cannot be taken in the interplay of $\mathcal{T}_1$ and $\mathcal{T}_2^G$. For instance, $\mathcal{T}_2^G$ ensures that go_2 is played in the very first time step as well as every time at most one of the robots arrived at the crossing. As $\mathcal{T}_1$ is always in state u_0 in this situation, $\hat{\mathcal{T}}_1$ does not have an outgoing transition for $\neg go_2$ in state u_0.

Robot r_2's local strategy $\hat{\mathcal{T}}_2$, depicted in Figure 4.4, differs from $\mathcal{T}_2$ in the number of states and the transition structure. This is necessary to correctly incorporate r_1's guaranteed behavior: in $\mathcal{T}_2$, we are not able to distinguish even and odd positions of a sequence of consecutive time steps in which both robots arrive at the crossing. Therefore, we cannot accurately capture r_1's guaranteed behavior, which differs in these situations, with a two-state transducer, and hence we need to enlarge the state space. Nevertheless, $\hat{\mathcal{T}}_2$ defines an analogous behavior as $\mathcal{T}_2$, yet having no outgoing transitions for situations that cannot occur in the interplay of $\mathcal{T}_2$ and $\mathcal{T}_1^G$. △

Note that the requirements on the finiteness and infiniteness of computations posed by the definition of local strategies ensure that a finite-state transducer representing a local strategy

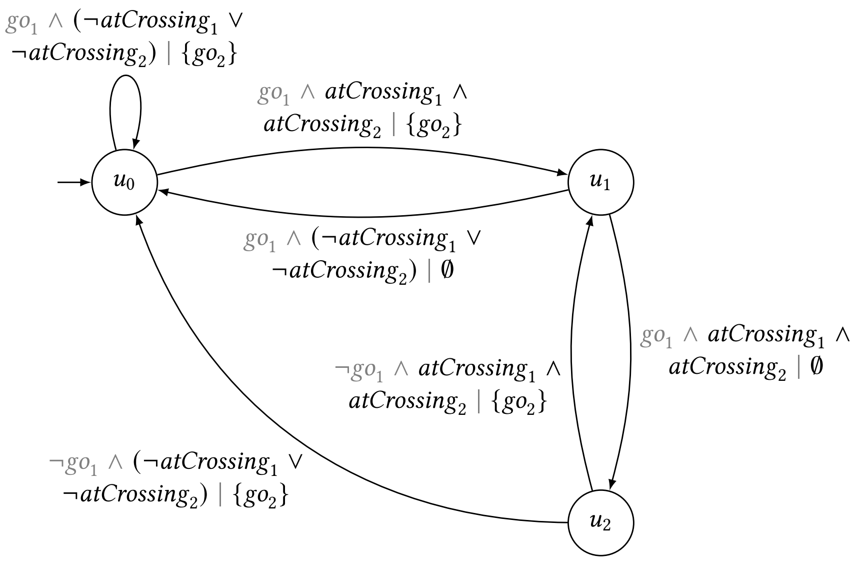

$go_1 \wedge (\neg atCrossing_1 \vee \neg atCrossing_2) \mid \{go_2\}$
$go_1 \wedge atCrossing_1 \wedge atCrossing_2 \mid \{go_2\}$
u_0
u_1
$go_1 \wedge (\neg atCrossing_1 \vee \neg atCrossing_2) \mid \emptyset$
$go_1 \wedge atCrossing_1 \wedge atCrossing_2 \mid \emptyset$
$\neg go_1 \wedge atCrossing_1 \wedge atCrossing_2 \mid \{go_2\}$
$\neg go_1 \wedge (\neg atCrossing_1 \vee \neg atCrossing_2) \mid \{go_2\}$
u_2

Figure 4.4.: Local strategy $\hat{\mathcal{T}}_2$ for robot r_2 from the running example. Atomic propositions denoting output variables of the other robot are highlighted in gray.

has at least one initial state. Since the transducer is also deterministic by definition, it thus follows that a labeling-incomplete finite-state Moore transducer representing a local strategy has exactly one initial state:

Proposition 4.2. *Let $p_i \in P^-$ be some system process. Let $\mathcal{P} \subseteq P^- \setminus \{p_i\}$ be a set of other system processes and let $\mathcal{G}$ be a set of guarantee transducers, one for each process in $\mathcal{P}$. Let $\hat{s}_i$ be a local strategy for p_i with respect to $\mathcal{G}$ and let $\hat{\mathcal{T}}_i = (\hat{T}_i, \hat{T}_{i,0}, \hat{\tau}_i, \hat{\ell}_i)$ be the finite-state transducer representing $\hat{s}_i$. Then, $|\hat{T}_{i,0}| = 1$ holds.*

Proof. Let $O_i^G = \bigcup_{p_j \in \mathcal{P}} O_j^G$. Let $\mathcal{T} = (T, T_0, \mathcal{T}, \ell)$ be the parallel composition of the guarantee transducers in $\mathcal{G}$. Since guarantee transducers are deterministic and complete Moore transducers and since the sets of output variables of different processes are disjoint by definition of system architectures, $\mathcal{T}$ is deterministic and complete and has Moore semantics by Lemma 2.1 as well. Hence, there exists a unique initial state $t_0 \in T_0$. Furthermore, there exists a unique valuation $o \in 2^{O_i^G}$ of the guarantee outputs of the processes in $\mathcal{P}$ such that $(t_0, o) \in \ell$ holds. Let $\rho \in (2^V)^\omega$ be some infinite sequence with $\rho_0 \cap O_i^G = o$. Then, in particular, $\rho_0 \cap O_j^G = comp(\mathcal{T}_j^G, \rho \cap I_j) \cap O_j^G$ holds for all $p_j \in \mathcal{P}$ and thus, by definition of valid histories, we have $\rho_{|1} \in \mathcal{H}_1^{\mathcal{G}}$.

Since $\hat{\mathcal{T}}_i$ is deterministic by definition and thus, in particular, transition-deterministic, we have $|\hat{T}_{i,0}| \leq 1$. Suppose that $|\hat{T}_{i,0}| = 0$ holds. Then, $|\sigma| = 0$ holds for the unique trace $\sigma \in Traces(\hat{\mathcal{T}}_i, \rho \cap I_i)$ of $\hat{\mathcal{T}}_i$ induced by $\rho \cap I_i$. Since $\hat{s}_i$ is a local strategy for p_i with respect to $\mathcal{G}$, we thus have $((\rho_0 \cap I_i) \cup o) \cup \gamma' \notin \mathcal{H}_1^{\mathcal{G}}$ for all $\gamma' \in (2^{V \setminus V_i})^\omega$ with $|\gamma'| = 1$ and all $o \in 2^{O_i}$. Thus, in particular $((\rho_0 \cap I_i) \cup (\rho_o \cap O_i)) \cup (\rho_0 \cap (V \setminus V_i) \notin\in \mathcal{H}_1^{\mathcal{G}}$ holds; contradicting $\rho_{|1} \in \mathcal{H}_1^{\mathcal{G}}$. Hence, we have $|\hat{T}_{i,0}| \neq 0$ and, since $|\hat{T}_{i,0}| \leq 1$ holds, $|\hat{T}_{i,0}| = 1$ follows. □

Utilizing the notion of local strategies, we now reformulate the definition of certifying synthesis with guarantee transducers from Section 4.3.

Definition 4.8 (Certifying Synthesis with Local Strategies)**.**
Let φ be an LTL formula over atomic propositions V with decomposition $\langle \varphi_1, \dots, \varphi_n \rangle$. Let $\mathcal{G} = \langle \mathcal{T}_1^G, \dots, \mathcal{T}_n^G \rangle$ be a vector of guarantee transducers for the system processes and, for $p_j \in P^-$, let $\mathcal{G}_j = \{\mathcal{T}_i^G \mid p_i \in P^- \setminus \{p_j\}\}$. Let $\hat{\mathcal{S}} = \langle \hat{s}_1, \dots, \hat{s}_n \rangle$ such that $\hat{s}_i$ is a local strategy for $p_i \in P^-$ with respect to $\mathcal{G}_i$. Let $\hat{\mathcal{T}_i}$ be the deterministic and labeling-complete finite-state Moore transducer representing $\hat{s}_i$. If $\hat{\mathcal{T}_i} \preceq \mathcal{T}_i^G$ holds and if, for all $\gamma \in (2^{I_i})^\omega$, $\gamma' \in (2^{V \setminus V_i})^\omega$, either $comp(\hat{s}_i, \gamma)$ is finite or $comp(\hat{s}_i, \gamma) \cup \gamma' \models \varphi$ holds, then we say that $(\hat{\mathcal{S}}, \mathcal{G})$ realizes φ. *Certifying synthesis* for φ derives vectors $\hat{\mathcal{S}}$ and $\mathcal{G}$ such that $(\hat{\mathcal{S}}, \mathcal{G})$ realizes φ.

Note here that the notion of transducer simulation for ensuring that a strategy does not deviate from its own certificate does not need to be altered when using local strategies instead of complete strategies: for $\hat{\mathcal{T}_i} \preceq \mathcal{T}_i^G$ to hold, transducer simulation requires a transition in the guarantee transducer $\mathcal{T}_i^G$ whenever there is a matching one in $\hat{\mathcal{T}_i}$. Hence, whenever the local strategy does not have an outgoing transition for a specific input in the current state, i.e., whenever the local strategy is transition-incomplete, transducer simulation does not require the existence of a transition in this situation either. Since guarantee transducers are complete, however, this permits *any* behavior in the guarantee transducer in such situations. While this differs from the previous definitions of certifying synthesis, both with guarantee transducers and LTL certificates, it does not affect soundness or completeness since, intuitively, situations in which a guarantee transducers' behavior does not match the local strategy cannot occur in the interplay of all strategies.

Certifying synthesis with local strategies indeed utilizes classical satisfaction instead of local satisfaction. Thus, we can reuse existing bounded synthesis frameworks and, in particular, valid annotations of run graphs (see Section 2.8.1), to determine whether a local strategy realizes an LTL formula. In the following, we study the relationship between certifying synthesis with local strategies as defined above (see Definition 4.8) and certifying synthesis with guarantee transducers as introduced in Section 4.3. First, we introduce the notions of *extending local strategies* and *restricting complete strategies*. This allows for comparing local strategies with classical satisfaction to complete strategies with local satisfaction. Afterward, we then utilize the concepts of strategy extension and restriction to show soundness and completeness of certifying synthesis with local strategies.

4.4.2. Strategy Extension and Restriction

Given a local strategy $\hat{s}_i$ for system process $p_i \in P^-$ and a set $\mathcal{G}$ of guarantee transducers, we can construct a complete strategy s_i, i.e., a strategy according to Definition 2.13 represented by a complete finite-state transducer, by, intuitively, *extending* $\hat{s}_i$ with its own guaranteed behavior. More precisely, s_i behaves as $\hat{s}_i$ up to the point in time $k \geq 0$ at which $\hat{s}_i$ does not produce any further valuation of output variables. From point in time k on, s_i then behaves as p_i's certificate, modeled by guarantee transducer $\mathcal{T}_i^G$, does. Note that whenever $\hat{s}_i$ produces an infinite computation, s_i does not switch from $\hat{s}_i$ to p_i's guarantee but always behaves as $\hat{s}_i$ does. Formally, we construct a deterministic and complete finite-state Moore transducer modeling the full strategy s_i from $\hat{s}_i$ as follows:

Definition 4.9 (Strategy Extension).
Let $p_i \in P^-$ be a system process and let $\mathcal{T}_i^G$ be a guarantee transducer for p_i. Let $\mathcal{P} \subseteq P^- \setminus \{p_i\}$ be a set of other system processes and let $\mathcal{G}$ be a set of guarantee transducers, one for each process in $\mathcal{P}$. Let $\hat{s}_i$ be a local strategy for p_i with respect to $\mathcal{G}$. Let $\hat{\mathcal{T}_i} = (\hat{T}_i, \hat{T}_{i,0}, \hat{\tau}_i, \hat{\ell}_i)$ be the transition-incomplete finite-state $(2^{I_i}, 2^{O_i})$-transducer representing $\hat{s}_i$. The extension extend$(\hat{s}_i, \mathcal{T}_i^G)$ of $\hat{s}_i$ is represented by a finite-state $(2^{I_i}, 2^{O_i})$-transducer $\mathcal{T}_i = (T_i, T_{i,0}, \tau_i, \ell_i)$ which we construct from $\hat{\mathcal{T}_i}$ and $\mathcal{T}_i^G$ as follows:

- $T_i = (\hat{T}_i \cup \{\bot\}) \times T_i^G$,
- $T_{i,0} = \hat{T}_{i,0} \times T_{i,0}^G$ and
- $((\hat{t}, t), \iota, (\hat{t}', t')) \in \tau_i$ if, and only if, $(t, \iota, t') \in \tau_i^G$ and either $(\hat{t}, \iota, \hat{t}') \in \hat{\tau}_i$, or both $\hat{t}' = \bot$ and $\forall \hat{t}'' \in \hat{T}.\ (\hat{t}, \iota, \hat{t}'') \notin \hat{\tau}_i$, or $\hat{t} = \bot$ holds.
- $((\hat{t}, t), o) \in \ell_i$ if, and only if, either both $\hat{t} \neq \bot$ and $(\hat{t}, o) \in \hat{\ell}_i$ hold, or we have both $\hat{t} = \bot$ and $o = \mathit{pick}\left(\{o \in 2^{O_i} \mid (t, o \cap O_i^G) \in \ell_i^G\}\right)$,

where $\mathit{pick}(M)$ picks one element from the non-empty set M.

Intuitively, the transducer representing extend$(\hat{s}_i, \mathcal{T}_i^G)$ keeps track of both the behavior of the possibly incomplete transducer $\hat{\mathcal{T}_i}$ representing the local strategy $\hat{s}_i$ and the behavior of the guarantee transducer $\mathcal{T}_i^G$ for system process $p_i \in P^-$. If, for some input sequence $\gamma \in (2^{I_i})^\omega$, the local strategy $\hat{s}_i$ produces an infinite computation, i.e., if $\hat{s}_i$ does not "get stuck" at some point in time, the extended strategy extend$(\hat{s}_i, \mathcal{G})$ always follows both $\hat{s}_i$ and $\mathcal{T}_i^G$ and produces $\hat{s}_i$'s outputs. As soon as $\hat{s}_i$ "gets stuck", however, the extended strategy extend$(\hat{s}_i, \mathcal{T}_i^G)$ cannot follow $\hat{s}_i$ anymore but only $\mathcal{T}_i^G$. It then produces some extension of the outputs of $\mathcal{T}_i^G$, which are a subset of all output variables of p_i, to the set O_i of all output variables.

Since the transducer $\mathcal{T}_i$ representing extend$(\hat{s}_i, \mathcal{T}_i^G)$ is built from the deterministic transducers $\hat{\mathcal{T}_i}$ and $\mathcal{T}_i^G$ and since the transition relation of $\mathcal{T}_i$ always follows one of them, $\mathcal{T}_i$ is transition-deterministic as well. Furthermore, $\mathcal{T}_i^G$ is complete, while $\hat{\mathcal{T}_i}$ might be transition-incomplete. Since the transition relation of $\mathcal{T}_i$ *always* follows $\mathcal{T}_i^G$, however, $\mathcal{T}_i$ is transition-complete as well. Since $\mathcal{T}_i$'s labeling relation follows $\hat{\mathcal{T}_i}$ for states $(\hat{t}, t)$ with $\hat{t} \neq \bot$ and since $\hat{\mathcal{T}_i}$ is both labeling-deterministic and labeling-complete, $\mathcal{T}_i$ has exactly one output for such states as well. For states $(\hat{t}, t)$ with $\hat{t} = \bot$, in contrast, $\mathcal{T}_i$'s output is defined by a *unique* valuation of output variables which, restricted to the guarantee outputs, is an output of $\mathcal{T}_i^G$ in state t. Note here that the set of all such valuations is non-empty since $\mathcal{T}_i^G$ is labeling-complete. In fact, the valuation of the guarantee outputs is already uniquely defined by ℓ_i^G since we consider $\mathcal{T}_i^G$ to be labeling-deterministic here as well. However, the function *pick* would ensure labeling-determinism for such states also if $\mathcal{T}_i^G$ would not be labeling-deterministic, Therefore, the transducer $\mathcal{T}_i$ is both labeling-deterministic and labeling-complete as well. Lastly, since both $\hat{\mathcal{T}_i}$ and $\mathcal{T}_i^G$ consist of a finite number of states, it follows immediately from the construction of $\mathcal{T}_i$ that it has a finite number of states as well. Thus, $\mathcal{T}_i$ is a deterministic and complete finite-state transducer and hence it indeed represents a complete strategy. Furthermore, $\hat{s}_i$ and $\text{extend}(\hat{s}_i, \mathcal{T}_i^G)$ agree on input sequences on which $\hat{s}_i$ produces infinite computations:

Lemma 4.3. *Let $p_i \in P^-$ be a system process and let $\mathcal{T}_i^G$ be a guarantee transducer for p_i. Let $\mathcal{P} \subseteq P^- \setminus \{p_i\}$ be a set of other system processes and let $\mathcal{G}$ be a set of guarantee transducers, one for each process in $\mathcal{P}$. Let $\hat{s}_i$ be a local strategy for p_i with respect to $\mathcal{G}$. Let $s_i := \text{extend}(\hat{s}_i, \mathcal{T}_i^G)$. Let $\gamma \in (2^{I_i})^\omega$. Then, $comp(\hat{s}_i, \gamma)_k = comp(s_i, \gamma)_k$ holds for all k with $0 \leq k < |comp(\hat{s}_i, \gamma)|$.*

Proof. Let $\mathcal{T}_i = (T_i, T_{i,0}, \tau_i, \ell_i)$ and $\hat{\mathcal{T}}_i = (\hat{T}_i, \hat{T}_{i,0}, \hat{\tau}_i, \hat{\ell}_i)$ be the finite-state transducers representing s_i and $\hat{s}_i$, respectively. By definition, $\hat{\mathcal{T}}_i$ is deterministic and labeling-complete. As outlined above, $\mathcal{T}_i$ is both deterministic and complete. Let $\gamma \in (2^{I_i})^\omega$. Let $\pi \in Paths(\mathcal{T}_i, \gamma)$ and $\hat{\pi} \in Paths(\hat{\mathcal{T}}_i, \gamma)$ be the unique paths produced by $\mathcal{T}_i$ and $\hat{\mathcal{T}}_i$, respectively, on γ. Then, it follows from the construction of $\mathcal{T}_i$ and thus from the definition of strategy extension, that for all k with $0 \leq k < |comp(\hat{s}_i, \gamma)|$, if $\hat{\pi}_k = (\hat{t}, o)$, then there exists some $t^G \in T_i^G$ such that $\pi_k = ((\hat{t}, t^G), o)$ holds. Thus, in particular, $\#_1(\hat{\pi}_k) = \#_1(\#_1(\pi_k))$ and $\#_2(\hat{\pi}_k) = \#_2(\pi_k)$ hold for all k with $0 \leq k < |\hat{\pi}|$. Let $\sigma \in Traces(\mathcal{T}_i, \gamma)$ and $\hat{\sigma} \in Traces(\hat{\mathcal{T}}_i, \gamma)$ be the unique traces corresponding to π and $\hat{\pi}$, respectively. Then, it follows that $\sigma_k = \hat{\sigma}_k$ for all k with $0 \leq k < |\hat{\sigma}|$ holds as well by definition of traces. Hence, by definition of computations, $comp(\hat{s}_i, \gamma)_k = comp(s_i, \gamma)_k$ holds for all points in time k with $0 \leq k < |comp(\hat{s}_i, \gamma)|$. □

The extension of a local strategy $\hat{s}_i$ for a set $\mathcal{G}_i$ of guarantee transducers according to Definition 4.9 to a complete strategy then preserves realization: if the local strategy $\hat{s}_i$ realizes an LTL formula φ_i, then the complete strategy $\text{extend}(\hat{s}_i, \mathcal{T}_i^G)$ *locally* realizes φ_i with respect to $\mathcal{G}_i$. Furthermore, it follows from the construction of $\mathcal{T}_i$ that $\mathcal{T}_i^G$ simulates $\mathcal{T}_i$. Lastly, the parallel composition of the local strategies and the parallel composition of their extensions coincide. Note that this immediately establishes that the parallel composition of the local strategies produces a unique and infinite computation for every input sequence. Therefore, we can lift a solution of certifying synthesis with local strategies to a solution of certifying synthesis with local satisfaction using strategy extension:

Lemma 4.4. *Let φ be an LTL formula over atomic propositions V with decomposition $\langle \varphi_1, \dots, \varphi_n \rangle$. Let $\mathcal{G} = \langle \mathcal{T}_1^G, \dots, \mathcal{T}_n^G \rangle$ be a vector of guarantee transducers for the system processes and, for $p_j \in P^-$, let $\mathcal{G}_j = \left\{ \mathcal{T}_i^G \mid p_i \in P^- \setminus \{p_j\} \right\}$. Let $\hat{\mathcal{S}} = \langle \hat{s}_1, \dots, \hat{s}_n \rangle$ be a vector of local strategies for the system processes such that $\hat{s}_i$ is a local strategy for $p_i \in P^-$ with respect to $\mathcal{G}_i$. Let $\mathcal{S} = \langle s_1, \dots, s_n \rangle$ such that $s_i = \text{extend}(\hat{s}_i, \mathcal{T}_i^G)$ holds for all $p_i \in P^-$. If $(\hat{\mathcal{S}}, \mathcal{G})$ realizes φ, then $(\mathcal{S}, \mathcal{G})$ realizes φ as well and $Traces(\hat{\mathcal{T}}_1 \mid\mid \dots \mid\mid \hat{\mathcal{T}}_n) = Traces(\mathcal{T}_1 \mid\mid \dots \mid\mid \mathcal{T}_n)$ holds, where $\mathcal{T}_i$ and $\hat{\mathcal{T}}_i$ are the finite-state transducers representing s_i and $\hat{s}_i$, respectively.*

Proof. Let $\hat{\mathcal{T}}_i = (\hat{T}_i, \hat{T}_{i,0}, \hat{\tau}_i, \hat{\ell}_i)$, $\mathcal{T}_i = (T_i, T_{i,0}, \tau_i, \ell_i)$, and $\mathcal{T}_i^G = (T_i^G, T_{i,0}^G, \tau_i^G, \ell_i^G)$. Let $(\hat{\mathcal{S}}, \mathcal{G})$ realize φ. Then, by definition of certifying synthesis with local strategies, we have, for all $p_i \in P^-$, both $\hat{\mathcal{T}}_i \preceq \mathcal{T}_i^G$ and, for all $\gamma \in (2^{I_i})^\omega$, $\gamma' \in (2^{V \setminus V_i})^\omega$, either $comp(\hat{s}_i, \gamma)$ is finite or $comp(\hat{s}_i, \gamma) \cup \gamma' \models \varphi_i$ holds. To prove that $(\mathcal{S}, \mathcal{G})$ realizes φ as well, we show that both $s_i \models_{\mathcal{G}_i} \varphi_i$ and $\mathcal{T}_i \preceq \mathcal{T}_i^G$ hold for all system processes $p_i \in P^-$. Let $p_i \in P^-$ be some system process.

First, we show that $\mathcal{T}_i \preceq \mathcal{T}_i^G$ holds. Since $(\hat{\mathcal{S}}, \mathcal{G})$ realizes φ by assumption, in particular $\hat{\mathcal{T}_i} \preceq \mathcal{T}_i^G$ holds. Let $R : \hat{T}_i \times T_i^G$ be the relation establishing the simulation. We construct a relation $R' : T_i \times T_i^G$ establishing the simulation $\mathcal{T}_i \preceq \mathcal{T}_i^G$ from R as follows: $((\hat{t}, t), t^G) \in R$ if, and only if, $t = t^G$ and either $(\hat{t}, t^G) \in R$ or $\hat{t} = \bot$ holds It remains to show that R' satisfies the properties of a simulation relation. Clearly, $(t_0, t_0^G) \in R'$ holds for all $t_0 \in T_0$ and all $t_0^G \in T_0^G$ by construction of $\mathcal{T}_i$ and R' and since R satisfies the properties of a simulation relation. Let $((\hat{t}, t^G), t^G) \in R'$. If $\hat{t} = \bot$ holds, then $\{o \mid (t^G, o) \in \ell_i^G\} = \{o \cap O_i^G \mid ((\hat{t}, t^G), o) \in \ell_i\}$ follows immediately from the definition of strategy extension. If $\hat{t} \neq \bot$ holds, then we have $(\hat{t}, t^G) \in R$ by construction of R' and thus, in particular, $\{o \mid (t^G, o) \in \ell_i^G\} = \{o \cap O_i^G \mid (\hat{t}, o) \in \hat{\ell}_i\}$ holds. Hence, $\{o \mid (t^G, o) \in \ell_i^G\} = \{o \cap O_i^G \mid ((\hat{t}, t^G), o) \in \ell_i\}$ follows with the construction of ℓ_i also if $\hat{t} \neq \bot$ holds. Furthermore, there is only a transition $((\hat{t}, t), \iota, (\hat{t}', t')) \in \tau_i$ in $\mathcal{T}_i$ if there is a transition $(t, \iota, t') \in \tau_i^G$ in $\mathcal{T}_i^G$ as well. Hence, the second requirement of simulation relations for transducer simulation is satisfied as well and thus $\mathcal{T}_i \preceq \mathcal{T}_i^G$ follows.

Second, we show that $s_i \models_{\mathcal{G}_i} \varphi_i$ holds, i.e., we prove that for all $\gamma \in (2^{I_i})^\omega$ and all $\gamma' \in (2^{V \setminus V_i})^\omega$, we have $comp(s_i, \gamma) \cup \gamma' \models_{\mathcal{G}_i} \varphi_i$. Let $\gamma \in (2^{I_i})^\omega$ and $\gamma' \in (2^{V \setminus V_i})^\omega$. Since $(\hat{\mathcal{S}}, \mathcal{G})$ realizes φ by assumption, either $comp(\hat{s}_i, \gamma)$ is finite or $comp(\hat{s}_i, \gamma) \cup \gamma' \models \varphi_i$ holds. If $comp(\hat{s}_i, \gamma)$ is infinite, then, by Lemma 4.3, $comp(\hat{s}_i, \gamma) = comp(s_i, \gamma)$ holds. Furthermore, we have $comp(\hat{s}_i, \gamma) \cup \gamma' \models \varphi_i$ and thus $comp(s_i, \gamma) \cup \gamma' \models \varphi_i$ follows. Hence, $comp(s_i, \gamma) \cup \gamma' \models_{\mathcal{G}_i} \varphi$ holds. Otherwise, i.e., if $comp(\hat{s}_i, \gamma)$ is finite, let $k := |comp(\hat{s}_i, \gamma)|$. By definition of local strategies, we then have $comp(\hat{s}_i, \gamma) \cdot (\gamma_k \cup o) \cup \gamma'' \notin \mathcal{H}_{k+1}^{\mathcal{G}_i}$ for all $o \in 2^{O_i}$ and all $\gamma'' \in (2^{V \setminus V_i})^*$ with $|\gamma''| = k+1$. Thus, in particular, $comp(\hat{s}_i, \gamma) \cdot (\gamma_k \cup o) \cup \gamma'_{|k+1} \notin \mathcal{H}_{k+1}^{\mathcal{G}_i}$ holds for all $o \in 2^{O_i}$. Furthermore, by Lemma 4.3, we have $comp(\hat{s}_i, \gamma)_{k'} = comp(s_i, \gamma)_{k'}$ for all points in time k' with $0 \leq k' < k$. Hence, we obtain $comp(s_i, \gamma)_{|k} = comp(\hat{s}_i, \gamma)$ and therefore $comp(s_i, \gamma)_{|k} \cdot (\gamma_k \cup o) \cup \gamma'_{|k+1} \notin \mathcal{H}_{k+1}^{\mathcal{G}_i}$ follows for all $o \in 2^{O_i}$. Thus, since we have $comp(s_i, \gamma) \cap I_i = \gamma$ by definition of computations, in particular $comp(s_i, \gamma)_{|k+1} \cup \gamma'_{|k+1} \notin \mathcal{H}_{k+1}^{\mathcal{G}_i}$ holds. Therefore, we have $comp(s_i, \gamma) \cup \gamma' \notin \mathcal{V}_{\mathcal{G}_i}$ and thus, by definition of local satisfaction, $comp(s_i, \gamma) \cup \gamma' \models_{\mathcal{G}_i} \varphi_i$ follows.

Lastly, we show that $Traces(\hat{s}_1 || \ldots || \hat{s}_n) = Traces(s_1 || \ldots || s_n)$ holds. For the sake of readability, let $\mathcal{T} = \mathcal{T}_1 \mid\mid \ldots \mid\mid \mathcal{T}_n$ and $\hat{\mathcal{T}} = \hat{\mathcal{T}_1} \mid\mid \ldots \mid\mid \hat{\mathcal{T}_n}$ as well as $\mathcal{T} = (T, T_0, \tau, \ell)$ and $\hat{\mathcal{T}} = (\hat{T}, \hat{T}_0, \hat{\tau}, \hat{\ell})$. Let $\gamma \in (2^{O_{env}})^\omega$ be some input sequence of the full system. Let $\pi \in Paths(\mathcal{T}, \gamma)$ and $\hat{\pi} \in Paths(\hat{\mathcal{T}}, \gamma)$ be the paths of $\mathcal{T}$ and $\hat{\mathcal{T}}$ induced by γ, respectively. Let $\sigma \in Traces(\mathcal{T}, \gamma)$ and $\hat{\sigma} \in Traces(\hat{\mathcal{T}}, \gamma)$ be the corresponding traces. Let $k := |\hat{\sigma}|$. Since all transducers $\mathcal{T}_i$ are deterministic and complete and have Moore semantics by construction and since the sets of output variables of different processes are disjoint by definition of architectures, $\mathcal{T}$ is deterministic and complete by Lemma 2.1 as well. Therefore, both π and σ are infinite. First, let $\pi^{G,i} \in (T_i^G \times 2^{O_i^G})^\omega$ be the sequence such that $\#_1(\pi_{k'}^{G,i}) = \#_2(\#_i(\#_1(\pi_{k'})))$ and $\#_2(\pi_{k'}^{G,i}) = \#_2(\pi_{k'}) \cap O_i^G$ holds for all $k' \geq 0$. Hence, intuitively, $\pi^{G,i}$ captures the part of π that corresponds to the guarantee for process p_i. By definition of strategy extension, every transition in $\mathcal{T}_i$ corresponds to a transition in $\mathcal{T}_i^G$. Furthermore, since $\mathcal{T}_i \preceq \mathcal{T}_i^G$ holds as shown above, the labeling of a state of $\mathcal{T}_i$ always agrees with the labeling of the guarantee part on the variables in O_i^G, no matter which case of the case distinction in the definition of the labeling function is applicable. Hence, $\pi^{G,i}$ defines a path in $\mathcal{T}_i^G$. More precisely, it follows with the definition of the parallel composition of finite-state transducers that $\pi^{G,i} \in Paths(\mathcal{T}_i^G, \sigma \cap I_i)$ holds. Let $\sigma^{G,i} \in Traces(\mathcal{T}_i^G, \sigma \cap I_i)$ be the corresponding trace. Then, $\sigma^{i,G} = comp(\mathcal{T}_i^G, \sigma \cap I_i)$ holds for all $p_i \in P^-$ since guarantee transducers are deterministic. Thus, by construction of the $\pi^{i,G}$, we have $\sigma \in \mathcal{V}_{\mathcal{G}_i}$ for all $p_i \in P^-$. Utilizing this observation, we now show that $\hat{\pi}$ is infinite and that $\#_1(\#_i(\#_1(\pi_k))) = \#_i(\#_1(\hat{\pi}_k))$ holds for all $p_i \in P^-$ and all points in time $k \geq 0$. Proof by induction on k.

- $k = 0$. By definition of strategy extension, we have $T_{i,0} = \hat{T}_{i,0} \times T_{i,0}^G$ for all $p_i \in P^-$. Thus, by definition of the parallel composition of finite-state transducers, the sets of initial states of $\hat{\mathcal{T}}$ and $\mathcal{T}$ are given by $\hat{T}_{1,0} \times \ldots \times \hat{T}_{n,0}$ and $(\hat{T}_{1,0} \times T_{1,0}^G) \times \ldots \times (\hat{T}_{n,0} \times T_{n,0}^G)$, respectively. By Proposition 4.2 we have $|\hat{T}_{i,0}| = 1$. Hence, $|\hat{\pi}| > 0$ holds and, for all $p_i \in P^-$, we have both $\#_i(\#_1(\hat{\pi}_0)) \in \hat{T}_{i,0}$ and $\#_1(\#_i(\#_1(\pi_0))) \in \hat{T}_{i,0}$. Since $|\hat{T}_{i,0}| = 1$ holds, $\#_1(\#_i(\#_1(\pi_0))) = \#_i(\#_1(\hat{\pi}_0))$ thus follows for all system processes $p_i \in P^-$.

- $k > 0$ and $\#_1(\#_i(\#_1(\pi_{k'}))) = \#_i(\#_1(\hat{\pi}_{k'}))$ holds for all $p_i \in P^-$ and all k' with $0 \leq k' < k$. For system process $p_i \in P^-$, let $\pi^i \in (\hat{T}_i \times 2^{O_i})^*$ be the finite sequence with $|\pi^i| = k$ such that both $\#_1(\pi^i_{k'}) = \#_1(\#_i(\#_1(\pi_{k'})))$ and $\#_2(\pi^i_{k'}) = \#_2(\pi_{k'}) \cap O_i$ hold for all k' with $0 \leq k' < k$. Since we have $\#_1(\#_i(\#_1(\pi_{k'}))) = \#_i(\#_1(\hat{\pi}_{k'}))$ for all $p_i \in P^-$ and all k' with $0 \leq k' < k$ by assumption, $\#_1(\#_i(\#_1(\pi_{k-1}))) \neq \bot$ holds. Hence, it follows from the definition of strategy extension that, for all $p_i \in P^-$, the finite sequence π^i is a prefix of $\rho^i \in Paths(\hat{\mathcal{T}}_i, \sigma \cap I_i)$. By definition of computations, $\sigma_{|k} \cap V_i$ is thus a prefix of $comp(\hat{s}_i, \sigma \cap I_i)$. As shown above, $\sigma \in \mathcal{V}_{\mathcal{G}_i}$ holds for all $p_i \in P^-$ and thus, in particular, we have $\sigma_{|k+1} \in \mathcal{H}_{k+1}^{\mathcal{G}_i}$ for all $p_i \in P^-$. Hence, since $\hat{s}_i$ is a local strategy for p_i and $\mathcal{G}_i$ by definition, it follows from the definition of local strategies that $|comp(\hat{s}_i, \sigma \cap I_i)| > k$ holds for all $p_i \in P^-$. Thus, for all $p_i \in P^-$, there exists a transition $(\#_1(\pi^i_{k-1}), \sigma_{k-1} \cap I_i, \#_1(\rho^i_k)) \in \hat{\tau}_i$. Therefore, there also exist transitions $(\#_i(\#_1(\pi_{k-1})), \sigma_{k-1} \cap I_i, \#_i(\#_1(\pi_k))) \in \mathcal{T}_i$ for all $p_i \in P^-$ and, in particular, $\#_1(\#_i(\#_1(\pi_k))) = \#_1(\hat{\rho}^i_k)$ holds by definition of strategy extension. By construction of π^i, we have $\#_1(\pi^i_{k-1}) = \#_1(\#_i(\#_1(\pi_{k-1})))$. Thus, $\#_1(\pi^i_{k-1}) = \#_i(\#_1(\hat{\pi}_{k-1}))$ follows with the assumption that $\#_1(\#_i(\#_1(\pi_{k-1}))) = \#_i(\#_1(\hat{\pi}_{k-1}))$ holds for all $p_i \in P^-$. By definition of paths and traces, we have $\sigma_{k-1} = \gamma_{k-1} \cup o$ and $\hat{\sigma}_{k-1} = \gamma_{k-1} \cup \hat{o}$, where $o, \hat{o} \in 2^{O_i}$ are the unique outputs of $\mathcal{T}_i$ and $\hat{\mathcal{T}}_i$ produced in state $\#_1(\pi_{k-1})$ and $\#_1(\hat{\pi}_{k-1})$, respectively. Since $\#_1(\#_i(\#_1(\pi_{k-1}))) = \#_i(\#_1(\hat{\pi}_{k-1}))$ holds for all $p_i \in P^-$ by assumption, it follows from the definition of strategy extension that $o = \hat{o}$ holds. Hence, $\sigma_{k-1} = \hat{\sigma}_{k-1}$ and therefore $(\#_i(\#_1(\hat{\pi}_{k-1})), \hat{\sigma}_{k-1} \cap I_i, \#_1(\rho^i_k)) \in \hat{\tau}_i$ for all $p_i \in P^-$ as well. Thus, by definition of strategy extension, $\#_i(\#_1(\hat{\pi}_k)) = \#_1(\rho^i_k)$ holds and hence, since $\#_1(\#_i(\#_1(\pi_k))) = \#_1(\hat{\rho}^i_k)$ as shown above, we have $\#_1(\#_i(\#_1(\pi_k))) = \#_i(\#_1(\hat{\pi}_k))$ for all $p_i \in P^-$.

Hence, we have $\#_1(\#_i(\#_1(\pi_k))) = \#_i(\#_1(\hat{\pi}_k))$ for all $p_i \in P^-$ and all points in time $k \geq 0$. Thus, by definition of the parallel composition of finite-state transducers as well as the definition of paths, we have $\#_2(\pi_k) \cap O_i = \#_2(\hat{\pi}_k) \cap O_i$ and therefore, since $\bigcup_{p_i \in P^-} O_i = O^-$ holds by definition, $\#_2(\pi_k) = \#_2(\hat{\pi}_k)$ follows. Hence, we also have $\sigma = \hat{\sigma}$ by construction of σ and $\hat{\sigma}$ as well as by definition of traces. Since we chose $\gamma \in (2^{O_{env}})^\omega$ as well as $\sigma \in Traces(\mathcal{T}, \gamma)$ and $\hat{\sigma} \in Traces(\hat{\mathcal{T}}, \gamma)$ arbitrarily, $Traces(\hat{s}_1 \mid\mid \ldots \mid\mid \hat{s}_n) = Traces(s_1 \mid\mid \ldots \mid\mid s_n)$ follows. □

Thus, since a solution of certifying synthesis with local strategies can be extended to a solution of certifying synthesis with local satisfaction, we can utilize the results from Section 4.3

to also reason about the version of certifying synthesis presented in this section. In particular, it follows immediately from Theorem 4.2 and Lemma 4.4 that certifying synthesis with local strategies and guarantee transducers is sound:

Corollary 4.1. *Let φ be an LTL formula over atomic propositions V and let $\langle \varphi_1, \dots, \varphi_n \rangle$ be its decomposition. Let $\mathcal{G} = \langle \mathcal{T}_1^G, \dots, \mathcal{T}_n^G \rangle$ be a vector of guarantee transducers for the system processes and, for $p_j \in P^-$, let $\mathcal{G}_j = \left\{ \mathcal{T}_i^G \mid p_i \in P^- \setminus \{p_j\} \right\}$. Let $\hat{\mathcal{S}} = \langle \hat{s}_1, \dots, \hat{s}_n \rangle$ be a vector of local strategies for the system processes such that $\hat{s}_i$ is a local strategy for $p_i \in P^-$ and $\mathcal{G}_i$. If $(\hat{\mathcal{S}}, \mathcal{G})$ realizes φ, then $\hat{s}_1 \mid\mid \dots \mid\mid \hat{s}_n \models \varphi$ holds.*

Vice versa, we can *restrict* a complete strategy to obtain a local strategy. The restriction is based on a set of guarantee transducers and the local strategy is restricted to those sequences that match computations of these guarantee transducers. Intuitively, the local strategy is a copy of the complete one; yet, we delete all transitions that can only be taken if some of the other (observable) system processes deviates from its certificates. Formally:

Definition 4.10 (Strategy Restriction).
Let $p_i \in P^-$ be a system process. Let $\mathcal{P} \subseteq P^- \setminus \{p_i\}$ be a set of other system processes and let $\mathcal{G}$ be a set of guarantee transducers, one for each process in $\mathcal{P}$. Let s_i be a strategy for p_i and let $\mathcal{T}_i = (T_i, T_{i,0}, \tau_i, \ell_i)$ be the deterministic and complete finite-state $(2^{I_i}, 2^{O_i})$-transducer with Moore semantics representing s_i. Let $\mathcal{T} = (T, T_0, \tau, \ell)$ be the parallel composition of the guarantee transducers in $\mathcal{G}$. We construct the *restriction* $\text{restrict}(s_i, \mathcal{G})$ *of* s_i *to* $\mathcal{G}_i$ by constructing a deterministic but possibly transition-incomplete finite-state $(2^{I_i}, 2^{O_i})$-transducer $\hat{\mathcal{T}}_i = (\hat{T}_i, \hat{T}_{i,0}, \hat{\tau}_i, \hat{\ell}_i)$ from $\mathcal{T}_i$ and $\mathcal{T}$ as follows:

- $\hat{T}_i := T_i \times 2^T$,
- $\hat{T}_{i,0} := T_{i,0} \times \{\{t_0\} \mid t_0 \in T_0\}$,
- $((t, M), \iota, (t', M')) \in \hat{\tau}_i$ if, and only if, $(t, \iota, t') \in \tau_i$ as well as $M' \neq \emptyset$ hold, and the set M' is uniquely defined by

$$\begin{aligned} M' := \Big\{ \tilde{t}' \in T \mid \exists o \in 2^{O_i}.\ \exists \tilde{t} \in M.\ \exists \tilde{o} \in 2^{O_i^G}.\ (\tilde{t}, \tilde{o}) \in \ell \ \wedge\ (t, o) \in \ell_i \\ \wedge \exists \iota' \in 2^{I_i^G}.\ \iota \cap O_i^G = \tilde{o} \cap I_i \ \wedge\ \iota' \cap O_i = o \cap I_i^G \\ \wedge\, \iota' \cap I_i = \iota \cap I_i^G \ \wedge\ (\tilde{t}, \iota', \tilde{t}') \in \tau \Big\}, \end{aligned}$$

  where $I_i^G := \bigcup_{p_j \in \mathcal{P}} I_j \setminus \bigcup_{p_j \in \mathcal{P}} O_j$ and $O_i^G = \bigcup_{p_j \in \mathcal{P}} O_j$, and
- $((t, M), o) \in \hat{\ell}_i$ if, and only if, $(t, o) \in \ell_i$

Intuitively, a computation of $\text{restrict}(s_i, \mathcal{G}_i)$ on input $\gamma \in (2^{I_i})^\omega$ follows both a computation of the complete strategy s_i on γ and a computation of the parallel composition of all guarantee transducers in $\mathcal{G}$ on some *matching* input sequence. This allows for tracking whether or not there exists some sequence $\gamma' \in (2^{V \setminus V_i})^\omega$ such that $comp(s_i, \gamma) \cup \gamma' \in \mathcal{V}_\mathcal{G}$ holds. The transducer representing restrict$(s_i, \mathcal{G}_i)$ then only contains those transitions of the transducer representing s_i that are taken in $comp(s_i, \gamma)$ up to the point in time at which the properties of a valid computation are satisfied for all $\gamma' \in (2^{V \setminus V_i})^\omega$. In the following, we show formally that restrict$(s_i, \mathcal{G})$ indeed satisfies the properties of a local strategy of p_i with respect to $\mathcal{G}$.

Lemma 4.5. *Let $p_i \in P^-$ be a system process. Let $\mathcal{P} \subseteq P^- \setminus \{p_i\}$ be a set of other system processes and let $\mathcal{G}$ be a set of guarantee transducers, one for each process in $\mathcal{P}$. Let s_i be a strategy for p_i and let $\hat{s}_i :=$ restrict$(s_i, \mathcal{G})$. Then, $\hat{s}_i$ is a local strategy for p_i with respect to $\mathcal{G}$.*

Proof. Let $I_i^G := \bigcup_{p_j \in \mathcal{P}} I_j \setminus \bigcup_{p_j \in \mathcal{P}} O_j$, let $O_i^G := \bigcup_{p_j \in \mathcal{P}} O_j$, and let $V_i^G = I_i^G \cup O_i^G$. Let $\mathcal{T}_i = (T_i, T_{i,0}, \tau_i, \ell_i)$ and $\hat{\mathcal{T}_i} = (\hat{T}_i, \hat{T}_{i,0}, \hat{\tau}_i, \hat{\ell}_i)$ be the finite-state transducers representing s_i and $\hat{s}_i$, respectively. Let $\mathcal{T} = (T, T_0, \tau, \ell)$ be the parallel composition the guarantee transducers in $\mathcal{G}$. By definition, $\mathcal{T}_i$ is a deterministic and complete Moore transducer. Thus, it follows from the definition of strategy restriction that $\hat{\mathcal{T}_i}$ is both labeling-deterministic and labeling-complete and has Moore semantics. Furthermore, for each $(t, M) \in \hat{T}_i$, there only exists a single $t' \in T_i$ such that $((t, M), \iota, (t', M')) \in \hat{\tau}_i$ holds for some $M' \in 2^T$. Hence, since M' is uniquely defined, it follows from the construction of $\hat{\mathcal{T}_i}$ that $\hat{\mathcal{T}_i}$ is transition-deterministic as well.

Next, we show that $\hat{s}_i$ satisfies the properties of a local strategy for p_i with respect to $\mathcal{G}$ regarding the finiteness and infiniteness of computations. Let $\gamma \in (2^{I_i})^\omega$ and let $\hat{\pi} \in Paths(\hat{\mathcal{T}_i}, \gamma)$ be the unique path produced by $\hat{\mathcal{T}_i}$ on input γ. Let $\hat{\sigma} \in Traces(\hat{\mathcal{T}_i}, \gamma)$ be the corresponding trace. Let $k := |\hat{\sigma}|$. First, we show a fact regarding the connection of paths of $\hat{\mathcal{T}_i}$ and paths of $\mathcal{T}$. Afterward, we utilize this result to show that $\hat{s}_i$ satisfies the properties of a local strategy.

Fact (A): For all k' with $0 \leq k' \leq k$, we have $\tilde{t} \in \#_2(\#_1(\hat{\pi}_{k'}))$ if, and only if, there exists some $\tilde{\gamma} \in (2^{I_i^G})^\omega$ and some $\tilde{\pi} \in Paths(\mathcal{T}, \tilde{\gamma})$ with $\#_1(\tilde{\pi}_{k'}) = \tilde{t}$ such that $\gamma_{k''} \cap O_i^G = (\tilde{\sigma}_{k''} \cap O_i^G) \cap I_i$, $\tilde{\gamma}_{k''} \cap O_i = (\hat{\sigma}_{k''} \cap O_i) \cap I_i^G$, and $\gamma_{k''} \cap I_i^G = \tilde{\gamma}_{k''} \cap I_i$ hold for all k'' with $0 \leq k'' < k'$, where $\tilde{\sigma} \in Traces(\mathcal{T}, \tilde{\gamma})$ is the trace corresponding to $\tilde{\pi}$. Proof by induction on the point in time k'.

- $k' = 0$. By definition of strategy restriction, we have $\hat{T}_{i,0} = T_{i,0} \times \{\{t_0\} \mid t_0 \in T_0\}$. Hence, in particular, $\#_2(\#_1(\hat{\pi}_0)) \in \{\{t_0\} \mid t_0 \in T_0\}$ holds by definition of paths. Furthermore, we have $\#_1(\tilde{\pi}_0) \in T_0$ for all $\tilde{\pi} \in Traces(\mathcal{T})$. Since $k' - 1 < 0$ holds, the claim thus follows.
- $0 < k' \leq k$ and we have $\tilde{t} \in \#_2(\#_1(\hat{\pi}_{k'-1}))$ if, and only if, there exist some $\tilde{\gamma} \in (2^{I_i^G})^\omega$ and some $\tilde{\pi} \in Paths(\mathcal{T}, \tilde{\gamma})$ with $\#_1(\tilde{\pi}_{k'}) = \tilde{t}$ such that $\gamma_{k''} \cap O_i^G = (\tilde{\sigma}_{k''} \cap O_i^G) \cap I_i$, $\tilde{\gamma}_{k''} \cap O_i = (\hat{\sigma}_{k''} \cap O_i) \cap I_i^G$, and $\gamma_{k''} \cap I_i^G = \tilde{\gamma}_{k''} \cap I_i$ hold for all k'' with $0 \leq k'' < k' - 1$, where $\tilde{\sigma} \in Traces(\mathcal{T}, \tilde{\gamma})$ is the trace corresponding to $\tilde{\pi}$. First, suppose that there exist some $\tilde{\gamma} \in (2^{I_i^G})^\omega$, some $\tilde{\pi} \in Paths(\mathcal{T}, \tilde{\gamma})$, and some $\tilde{\sigma} \in Traces(\pi, \tilde{\gamma})$ such that we have $\gamma_{k''} \cap O_i^G = (\tilde{\sigma}_{k''} \cap O_i^G) \cap I_i$, $\tilde{\gamma}_{k''} \cap O_i = (\hat{\sigma}_{k''} \cap O_i) \cap I_i^G$, and $\gamma_{k''} \cap I_i^G = \tilde{\gamma}_{k''} \cap I_i$ for all k'' with $0 \leq k'' < k' - 1$. Then, by assumption, $\#_1(\tilde{\pi}_{k'-1}) \in \#_2(\#_1(\hat{\pi}_{k'-1}))$ holds. Since guarantee transducers are complete Moore transducers and since the sets of output variables of different processes are disjoint, $\mathcal{T}$ is a complete Moore transducer by Lemma 2.1 as well. Hence, there exists some transition $(\#_1(\tilde{\pi}_{k'-1}), \tilde{\gamma}_{k'-1}, \#_1(\tilde{\pi}_{k'})) \in \tau$. By definition of paths and traces, both $(\#_1(\tilde{\pi}_{k'}), \tilde{\sigma}_{k'} \cap O_i^G) \in \ell$ and $(\#_1(\hat{\pi}_{k'}), \hat{\sigma}_{k'} \cap O_i) \in \hat{\ell}_i$ hold. Hence, we have $(\#_1(\#_1(\hat{\pi}_{k'})), \hat{\sigma}_{k'} \cap O_i) \in \ell_i$ by construction of $\hat{\mathcal{T}_i}$ as well. Therefore, it follows from the assumption as well as from the definition of strategy restriction, in particular the definition

of the set M', that $\#_1(\tilde{\pi}_{k'}) \in \#_2(\#_1(\hat{\pi}_{k'}))$ holds. Second, suppose that there do not exist $\tilde{\gamma} \in (2^{I_i^G})^\omega$, $\tilde{\pi} \in \mathit{Paths}(\mathcal{T}, \tilde{\gamma})$, and $\tilde{\sigma} \in \mathit{Traces}(\pi, \tilde{\gamma})$ such that $\gamma_{k''} \cap O_i^G = (\tilde{\sigma}_{k''} \cap O_i^G) \cap I_i$, $\tilde{\gamma}_{k''} \cap O_i = (\hat{\sigma}_{k''} \cap O_i) \cap I_i^G$, and $\gamma_{k''} \cap I_i^G = \tilde{\gamma}_{k''} \cap I_i$ hold for all k'' with $0 \leq k'' < k'-1$. Then, by assumption, we have $\#_1(\tilde{\pi}_{k'-1}) \notin \#_2(\#_1(\hat{\pi}_0))$. By definition of strategy restriction, it thus follows that $\#_1(\tilde{\pi}_{k'}) \notin \#_2(\#_1(\hat{\pi}_{k'}))$ holds.

Utilizing fact (A), we show that $\hat{s}_i$ satisfies the requirements of a local strategy for p_i and $\mathcal{G}$ regarding finiteness and infiniteness of computations, i.e., we show that if the trace $\hat{\sigma}$ is infinite, then there exists some $\gamma' \in (2^{V \setminus V_i})^\omega$ such that $\hat{\sigma} \cup \gamma' \in \mathcal{V}_\mathcal{G}$ holds and otherwise, if $\hat{\sigma}$ is finite, then we have $\hat{\sigma} \cdot (\gamma_k \cup o) \cup \gamma' \notin \mathcal{H}_{k+1}^\mathcal{G}$ for all $o \in 2^{O_i}$ and all $\gamma' \in (2^{V \setminus V_i})^*$ with $|\gamma'| = k+1$.

First, let $\hat{\sigma}$ be infinite. Then, by definition of strategy restriction, in particular $\#_2(\#_1(\hat{\pi}_{k'})) \neq \emptyset$ holds for all points in time $k' \geq 0$. Hence, it follows with fact (A) that there exist some $\tilde{\gamma} \in (2^{I_i^G})^\omega$ and some $\tilde{\pi} \in \mathit{Paths}(\mathcal{T}, \tilde{\gamma})$ with corresponding trace $\tilde{\sigma} \in \mathit{Traces}(\mathcal{T}, \tilde{\gamma})$ such that $\gamma_{k'} \cap O_i^G = (\tilde{\sigma}_{k'} \cap O_i^G) \cap I_i$, $\tilde{\gamma}_{k'} \cap O_i = (\hat{\sigma}_{k'} \cap O_i) \cap I_i^G$, and $\gamma_{k'} \cap I_i^G = \tilde{\gamma}_{k'} \cap I_i$ as well as $\#_1(\tilde{\pi}_{k'}) \in \#_2(\#_1(\hat{\pi}_{k'}))$ hold for all $k' \geq 0$. Note that $\gamma = \hat{\sigma} \cap I_i$ and $\tilde{\gamma} = \tilde{\sigma} \cap I_i^G$ hold. Therefore, $(\hat{\sigma} \cap I_i) \cap O_i^G = (\tilde{\sigma} \cap O_i^G) \cap I_i$, $(\tilde{\sigma} \cap I_i^G) \cap O_i = (\hat{\sigma} \cap O_i) \cap I_i^G$, and $(\hat{\sigma} \cap I_i) \cap I_i^G = (\tilde{\sigma} \cap I_i^G) \cap I_i$ follow. Thus, $\hat{\sigma}$ and $\tilde{\sigma}$ agree on all variables in $(I_i \cap O_i^G) \cup (O_i \cap I_i^G) \cup (I_i \cap I_i^G)$. Since $O_i \cap O_i^G = \emptyset$ follows from the definition of O_i^G as well as the disjointness of the sets of output variables of processes, $\hat{\sigma}$ and $\tilde{\sigma}$ further agree on all variables in $O_i \cap O_i^G$. Therefore, $\hat{\sigma} \cap (V_i \cap V_i^G) = \tilde{\sigma} \cap (V_i \cap V_i^G)$ follows. By construction of $\hat{\sigma}$ and $\tilde{\sigma}$, we have $\hat{\sigma} \in (2^{V_i})^\omega$ and $\tilde{\sigma} \in (2^{V_i^G})^\omega$ and hence $\hat{\sigma} \cap V_i = \sigma$ and $\tilde{\sigma} \cap V_i^G = \tilde{\sigma}$ hold. Thus, $\hat{\sigma} \cap V_i^G = \tilde{\sigma} \cap V_i$ follows and hence, in particular, $(\hat{\sigma} \cup (\tilde{\sigma} \cap (V_i^G \setminus V_i))) \cap V_i^G = \tilde{\sigma}$ holds. Since $\tilde{\sigma}$ is a trace of $\mathcal{T}$, we have $\tilde{\sigma} \cap V_j = \mathit{comp}(\mathcal{T}_j^G, \tilde{\sigma} \cap I_j)$ for all $p_j \in \mathcal{P}$ by Proposition 4.1 and thus $(\hat{\sigma} \cup (\tilde{\sigma} \cap (V_i^G \setminus V_i))) \cap V_j = \mathit{comp}(\mathcal{T}_j^G, (\hat{\sigma} \cup (\tilde{\sigma} \cap (V_i^G \setminus V_i))) \cap I_j)$ follows for all $p_j \in \mathcal{P}$ as well. Thus, $\hat{\sigma} \cup (\tilde{\sigma} \cap (V_i^G \setminus V_i)) \in \mathcal{V}_\mathcal{G}$ holds since we have $V_i \cup V_i^G = V$ by construction. Hence, there exists some $\gamma' \in (2^{V \setminus V_i})^\omega$, namely $\gamma' := \tilde{\sigma} \cap (V_i^G \setminus V_i)$, such that $\hat{\sigma} \cup \gamma' \in \mathcal{V}_\mathcal{G}$ holds.

Second, let $\hat{\sigma}$ be finite. Let $k = |\hat{\sigma}|$. By definition of strategy restriction, $\#_2(\#_1(\hat{\pi}_{k+1})) = \emptyset$ holds. Suppose that there are $o \in 2^{O_i}$ and $\gamma' \in (2^{V \setminus V_i})^*$ with $|\gamma'| = k+1$ such that $\hat{\sigma} \cdot (\gamma_k \cup o) \cup \gamma' \in \mathcal{H}_{k+1}^\mathcal{G}$ holds. Let $\rho := \hat{\sigma} \cdot (\gamma_k \cup o) \cup \gamma'_{|k+1}$. Let $\rho' \in (2^{V_i^G})^\omega$ be some infinite extension of ρ. Then, we have $\rho_{k'} \cap V_j = \mathit{comp}(\mathcal{T}_j^G, \rho' \cap I_j)_{k'} \cap V_j$ for all $p_j \in \mathcal{P}$ and all k' with $0 \leq k' < k+1$. Hence, similar to the proof of Proposition 4.1, it follows that ρ is the prefix of some trace of $\mathcal{T}$. Furthermore, clearly $\rho_{k''} \cap V_i = \hat{\sigma}_{k''} \cap V_i^G$ holds for all k'' with $0 \leq k'' < k$. Thus, in particular, $(\hat{\sigma}_{k''} \cap I_i) \cap O_i^G = (\rho_{k''} \cap O_i^G) \cap I_i$, $(\rho_{k''} \cap I_i^G) \cap O_i = (\hat{\sigma}_{k''} \cap O_i) \cap I_i^G$, and $(\hat{\sigma}_{k''} \cap I_i) \cap I_i^G = (\rho_{k''} \cap I_i^G) \cap I_i$ hold for all k'' with $0 \leq k'' < k+1$. Therefore, it follows with fact (A) that we have $\#_1(\tilde{\pi}_{k+1}) \in \#_2(\#_1(\hat{\pi}_{k+1}))$, where $\tilde{\pi} \in \mathit{Paths}(\mathcal{T}, \rho' \cap I_i^G)$ is the unique path produced by $\mathcal{T}$ on input $\rho' \cap I_i^G$; contradicting that $\#_2(\#_1(\hat{\pi}_{k+1})) = \emptyset$ holds. □

Furthermore, the computation of restrict$(s_i, \mathcal{G})$ on some input sequence $\gamma \in (2^{I_i})^\omega$ agrees with s_i's computation $\mathit{comp}(s_i, \gamma)$ if $\mathit{comp}(\text{restrict}(s_i, \mathcal{G}), \gamma)$ is infinite. Otherwise, $\mathit{comp}(s_i, \gamma)$ is an infinite extension of $\mathit{comp}(\text{restrict}(s_i, \mathcal{G}), \gamma)$. Additionally, it follows with the fact that restrict$(s_i, \mathcal{G})$ is a local strategy for p_i with respect to $\mathcal{G}$ that a computation of restrict$(s_i, \mathcal{G})$ on γ is infinite if, and only if, there exists some valuation of the variables that are unobservable for system process p_i, together with $\mathit{comp}(\text{restrict}(s_i, \mathcal{G}), \gamma)$, build a sequence that matches the guaranteed behavior of the system processes in $\mathcal{P}$:

Lemma 4.6. *Let $p_i \in P^-$ be a system process. Let $\mathcal{P} \subseteq P^- \setminus \{p_i\}$ be a set of other system processes and let $\mathcal{G}$ be a set of guarantee transducers, one for each process in $\mathcal{P}$. Let s_i be a strategy for p_i and let $\hat{s}_i := \mathrm{restrict}(s_i, \mathcal{G})$. Let $\gamma \in (2^{I_i})^\omega$. Then, $comp(\hat{s}_i, \gamma)_{k'} = comp(s_i, \gamma)_{k'}$ holds for all k' with $0 \leq k' < |comp(\hat{s}_i, \gamma)|$ and all $\gamma \in (2^{I_i})^\omega$. Furthermore, $comp(\hat{s}_i, \gamma)$ is infinite if, and only if, there exists some $\gamma' \in (2^{V \setminus V_i})^\omega$ such that $comp(s_i, \gamma) \cup \gamma' \in \mathcal{V}_\mathcal{G}$ holds.*

Proof. Let $\mathcal{T}_i = (T_i, T_{i,0}, \tau_i, \ell_i)$ and $\hat{\mathcal{T}}_i = (\hat{T}_i, \hat{T}_{i,0}, \hat{\tau}_i, \hat{\ell}_i)$ be the finite-state transducers representing s_i and $\hat{s}_i$, respectively. Let $\mathcal{T} = (T, T_0, \tau, \ell)$ be the parallel composition the guarantee transducers in $\mathcal{G}$. Let $\gamma \in (2^{I_i})^\omega$ be some input sequence. Let $\pi \in Paths(\mathcal{T}_i, \gamma)$ and $\hat{\pi} \in Paths(\hat{\mathcal{T}}_i, \gamma)$ be the unique paths produced by $\mathcal{T}_i$ and $\hat{\mathcal{T}}_i$, respectively, on γ. Let $\sigma \in Traces(\mathcal{T}_i, \gamma)$ and $\hat{\sigma} \in Traces(\hat{\mathcal{T}}_i, \gamma)$ be the corresponding unique traces. Let $k := |\hat{\sigma}|$. By definition of strategy restriction, every transition $(t, \iota, t') \in \hat{\tau}_i$ in $\hat{\mathcal{T}}_i$ is contained in $\mathcal{T}_i$ as well, i.e., $(t, \iota, t') \in \tau_i$ holds. Thus, since both $\mathcal{T}_i$ and $\hat{\mathcal{T}}_i$ are deterministic by construction and by Lemma 4.5, respectively, and thus π and $\hat{\pi}$ are unique, we have $\pi_{k'} = \hat{\pi}_{k'}$ for all points in time k' with $0 \leq k' < |\hat{\pi}|$. Hence, $comp(\hat{s}_i, \gamma)_{k'} = comp(s_i, \gamma)_{k'}$ holds for all k' with $0 \leq k' < |comp(\hat{s}_i, \gamma)|$ follows from the definition of computations and the fact that $|comp(\hat{s}_i, \gamma)| \leq |\hat{\pi}|$ holds by definition of traces.

First, let $\hat{\sigma}$ be infinite. Then, since $\hat{s}_i$ is a local strategy for p_i with respect to $\mathcal{G}$ by Lemma 4.5, there exists some $\gamma' \in (2^{V \setminus V_i})^\omega$ such that $\hat{\sigma} \cup \gamma' \in \mathcal{V}_\mathcal{G}$ holds. Furthermore, since $comp(\hat{s}_i, \gamma)$ is infinite, we have $comp(\hat{s}_i, \gamma) = comp(s_i, \gamma)$ as shown above and thus, in particular, $\sigma = \hat{\sigma}$ holds since both $\mathcal{T}_i$ and $\hat{\mathcal{T}}_i$ are deterministic by definition and by Lemma 4.5, respectively. Therefore, it follows that there exists some $\gamma' \in (2^{V \setminus V_i})^\omega$ such that $\sigma \cup \gamma' \in \mathcal{V}_\mathcal{G}$ holds.

Second, let $\hat{\sigma}$ be finite. Then, since $\hat{s}_i$ is a local strategy for p_i with respect to $\mathcal{G}$ by Lemma 4.5, we have $\hat{\sigma} \cdot (\gamma_k \cup o) \cup \gamma' \notin \mathcal{H}^\mathcal{G}_{k+1}$ for all $o \in 2^{O_i}$ and all $\gamma' \in (2^{V \setminus V_i})^*$ with $|\gamma'| = k+1$. Furthermore, we have $comp(\hat{s}_i, \gamma)_{k'} = comp(s_i, \gamma)_{k'}$ for all k' with $0 \leq k' < k$ as shown above and thus, in particular $\sigma_{k'} = \hat{\sigma}_{k'}$ for all k' with $0 \leq k' < k$ since both $\mathcal{T}_i$ and $\hat{\mathcal{T}}_i$ are deterministic by definition and by Lemma 4.5, respectively. Hence, we have $\sigma \cdot (\gamma_k \cup o) \cup \gamma' \notin \mathcal{H}^\mathcal{G}_{k+1}$ for all $o \in 2^{O_i}$ and all $\gamma' \in (2^{V \setminus V_i})^*$ with $|\gamma'| = k + 1$ and thus $\sigma \cup \gamma' \notin \mathcal{V}_\mathcal{G}$ follows for all $\gamma' \in (2^{V \setminus V_i})^\omega$. □

The existence of a transition in the finite-state transducer $\hat{\mathcal{T}}_i$ representing $\hat{s}_i := \mathrm{restrict}(s_i, \mathcal{G}_i)$, however, might also depend on *unobservable behavior* of other system processes. The valuation of variables outside of V_i cannot be observed by process p_i. However, whether or not a transition of the transducer $\mathcal{T}_i$ representing thee full strategy s_i is also contained in $\hat{\mathcal{T}}_i$ does not only depend on the existence of an input sequence $\gamma \in (2^{I_i})^\omega$ and s_i's behavior on γ but also on the existence of a sequence $\gamma' \in (2^{V \setminus V_i})^\omega$. Hence, whenever there is *some* unobservable behavior of the other system processes that matches their guaranteed behavior, a transition from $\mathcal{T}_i$ is preserved in $\hat{\mathcal{T}}_i$ and thus, as shown in Lemma 4.6, the computations of s_i and $\hat{s}_i$ coincide on γ.

When determining whether or not $\hat{s}_i$ realizes an LTL formula φ_i, we determine whether for all $\gamma \in (2^{I_i})^\omega$ and all $\gamma' \in (2^{V \setminus V_i})^\omega$ either $comp(\hat{s}_i, \gamma)$ is finite or $comp(\hat{s}_i, \gamma) \cup \gamma' \models \varphi_i$ holds. Hence, we consider a concrete sequence γ' of valuations of unobservable variables. However, whether $comp(\hat{s}_i, \gamma)$ is infinite only depends on the existence of *some* sequence of unobservable variables, not the concretely considered one. Thus, $comp(\hat{s}_i, \gamma)$ might be infinite although $comp(\hat{s}_i, \gamma) \cup \gamma' \notin \mathcal{V}_{\mathcal{G}_i}$ holds. Hence, requiring $\hat{s}_i \models \varphi_i$ then also requires $comp(\hat{s}_i, \gamma) \cup \gamma' \models \varphi_i$ to hold although $comp(\hat{s}_i, \gamma) \cup \gamma' \notin \mathcal{V}_{\mathcal{G}_i}$. This is in contrast to local satisfaction, where $comp(\hat{s}_i, \gamma) \cup \gamma' \models_{\mathcal{G}_i} \varphi_i$ holds if we have $comp(\hat{s}_i, \gamma) \cup \gamma' \notin \mathcal{V}_{\mathcal{G}_i}$.

Therefore, not every solution of certifying synthesis with local satisfaction can be translated into one for certifying synthesis with local strategies when utilizing strategy restriction as defined in Definition 4.10. The former requires a strategy to realize the specification if *all other system processes* do not deviate from their guaranteed behavior. The latter, in contrast, requires a strategy to satisfy the specification if *the observable behavior of all other system processes* does not deviate from their guaranteed behavior. Thus, if system process $p_i \in P^-$ cannot observe whether or not another system process $p_j \in P^- \setminus \{p_i\}$ deviates from its certificate, satisfaction with local strategies requires the strategy to satisfy the specification while local satisfaction for complete strategies does not. However, as long as the satisfaction of the specification does not depend on unobservable variables, i.e., as long as $prop(\varphi_i) \subseteq V_i$ holds for all system processes $p_i \in P^-$, then the existence of some sequence $\gamma' \in (2^{V\setminus V_i})^\omega$ such that $comp(s_i, \gamma) \cup \gamma' \models \varphi_i$ holds implies that $comp(s_i, \gamma) \cup \gamma' \models \varphi_i$ holds for *all* such sequences γ'. Hence, satisfaction of local strategies can be concluded from local satisfaction for complete strategies when utilizing strategy restriction for obtaining the local strategies:

Lemma 4.7. *Let φ be an LTL formula over atomic propositions V with decomposition $\langle \varphi_1, \dots, \varphi_n \rangle$. Let $\mathcal{S} = \langle s_1, \dots, s_n \rangle$ and $\mathcal{G} = \langle \mathcal{T}_1^G, \dots, \mathcal{T}_n^G \rangle$ be vectors of strategies and guarantee transducers for the system processes. For $p_j \in P^-$, let $\mathcal{G}_j = \left\{ \mathcal{T}_i^G \mid p_i \in P^- \setminus \{p_j\} \right\}$. Let $\hat{\mathcal{S}} = \langle \hat{s}_1, \dots, \hat{s}_n \rangle$ such that $\hat{s}_i = \mathrm{restrict}(s_i, \mathcal{G}_i)$ holds for all $p_i \in P^-$. If $(\mathcal{S}, \mathcal{G})$ realizes φ and if $prop(\varphi_i) \subseteq V_i$ holds for all $p_i \in P^-$, then $(\hat{\mathcal{S}}, \mathcal{G})$ realizes φ as well.*

Proof. Let $\hat{\mathcal{T}}_i = (\hat{T}_i, \hat{t}_{i,0}, \hat{\tau}_i, \ell_i)$ and $\mathcal{T}_i = (T_i, t_{i,0}, \tau_i, \ell_i)$ be the finite-state transducers representing $\hat{s}_i$ and s_i, respectively. Let $\mathcal{T}_i^G = (T_i^G, t_{i,0}^G, \tau_i^G, \ell_i^G)$. Assume that $(\hat{\mathcal{S}}, \mathcal{G})$ realizes φ. Then, by definition of certifying synthesis with guarantee transducers and local satisfaction, we have both $s_i \models_{\mathcal{G}_i} \varphi_i$ and $\mathcal{T}_i \preceq \mathcal{T}_i^G$ for all $p_i \in P^-$. To prove that $(\hat{\mathcal{S}}, \mathcal{G})$ realizes φ as well, we show that both $\hat{s}_i \models \varphi_i$ and $\hat{\mathcal{T}}_i \preceq \mathcal{T}_i^G$ hold for all $p_i \in P^-$. Let $p_i \in P^-$ be some system process.

First, we show that $\hat{\mathcal{T}}_i \preceq \mathcal{T}_i^G$ holds. Since $\mathcal{T}_i \preceq \mathcal{T}_i^G$ holds by assumption, there exists a simulation relation $R : T_i \times T_i^G$ that establishes that $\mathcal{T}_i^G$ simulates $\mathcal{T}_i$. We construct a simulation relation $\hat{R} : \hat{T}_i \times T_i^G$ that establishes $\hat{\mathcal{T}}_i \preceq \mathcal{T}_i^G$ as follows: $((t, M), t^G) \in \hat{R}$ holds if, and only if, we have $(t, t^G) \in R$. Since $t_0 \in T_{i,0}$ holds for all $(t_0, M_0) \in \hat{T}_{i,0}$ by definition of strategy restriction, it follows immediately from the construction of $\hat{R}$ and the fact that R is a valid simulation relation that $(\hat{t}_0, t_0^G) \in \hat{R}$ holds for all $\hat{t}_0 \in \hat{T}_{i,0}$ and all $t_0^G \in T_{i,0}^G$. If $((t, M), \iota, (t', M')) \in \hat{\tau}_i$ holds, then, by construction of $\hat{\mathcal{T}}_i$, we have $(t, \iota, t') \in \tau_i$ as well. Hence, since $\mathcal{T}_i \preceq \mathcal{T}_i^G$ holds by assumption, it follows from the construction of $\hat{R}$ that the second requirement of simulation relations for transducer simulation is satisfied by $\hat{R}$ as well. Thus, $\hat{\mathcal{T}}_i \preceq \mathcal{T}_i^G$ holds.

Second, we show that $\hat{s}_i \models \varphi_i$ holds, i.e., we prove that for all $\gamma \in (2^{I_i})^\omega$ and all $\gamma' \in (2^{V\setminus V_i})^\omega$, either $comp(\hat{s}_i, \gamma)$ is finite or $comp(\hat{s}_i, \gamma) \cup \gamma' \models \varphi$ holds. Let $\gamma \in (2^{I_i})^\omega$ and $\gamma' \in (2^{V\setminus V_i})^\omega$. Since we have $s_i \models_{\mathcal{G}_i} \varphi_i$ by assumption, $comp(s_i, \gamma) \cup \gamma' \models_{\mathcal{G}_i} \varphi_i$ holds. Thus, we have either $comp(s_i, \gamma) \cup \gamma' \notin \mathcal{V}_{\mathcal{G}_i}$ or both $comp(s_i, \gamma) \cup \gamma' \in \mathcal{V}_{\mathcal{G}_i}$ and $comp(s_i, \gamma) \cup \gamma' \models \varphi_i$ hold. If the latter holds, then it follows with Lemma 4.6 that $comp(\hat{s}_i, \gamma)$ is infinite and that we have $comp(\hat{s}_i, \gamma) = comp(s_i, \gamma)$. Since $comp(s_i, \gamma) \cup \gamma' \models \varphi_i$ holds by assumption, $comp(\hat{s}_i, \gamma) \cup \gamma' \models \varphi_i$ thus holds as well. If $comp(s_i, \gamma) \cup \gamma' \notin \mathcal{V}_{\mathcal{G}_i}$ holds, then, we distinguish two cases. If $comp(s_i, \gamma) \cup \gamma'' \notin \mathcal{V}_{\mathcal{G}_i}$ holds for all $\gamma' \in (2^{V\setminus V_i})^\omega$, then $comp(\hat{s}_i, \gamma)$ is finite by Lemma 4.6. Otherwise, i.e., if there is some $\gamma'' \in (2^{V \setminus V_i})^\omega$ such that $comp(s_i, \gamma) \cup \gamma'' \in \mathcal{V}_{\mathcal{G}_i}$ holds, then $comp(\hat{s}_i, \gamma) = comp(s_i, \gamma)$ holds by Lemma 4.6. Furthermore, since both $comp(s_i, \gamma) \cup \gamma'' \in \mathcal{V}_{\mathcal{G}_i}$ and $s_i \models_{\mathcal{G}_i} \varphi_i$ hold by assumption, we have $comp(s_i, \gamma) \cup \gamma'' \models \varphi_i$. Moreover, $prop(\varphi_i) \subseteq V_i$ holds by assumption and therefore the satisfaction of φ_i does not depend on the variables in $V \setminus V_i$. Thus, the satisfaction of φ_i is independent of γ''. Hence, it follows that $comp(s_i, \gamma) \cup \gamma' \models \varphi_i$ holds for γ' as well. Since we have $comp(\hat{s}_i, \gamma) = comp(s_i, \gamma)$ as shown above, $comp(\hat{s}_i, \gamma) \cup \gamma' \models \varphi_i$ follows. Hence, we have shown that in both cases either $comp(\hat{s}_i, \gamma)$ is finite or $comp(\hat{s}_i, \gamma) \cup \gamma' \models \varphi_i$ holds. □

Thus, since a solution of certifying synthesis with local satisfaction and complete strategies can be restricted to a solution of certifying synthesis with local strategies as long as $prop(\varphi_i) \subseteq V_i$ holds for all $p_i \in P^-$, we can utilize the results from Section 4.3 to reason about certifying synthesis with local strategies. In particular, it follows from Theorem 4.2 and Lemma 4.7 that certifying synthesis with local strategies is complete if $prop(\varphi_i) \subseteq V_i$ holds for all $p_i \in P^-$:

Corollary 4.2. *Let φ be an LTL formula over atomic propositions V and let $\langle \varphi_1, \ldots, \varphi_n \rangle$ be its decomposition. Let $\mathcal{S} = \langle s_1, \ldots, s_n \rangle$ be a vector of strategies for the system processes. If, for all $p_i \in P^-$, both $prop(\varphi_i) \subseteq V_i$ and $s_1 \mid\mid \ldots \mid\mid s_n \models \varphi$ hold, then there exist vectors $\mathcal{G} = \langle \mathcal{T}_1^G, \ldots, \mathcal{T}_n^G \rangle$ and $\hat{\mathcal{S}} = \langle \hat{s}_1, \ldots, \hat{s}_n \rangle$ of guarantee transducers and local strategies such that $\hat{s}_i$ is a local strategy for $p_i \in P^-$ and $\mathcal{G}_i$, where $\mathcal{G}_i = \left\{ \mathcal{T}_j^G \mid p_j \in P^- \setminus \{p_i\} \right\}$, and such that $(\hat{\mathcal{S}}, \mathcal{G})$ realizes φ.*

The slight difference between local strategies and local satisfaction yielding only conditional completeness for certifying synthesis with local strategies is needed in order to technically incorporate the requirements of certifying synthesis into the strategy and thus to be able to reuse existing bounded synthesis frameworks. Although this is at general completenesses expanse, we experienced that, in practice, many distributed systems indeed satisfy the condition that is needed for completeness, i.e., that $prop(\varphi_i) \subseteq V_i$ holds for all $p_i \in P^-$. In fact, all benchmarks described in Section 4.7 satisfy it.

Therefore, we utilize local strategies for certifying synthesis in the remainder of this chapter. For practically synthesizing solutions for certifying synthesis, it is thus crucial to formalize local strategies and, in particular, to identify valid computations only to construct strategies that adhere to the definition of local strategies. In the following section, we present how this identification can be carried out before we introduce the SAT encoding for certifying synthesis with local strategies and guarantee transducers in Section 4.4.4.

4.4.3. Identifying Valid Computations

For synthesizing local strategies and guarantee transducers that satisfy the requirements of certifying synthesis, it is crucial to determine whether an infinite sequence is a valid computation with respect to a set of guarantee transducers as valid computations play a major role in the definition of local strategies. To ensure that the encoding searches for Moore transducers that adhere to the definition of local strategies, we thus need to identify whether computations of transition-incomplete Moore transducers allow for valid computations.

To do so, we *augment* a local strategy for system process $p_i \in P^-$ with additional *associated outputs*. Such an associated output is a variable that is an input variable of p_i and an output variables of some other system process $p_j \in P^- \setminus \{p_i\}$, i.e., the set O_i^A of associated outputs of p_i is defined by $O_i^A = I_i \cap \bigcup_{p_j \in P^- \setminus \{p_i\}} O_j$. An *augmented local strategy* then does not only produce a valuation of output variables of p_i but of outputs and associated outputs of p_i. Furthermore, we ensure that, intuitively, the sequence of associated outputs produced by an augmented local strategy always matches the sequences produced by the guarantee transducers of the other system processes. Lastly, an augmented local strategy has a transition with source state t and input ι if, and only if, the valuations of the associated outputs produced by the augmented local strategy in t match their valuations in ι. Intuitively, an augmented local strategy $\tilde{s}_i$ thus only produces outputs on some input sequence $\gamma \in (2^{I_i})^\omega$ as long as the prefix of $comp(\tilde{s}_i, \gamma) \cup \gamma'$ up to the current point in time is a valid history with respect to $\mathcal{G}_i$ for some $\gamma' \in (2^{V \setminus V_i})^\omega$. Thus, in particular, $\tilde{s}_i$ satisfies the same requirements regarding finiteness and infiniteness of computations as local strategies. That is, given some input sequence $\gamma \in (2^{I_i})^\omega$, if $comp(\tilde{s}_i, \gamma)$ is infinite, then there exists some sequence $\gamma' \in (2^{V \setminus V_i})^\omega$ such that $comp(\tilde{s}_i, \gamma) \cup \gamma' \in \mathcal{V}_{\mathcal{G}_i}$ holds. If $comp(\tilde{s}_i, \gamma)$ is finite and of length k, in contrast, then we have $comp(\tilde{s}_i, \gamma) \cdot (\gamma_k \cup o) \cup \gamma' \notin \mathcal{H}_{k+1}^{\mathcal{G}_i}$ for all $o \in 2^{O_i}$ and all $\gamma' \in (2^{V \setminus V_i})^*$ with $|\gamma'| = k + 1$.

Note that, although local strategies and thus also augmented local strategies can be transition-incomplete, they are always labeling-deterministic and labeling-complete. Moreover, they have Moore semantics. For the sake of readability, we thus depict their labeling relations as *functions* that map a state to a unique valuation of output variables instead of relations in the remainder of this section. Formally, we define augmented local strategies as follows:

Definition 4.11 (Augmented Local Strategy)**.**
Let $p_i \in P^-$ be some system process. Let $\mathcal{P} \subseteq P^- \setminus \{p_i\}$ be a set of other system processes and let $\mathcal{G}$ be a set of guarantee transducers, one for each process in $\mathcal{P}$. An *augmented local strategy* $s_i : (2^{V_i})^* \times 2^{I_i} \to 2^{O_i \cup O_i^A}$ for p_i with respect to $\mathcal{G}$ is represented by a deterministic finite-state $(2^{I_i}, 2^{O_i \cup O_i^A})$-transducer $\mathcal{T}_i = (T, T_0, \tau, \ell)$ with Moore semantics. It holds that (i) for every $t \in T$ and every $\iota \in 2^{I_i}$, there is some $t' \in T$ with $(t, \iota, t') \in \tau$ if, and only if $\ell(t) \cap O_i^A = \iota \cap O_i^A$ holds, and (ii) for every $p_j \in P^- \setminus \{p_i\}$ with guarantee transducer $\mathcal{T}_j^G = (T_j^G, T_{j,0}^G, \tau_j^G, \ell_j^G)$ such that $\mathcal{T}_j^G \in \mathcal{G}$, there exists a relation $S_j^i : T_j^G \times T$ such that

- $(t_0^G, t_0) \in S_j^i$ for all $t_0^G \in T_0^G$ and all $t_0 \in T_i$, and
- for all $(t^G, t) \in S_j^i$, we have $\ell_j^G(t^G) \cap O_i^A = \ell(t) \cap O_j^G$ and, for all $\iota \in 2^{I_j}$, $\iota' \in 2^{I_i}$ with $\iota \cap I_i = \iota' \cap I_j$ as well as $\ell_i(t) \cap O_i^A = \iota' \cap O_i^A$, and all $t^{G'} \in T_j^G$, if $(t^G, \iota, t^{G'}) \in \tau_j^G$ holds, then there exists some $t' \in T_i$ such that $(t, \iota', t') \in \tau$ and $(t^{G'}, t') \in S_j^i$ hold.

Although the sets of inputs and outputs of an augmented local strategy are not disjoint, requirement (i) ensures that no paths with contradictory valuations of shared variables can be produced. Therefore, traces of augmented local strategies are well-defined.

Furthermore, note that the relations S^j which are required in the definition of augmented local strategies resemble simulation relations as defined for transducer simulation in Definition 4.5. However, they do not require that the transducers have the same input variables but consider all combinations of valuations of inputs that agree on shared input variables. Similarly, they do not require that the output variables of $\mathcal{T}_i^G$ are a subset of the outputs of $\hat{\mathcal{T}_i}$ but only require that the two transducers agree on all shared output variables. The relations S^j can thus be seen as a more general version of transducer simulation. However, the definition of the relations S^j further poses the additional restriction that $\ell_i(t) \cap O_i^A = \iota' \cap O_i^A$ holds in the second requirement. This is due to the fact $\mathcal{T}_i^G$ is a complete Moore transducers while $\hat{\mathcal{T}_i}$ is not. When omitting this restriction, then the second requirement would ensure that $\hat{\mathcal{T}_i}$ is complete as well as $\hat{\mathcal{T}_i}$ would need to provide a matching transition for *all* transitions in $\mathcal{T}_i^G$, even if the considered sequence does not match a valid computation anymore. This would contradict that augmented local strategies only have a transition if the input matches the associated outputs. Therefore, we add the additional restriction that ensures that we only require a matching transition in $\hat{\mathcal{T}_i}$ if the input indeed matches the associated outputs. We first show that augmenting a local strategy with associated outputs indeed allows for determining whether or not the properties of local strategies are satisfied. In particular, we show that an augmented local strategy $\tilde{s}_i$ satisfies the same requirements as local strategies regarding finite and infinite computations:

Lemma 4.8. *Let $p_i \in P^-$ be some system process. Let $\mathcal{P} \subseteq P^- \setminus \{p_i\}$ be a set of other system processes and let $\mathcal{G}$ be a set of guarantee transducers, one for each process in $\mathcal{P}$. Let $\tilde{s}_i$ be an augmented local strategy for $p_i \in P^-$ with respect to $\mathcal{G}$. Let $\tilde{\mathcal{T}_i}$ be the finite-state transducer representing $\tilde{s}_i$. Then, for all $\gamma \in (2^{I_i})^\omega$ and all $\sigma \in Traces(\tilde{\mathcal{T}_i}, \gamma)$, it holds that (i) if σ is infinite, then there exists some $\gamma' \in (2^{V \setminus V_i})^\omega$ such that $\sigma \cup \gamma' \in \mathcal{V}_\mathcal{G}$ holds, and (ii) if σ is finite, then $\sigma \cdot (\gamma_{|\sigma|} \cup o) \cup \gamma' \notin \mathcal{H}^\mathcal{G}_{|\sigma|+1}$ holds for all $o \in 2^{O_i}$ and all $\gamma' \in (2^{V \setminus V_i})^\omega$ with $|\gamma'| = |\sigma| + 1$.*

Proof. For $p_j \in \mathcal{P}$, let $\mathcal{T}_j^G = (T_j^G, T_{j,0}^G, \tau_j^G, \ell_j^G)$ be the guarantee transducer contained in $\mathcal{G}$ and let S_j^i be the relation establishing that $\tilde{s}_i$ is an augmented local strategy with respect to p_j's guarantee transducer $\mathcal{T}_j^G$. Let $\tilde{\mathcal{T}_i} = (\tilde{T}_i, \tilde{T}_{i,0}, \tilde{\tau}_i, \tilde{\ell}_i)$. Let $\gamma \in (2^{I_i})^\omega$ and let $\tilde{\pi} \in Paths(\tilde{\mathcal{T}_i}, \gamma)$ be the unique path produced by $\tilde{\mathcal{T}_i}$ on input γ. Let $\tilde{\sigma} \in Traces(\tilde{\mathcal{T}_i}, \gamma)$ be the corresponding trace. Let $k := |\tilde{\pi}|$. Let $\rho \in (2^V)^\omega$ be some infinite sequence such that $\rho \cap V_i = \tilde{\sigma}$ holds if $\tilde{\sigma}$ is infinite and such that both $\rho \cap I_i = \gamma$ and $\rho_{k'} \cap V_i = \tilde{\sigma}_{k'} \cap V_i$ holds for all points in time k' with $0 \leq k' < |\tilde{\sigma}|$ otherwise. We first show a fact on the relationship of traces of $\tilde{\mathcal{T}_i}$ and valid computations. Afterward, we utilize this result to show that $\tilde{\mathcal{T}_i}$ satisfies the properties of a local strategy regarding infinite computations.

Fact (A): For all $p_j \in \mathcal{P}$, we have $(\#_1(\pi^j_{k'}), \#_1(\tilde{\pi}_{k'})) \in S_j^i$ for all points in time k' with $0 \leq k' < k$, where $\pi^j \in Paths(\mathcal{T}_j^G, \rho \cap I_j)$. Proof by induction on k'.

- $k' = 0$. By definition of the relations S_j^i, we have, for all $p_j \in \mathcal{P}$, that $(t_{j,0}^G, t_{i,0}) \in S_j^i$ holds for all $t_{j,0}^G \in T_{j,0}^G$ and all $t_{i,0} \in T_{i,0}$. Thus, since $\#_1(\pi_0^j) \in T_{j,0}^G$ and $\#_1(\tilde{\pi}_0) \in T_{i,0}$ hold by definition of paths, we have $(\#_1(\pi_0^j), \#_1(\tilde{\pi}_0)) \in S_j^i$.
- $0 < k' < k$ and $(\#_1(\pi^j_{k'-1}), \#_1(\tilde{\pi}_{k'-1})) \in S_j^i$ holds for all $p_j \in \mathcal{P}$. Since guarantee transducers are complete, there exist transitions $(\#_1(\pi^j_{k'-1}), \rho_{k'-1} \cap I_j, \#_1(\pi^j_{k'})) \in \tau_j^G$ for all processes $p_j \in \mathcal{P}$. Furthermore, since $k' < k$ holds by assumption, there exists a transition $(\#_1(\tilde{\pi}_{k'-1}), \gamma_{k'-1}, \#_1(\tilde{\pi}_{k'})) \in \tilde{\tau}_i$. Moreover, since $k' < k$, we have $\rho_{k'-1} \cap V_i = \tilde{\sigma}_{k'-1}$ and thus, in particular, $(\rho_{k'-1} \cap I_j) \cap I_i = \gamma_{k'-1} \cap I_j$ holds. Hence, by condition (i) of the

definition of augmented local strategies, we have $\tilde{\ell}(\#_1(\tilde{\pi}_{k'-1})) \cap O_i^A = \gamma_{k'} \cap O_i^A$. Since $(\#_1(\pi^j_{k'-1}), \#_1(\tilde{\pi}_{k'-1})) \in S^i_j$ holds for all $p_j \in \mathcal{P}$ by assumption, it thus follows with the definition of the relations S^i_j that $(\#_1(\pi^j_{k'}), \#_1(\tilde{\pi}_{k'})) \in S^i_j$ holds for all $p_j \in \mathcal{P}$.

Utilizing fact (A), we now show that $\tilde{\mathcal{T}_i}$ indeed satisfies the properties of a local strategy regarding (in)finiteness of computations.

First, let $\tilde{\pi}$ be infinite. Then, $\tilde{\sigma}$ is infinite as well. For $p_j \in \mathcal{P}$, let $\pi^j \in \mathit{Paths}(\mathcal{T}^G_j, \tilde{\sigma} \cap I_j)$ and let $\sigma^j \in \mathit{Traces}(\mathcal{T}^G_j, \tilde{\sigma} \cap I_j)$ be the corresponding trace Then, it follows with fact (A) that $(\#_1(\pi^j_k), \#_1(\tilde{\pi}_k)) \in S^i_j$ holds for all $k \geq 0$. Thus, by definition of the relations S^i_j, in particular $\ell^G_j(\#_1(\pi^j_k)) \cap O^A_i = \tilde{\ell}_i(\#_1(\tilde{\pi}_k)) \cap O^G_j$ holds for all $p_j \in \mathcal{P}$ and all $k \geq 0$. Therefore, by definition of paths, $\#_2(\pi^j_k) \cap O^A_i = \#_2(\tilde{\pi}_k) \cap O^G_j$ holds for all $p_j \in \mathcal{P}$ and all $k \geq 0$. Thus, with the definition of traces $(\sigma^j \cap O^G_j) \cap O^A_i = (\tilde{\sigma} \cap O^A_i) \cap O^G_j$ follows for all $p_j \in \mathcal{P}$. Let $\gamma' \in (2^{V \setminus V_i})^\omega$ be an infinite sequence such that $\gamma' \cap O^G_j = \sigma^j \cap (O^G_j \setminus V_i)$ holds for all $p_j \in \mathcal{P}$. Since the sets of output variables of different processes are disjoint by definition, $\gamma' \cap O^G_j = \sigma^j \cap (O^G_j \setminus O^A_i)$ follows for all $p_j \in \mathcal{P}$ with the definition of associated outputs. Thus, we obtain $\sigma^j \cap O^G_j = (\tilde{\sigma} \cup \gamma') \cap O^G_j$ and hence $(\tilde{\sigma} \cup \gamma') \cap O^G_j = \mathit{comp}(\mathcal{T}^G_j, \rho \cap I_j)$ follows for all $p_j \in \mathcal{P}$ with the definition of computations and the construction of σ^j. Therefore $\tilde{\sigma} \cup \gamma' \in \mathcal{V}_\mathcal{G}$ holds.

Second, let $\tilde{\pi}$ be finite. Then, by condition (i) of the definition of augmented local strategies, we have $\tilde{\ell}_i(\#_1(\tilde{\pi}_{k-1})) \cap O^A_i \neq \gamma_{k-1} \cap O^A_i$. Furthermore, we have $|\tilde{\sigma}| = \max\{0, k-1\}$. Let $\rho \in (2^V)^\omega$ be some sequence such that both $\rho \cap I_i = \gamma$ and $\rho_{k'} \cap V_i = \tilde{\sigma}_{k'} \cap V_i$ hold for all k' with $0 \leq k' < |\tilde{\sigma}|$. For $p_j \in \mathcal{P}$, let $\pi^j \in \mathit{Paths}(\mathcal{T}^G_j, \rho \cap I_j)$. Then, it follows with fact (A) that $(\#_1(\pi^j_{k'}), \#_1(\tilde{\pi}_{k'})) \in S^i_j$ holds for all k' with $0 \leq k' < k$ and for all $p_j \in \mathcal{P}$. Thus, we have $(\#_1(\pi^j_{k-1}), \#_1(\tilde{\pi}_{k-1})) \in S^i_j$. By definition of the S^i_j, we have $\ell^G_j(\#_1(\pi^j_{k-1})) \cap O^A_i = \tilde{\ell}_i(\#_1(\tilde{\pi}_{k-1})) \cap O^G_j$ for all $p_j \in \mathcal{P}$. Since we have $\tilde{\ell}_i(\#_1(\tilde{\pi}_{k-1})) \cap O^A_i \neq \gamma_{k-1} \cap O^A_i$ as shown above, $\ell^G_j(\#_1(\pi^j_{k-1})) \cap O^A_i \neq (\gamma_{k-1} \cap O^A_i) \cap O^G_j$ follows. By definition of associated outputs and by construction of γ, thus $\ell^G_j(\#_1(\pi^j_{k-1})) \cap I_i \neq \gamma_{k-1} \cap O^G_j$ holds. Hence, we have $\#_2(\pi^j_{k-1}) \cap I_i \neq \gamma_{k-1} \cap O^G_j$ by definition of paths and hence it follows with the definition of computations that $(\mathit{comp}(\mathcal{T}^G_j, \rho \cap I_j) \cap O_j) \cap I_i \neq \gamma_{k-1} \cap O^G_j$ holds for all $p_j \in \mathcal{P}$. Therefore, we have $\tilde{\sigma} \cdot (\gamma_{k-1} \cup o) \cup \gamma' \notin \mathcal{H}^\mathcal{G}_k$ for all $o \in 2^{O_i}$ and all $\gamma' \in (2^{V \setminus V_i})^\omega$. □

We now define certifying synthesis with augmented local strategies and guarantee transducers. It is similar to certifying synthesis with local strategies and guarantee transducers as presented in Definition 4.8 but uses augmented local strategies instead of local ones. Note that when posing the requirements such as satisfaction of the subspecification and that the strategy is simulated by the guarantee transducer, we do not use the augmented local strategy itself but a slightly modified version: local$(\tilde{s}_i)$ denotes the restriction of the augmented local strategy $\tilde{s}_i$ to the output variables of process p_i. That is, local$(\tilde{s}_i)$ is represented by a finite-state transducer that is a copy of the finite-state transducer $\tilde{\mathcal{T}_i} = (\tilde{T}_i, \tilde{T}_{i,0}, \tilde{\tau}_i, \tilde{\ell}_i)$ representing $\tilde{s}_i$, but modifies the labeling function to $\tilde{\ell}_i(t) \cap O_i$. Since $\tilde{s}_i$ satisfies the properties of a local strategy regarding finiteness and infiniteness of computations by Lemma 4.8, local$(\tilde{s}_i)$ is a local strategy. Furthermore, since condition (i) of the definition of augmented local strategies ensures that augmented local strategies produce well-defined computations only, the computations of $\tilde{s}_i$ and local$(\tilde{s}_i)$ coincide for all input sequences. Thus, it is not necessary to use local$(\tilde{s}_i)$ instead of $\tilde{s}_i$ for stating the satisfaction of the subspecification. The definition of transducer simulation, however, requires that the set of output variables of the transducer which is simulated by another transducer needs to be a subset if the set of outputs of the simulating transducer. Hence, since $O_i \cup O_i^A \subseteq O_i^G$ does not hold if $O_i^A \neq \emptyset$, stating that a strategy does not deviate from its own guaranteed behavior requires using local$(\tilde{s}_i)$. For ease of presentation, we then use local$(\tilde{s}_i)$ in the definition of certifying synthesis with augmented local strategies for both requirements.

Definition 4.12 (Certifying Synthesis with Augmented Local Strategies)**.**
Let φ be an LTL formula over atomic propositions V with decomposition $\langle \varphi_1, \dots, \varphi_n \rangle$. Let $\mathcal{G} = \langle \mathcal{T}_1^G, \dots, \mathcal{T}_n^G \rangle$ be a vector of guarantee transducers for the system processes. For $p_j \in P^-$, let $\mathcal{G}_j = \left\{ \mathcal{T}_i^G \mid p_i \in P^- \setminus \{p_j\} \right\}$. Let $\tilde{\mathcal{S}} = \langle \tilde{s}_1, \dots, \tilde{s}_n \rangle$ such that $\tilde{s}_i$ is an augmented local strategy for $p_i \in P^-$ with respect to $\mathcal{G}_i$. Let $\hat{\mathcal{T}}_i$ be the deterministic and complete finite-state transducer representing local$(\tilde{s}_i)$. If, for all $p_i \in P^-$, both $\hat{\mathcal{T}}_i \preceq \mathcal{T}_i^G$ and, for all $\gamma \in (2^{I_i})^\omega$, $\gamma' \in (2^{V \setminus V_i})^\omega$, either $comp(\text{local}(\tilde{s}_i), \gamma)$ is finite or $comp(\text{local}(\tilde{s}_i), \gamma) \cup \gamma' \models \varphi$ holds, then we say that $(\tilde{\mathcal{S}}, \mathcal{G})$ realizes φ. *Certifying synthesis* for φ derives vectors $\tilde{\mathcal{S}}$ and $\mathcal{G}$ such that $(\tilde{\mathcal{S}}, \mathcal{G})$ realizes φ.

In the following, we prove soundness and completeness of certifying synthesis with augmented local strategies. Since the requirements of certifying synthesis are posed on the strategies local$(\tilde{s}_i)$ instead of the strategies $\tilde{s}_i$, it follows immediately from Lemma 4.8 and Corollary 4.1 that the parallel composition of the strategies local$(\tilde{s}_i)$ satisfies the specification φ if there exist augmented local strategies $\tilde{s}_i$ and guarantee transducers $\mathcal{T}_i^G$ that constitute a solution of certifying synthesis. Soundness then follows with the observation that the parallel composition of the strategies local$(\tilde{s}_i)$ and $\tilde{s}_i$ coincide:

Lemma 4.9. *Let φ be an LTL formula over atomic propositions V and let $\langle \varphi_1, \dots, \varphi_n \rangle$ be its decomposition. Let $\mathcal{G} = \langle \mathcal{T}_1^G, \dots, \mathcal{T}_n^G \rangle$ be a vector of guarantee transducers for the system processes and, for $p_j \in P^-$, let $\mathcal{G}_j = \left\{ \mathcal{T}_i^G \mid p_i \in P^- \setminus \{p_j\} \right\}$. Let $\tilde{\mathcal{S}} = \langle \tilde{s}_1, \dots, \tilde{s}_n \rangle$ be a vector of augmented local strategies for the system processes such that $\tilde{s}_i$ is an augmented local strategy for $p_i \in P^-$ with respect to $\mathcal{G}_i$. If $(\tilde{\mathcal{S}}, \mathcal{G})$ realizes φ, then $\tilde{s}_1 \mid\mid \dots \mid\mid \tilde{s}_n \models \varphi$.*

Proof. For $p_i \in P^-$, let $\tilde{\mathcal{T}}_i$ be the finite-state transducer representing $\tilde{s}_i$ and let $\hat{\mathcal{T}}_i$ be the finite-state transducer representing local$(\hat{s})_i$. Let $\hat{\mathcal{S}} = \langle \text{local}(\tilde{s}_1), \dots, \text{local}(\tilde{s}_n) \rangle$. By Lemma 4.8, augmented local strategies satisfy the requirements of local strategies regarding finiteness and infiniteness of computations. Thus, since $\tilde{s}_i$ is an augmented local strategy for $p_i \in P^-$ with respect to $\mathcal{G}_i$ by construction, it follows from its construction that local$(\tilde{s}_i)$ is a local strategy for p_i with respect to $\mathcal{G}_i$. Furthermore, since $(\tilde{\mathcal{S}}, \mathcal{G})$ realizes φ by assumption, for all $p_i \in P^-$, we have both $\hat{\mathcal{T}}_i \preceq \mathcal{T}_i^G$ and, for all $\gamma \in (2^{I_i})^\omega$, $\gamma' \in (2^{V \setminus V_i})^\omega$, either $comp(\text{local}(\tilde{s}_i), \gamma)$ is finite or $comp(\text{local}(\tilde{s}_i), \gamma) \cup \gamma' \models \varphi$ holds. Thus, it follows immediately with the definition of certifying synthesis with local strategies that $(\hat{\mathcal{S}}, \mathcal{G})$ realizes φ as well. Therefore, by Corollary 4.1, we have local$(\tilde{s}_1) \mid\mid \dots \mid\mid$ local$(\tilde{s}_n) \models \varphi$. Additionally, by Lemma 4.4, the traces produced by $\hat{\mathcal{T}}_1 \mid\mid \dots \mid\mid \hat{\mathcal{T}}_n$ coincide with the traces produced by the parallel composition of the transducers $\tilde{\mathcal{T}}_i$

representing the strategy extensions $s_i = \text{extend}(\text{local}(\tilde{s}_i), \mathcal{T}_i^G)$ of the local strategies $\hat{\mathcal{T}}_i$ as defined in Definition 4.9. Since $\mathcal{T}_i$ is a deterministic and complete Moore transducer and since the sets of output variables of different processes are disjoint, it follows with Lemma 2.1 that that the parallel composition of the transducers representing the extended strategies is deterministic and complete as well. Thus, for all $\gamma \in (2^{O_{env}})^\omega$, we have $|\mathit{Traces}(\mathcal{T}_1 \,||\, \ldots \,||\, \mathcal{T}_n, \gamma)| = 1$ and therefore $|\mathit{Traces}(\hat{\mathcal{T}}_1 \,||\, \ldots \,||\, \hat{\mathcal{T}}_n, \gamma)| = 1$ follows. Furthermore, since deterministic and complete transducers only produce infinite traces, all traces produced by $\mathcal{T}_1 \,||\, \ldots \,||\, \mathcal{T}_n$ and thus also all traces produced by $\hat{\mathcal{T}}_1 \,||\, \ldots \,||\, \hat{\mathcal{T}}_n$ are infinite.

Let $\gamma \in (2^{O_{env}})^\omega$. Let $\sigma \in \mathit{Traces}(\hat{\mathcal{T}}_1 \,||\, \ldots \,||\, \hat{\mathcal{T}}_n, \gamma)$ be the unique trace produced by the parallel composition of the local strategies $\text{local}(\tilde{s}_i)$ on input γ. Then, since σ is infinite as shown above, $\sigma \cap V_i \in \mathit{Traces}(\hat{\mathcal{T}}_i, \rho \cap I_i)$ holds for all $p_i \in P^-$ by Proposition 4.1. Therefore, by construction of $\text{local}(\tilde{s}_i)$ and $\hat{\mathcal{T}}_i$, we have $\sigma \cap V_i \in \mathit{Traces}(\tilde{\mathcal{T}}_i, \rho \cap I_i)$ for all $p_i \in P^-$ as well. Hence, $\sigma \in \mathit{Traces}(\tilde{\mathcal{T}}_1 \,||\, \ldots \,||\, \tilde{\mathcal{T}}_n)$ follows with Proposition 4.1 and thus, in particular $\sigma \in \mathit{Traces}(\tilde{\mathcal{T}}_1 \,||\, \ldots \,||\, \tilde{\mathcal{T}}_n, \gamma)$ holds. Since the sets output variables of different processes are disjoint by the definition of archictures and since augmented local strategies are deterministic Moore transducers, $\tilde{\mathcal{T}}_1 \,||\, \ldots \,||\, \tilde{\mathcal{T}}_n$ is a deterministic Moore transducer by Lemma 2.1 as well and therefore σ is the unique trace produced by $\tilde{\mathcal{T}}_1 \,||\, \ldots \,||\, \tilde{\mathcal{T}}_n$ on input γ. Thus, for all $\gamma \in (2^{O_{env}})^\omega$, we have $\mathit{Traces}(\hat{\mathcal{T}}_1 \,||\, \ldots \,||\, \hat{\mathcal{T}}_n, \gamma) = \mathit{Traces}(\tilde{\mathcal{T}}_1 \,||\, \ldots \,||\, \tilde{\mathcal{T}}_n, \gamma)$. Since $\text{local}(\tilde{s}_1) \,||\, \ldots \,||\, \text{local}(\tilde{s}_n) \models \varphi$ holds as shown above, it thus follows that $\tilde{s}_1 \,||\, \ldots \,||\, \tilde{s}_n \models \varphi$ holds as well. □

Hence, when obtaining a solution of certifying synthesis with augmented local strategies, the parallel composition of the augmented local strategies restricted to the respective output variables is guaranteed to realize the specification. Completeness is ensured as long as for every system process $p_i \in P^-$, the behavior of the parallel composition of the guarantee transducers of all other system processes is deterministic in the sense that the paths produced by the parallel composition of the guarantee transducers are the same on input sequences that p_i cannot distinguish: an augmented local strategy needs to keep track of the matching traces in the guarantee transducers of all other system processes for the relations S^i_j to exist. If one of the other processes behaves differently on two sequences p_i cannot distinguish, then the definition of the relation S^i_j would require contradicting valuations of associated outputs in some state of the augmented local strategy. We call this kind of determinism *observation determinism* and define a slightly more general form as follows:

Definition 4.13 (Observation Determinism).
Let $p_i \in P^-$ be some system process. Let $I \subseteq V$ and $O \subseteq O^-$ be finite sets of input and output variables. Let $\mathcal{P} \subseteq P^- \setminus \{p_i\}$. Let $\mathcal{M}$ be a set of complete finite-state transducers, one for each $p_j \in \mathcal{P}$. Let $I = \bigcup_{p_j \in \mathcal{P}} I_j \cup \bigcup_{p_j \in \mathcal{P}} O_j$. We call $\mathcal{M}$ observation-deterministic for p_i if, and only if, we have $\mathit{Paths}(\mathcal{T}, \gamma) = \mathit{Paths}(\mathcal{T}, \gamma')$ for all $\gamma, \gamma' \in (2^I)^\omega$ with $\gamma \cap V_i = \gamma' \cap V_i$.

In the following completeness proof for certifying synthesis with augmented local strategies, we require the strategies whose parallel composition realizes the specification to ensure observation determinism for every process $p_i \in P^-$ and the parallel composition of the other *strategies.* Then, by Theorem 4.2 and, in particular, Lemma 4.2, there exist complete strategies and guarantee transducers that realize the specification. Furthermore, the construction of the guarantee transducers Lemma 4.2 preserves observation determinism as the guarantee transducers are copies of the strategies which are restricted to the guarantee outputs. Strategy restriction as defined in Definition 4.10 then allows for constructing local strategies that, together with the very same guarantee transducers as before, realize φ as well (see Lemma 4.7). Since, for each $p_i \in P^-$, the parallel composition of the guarantee transducers of the other processes is observation-deterministic, it follows that in every state of the local strategy the part which represents a set of states in the parallel composition of the guarantee transducers is a singleton. We can thus utilize a slightly modified version of strategy restriction which also preserves the outputs of the guarantee transducers to obtain augmented local strategies. Then, in fact, the resulting transducer represents an augmented local strategy that (i) produces the same traces when restricting them to the outputs of the considered system process and (ii) if the local strategy is simulated by the guarantee transducer, then so is the augmented local strategy constructed in this way. Hence, together with Corollary 4.2, we obtain the following completeness result:

Lemma 4.10. *Let φ be an LTL formula over atomic propositions V and let $\langle \varphi_1, \dots, \varphi_n \rangle$ be its decomposition. Let $s_1, \dots, s_n$ be strategies for the system processes represented by finite-state transducers $\mathcal{T}_1, \dots, \mathcal{T}_n$. If we have $prop(\varphi_i) \subseteq V_i$ for all $p_i \in P^-$, and if, for all $p_i \in P^-$, the set $\{\mathcal{T}_j \mid p_j \in P^- \setminus \{p_i\}\}$ is observation-deterministic for p_i, and if $s_1 \mid\mid \dots \mid\mid s_n \models \varphi$ holds, then there exists a vector $\mathcal{G} = \langle \mathcal{T}_1^G, \dots, \mathcal{T}_n^G \rangle$ of guarantee transducers for the system processes and a vector $\tilde{\mathcal{S}} = \langle \tilde{s}_1, \dots, \tilde{s}_2 \rangle$ such that $\tilde{s}_i$ is an augmented local strategy for $p_i \in P^-$ with respect to $\mathcal{G}_i$, where $\mathcal{G}_i = \left\{\mathcal{T}_j^G \mid p_j \in P^- \setminus \{p_i\}\right\}$, such that $(\tilde{\mathcal{S}}, \mathcal{G})$ realizes φ.*

Proof. Assume that $s_1 \mid\mid \dots \mid\mid s_n \models \varphi$ holds and that, for all $p_i \in P^-$, the set $\{\mathcal{T}_j \mid p_j \in P^- \setminus \{p_i\}\}$ is observation-deterministic for p_i. Let $\mathcal{S} = \langle s_1, \dots, s_n \rangle$ and let $\mathcal{T}_i$ be the finite-state transducer representing s_i. Then, by Theorem 4.2 there exists a vector $\mathcal{G} = \langle \mathcal{T}_1^G, \dots, \mathcal{T}_n^G \rangle$ of guarantee transducers such that $(\mathcal{S}, \mathcal{G})$ realizes φ. In particular, this holds for the guarantee transducers which are copies of the transducers representing the corresponding strategy that are restricted to the guarantee outputs since this construction is used in the completeness proof (see Lemma 4.2). In the following, we assume that this vector of guarantee transducers is given, i.e., that, for all $p_i \in P^-$ and all $\gamma \in (2^{I_i})^\omega$, we have both $\#_1(\pi_k^{i,G}) = \#_1(\pi_k^i)$ and $\#_2(\pi_k^{i,G}) = \#_1(\pi_k^i) \cap O_2^G$ for all paths $\pi^{i,G} \in Paths(\mathcal{T}_i^G, \gamma)$ and $\pi^i \in Paths(\mathcal{T}_i, \gamma)$ as well as all points in time $k \geq 0$. For $p_j \in P^-$, let $\mathcal{G}_j = \{\mathcal{T}_i^G \mid p_i \in P^- \setminus \{p_j\}\}$. Since, for all $p_i \in P^-$, the set $\{\mathcal{T}_j \mid p_j \in P^- \setminus \{p_i\}\}$ is observation-deterministic for p_i, it follows immediately that, for all $p_i \in P^-$, the set $\mathcal{G}_i$ is observation-deterministic for p_i as well.

We construct augmented local strategies $\tilde{s}_i$ similar to the construction of local strategies from complete strategies as defined in Definition 4.10. The only difference lies in the labeling function: instead of defining the label of a state (t, M) to be $\ell_i(t)$, we define it to be $\ell_i(t) \cup (\ell(t') \cap O_i^A)$, where $t' \in M$. Note that, since for all $p_i \in P^-$, the set $\mathcal{G}_i$ is observation-deterministic for p_i, the second component of a state in restrict$(s_i, \mathcal{G}_i)$ is always a singleton. Hence, the labeling function of the constructed augmented local strategy is well-defined. We show that the transducers constructed $\tilde{\mathcal{T}}_1, \dots, \tilde{\mathcal{T}}_n$ in this way indeed represent augmented local strategies. Since we only alter the labeling function in a well-defined way, finiteness of the set of states as well as determinism and completeness of the transducers follow from the respective properties for the local strategies obtained with strategy restriction. Next, for $p_i \in P^-$, let $\tilde{\mathcal{T}}_i = (\tilde{T}_i, \tilde{T}_{i,0}, \tilde{\tau}_i, \tilde{\ell}_i)$, let $\mathcal{T}_i^G = (T_i^G, T_{i,0}^G, \tau_i^G, \ell_i^G)$ and let $\mathcal{T} = (T, T_0, \tau_0, \ell)$ be the transducer representing the parallel composition of the guarantee transducers of the other processes. Let $p_i \in P^-$.

First, we show that condition (i) is satisfied. Let $(t, M) \in \tilde{T}_i$ be a state of $\tilde{\mathcal{T}}_i$. As shown above, M is a singleton. Let $M = \{t^G\}$. Suppose that there exists some transition $((t, M), \iota, (t', M')) \in \tilde{\tau}_i$. Then, by construction of $\tilde{\tau}_i$, we have $\iota \cap O^- = \ell(t^G) \cap I_i$. Furthermore, by construction of the labeling function, $\tilde{\ell}_i((t, M)) \cap O_i^A = \ell(t^G) \cap O_i^A$ holds. Since $\ell(t^G) \cap I_i = \ell(t^G) \cap O_i^A$ holds by definition of associated outputs, $\iota \cap O^- = \tilde{\ell}_i((t, M)) \cap O_i^A$ thus follows. Next, suppose that there is some $\iota \in 2^{I_i}$ such that $\iota \cap O^- = \tilde{\ell}_i((t, M)) \cap O_i^A$ holds. Since $\tilde{\mathcal{T}}_i$ is constructed from a complete strategy s_i, there exists a transition $(t, \iota, t') \in \tau_i$, where τ_i is the transition relation of the finite-state transducer representing s_i. Moreover, there clearly exists some $\iota' \in 2^{I_i^G}$, where $I_i^G = \bigcup_{p_j \in P^- \setminus \{p_i\}} I_i \setminus \bigcup_{p_j \in P^- \setminus \{p_i\}} O_i$, such that $\iota \cap I_i^G = \iota' \cap I_i$ and $\iota' \cap O_i = \ell_i(t) \cap I_i^G$ hold. Since $\iota \cap O^- = \tilde{\ell}_i((t, M)) \cap O_i^A$ holds by assumption, in particular, $\iota \cap O_i^G = \tilde{\ell}_i((t, M)) \cap I_i$ holds by definition of associated outputs, where $O_i^G = \bigcup_{p_j \in P^- \setminus \{p_i\}} O_i$. Furthermore, since the sets of output variables of different processes are disjoint by definition of architectures and since guarantee transducers are complete Moore transducers, their parallel composition $\mathcal{T}$ is, by Lemma 2.1, complete as well and thus, in particular, there exists a transition $(t^G, \iota', t^{G'}) \in \tau$. Thus, by definition of $\tilde{\tau}_i$, we have $((t, M), \iota, (t', M')) \in \tilde{\tau}_i$, where $M' = \{t^{G'}\}$. Hence, condition (i) of the definition of augmented local strategies is satisfied.

Next, we construct relations S_j^i for all $p_j \in P^- \setminus \{p_i\}$ as follows: $(t_j^G, (t, M)) \in S_j^i$ if, and only if, $M = \{t^G\}$ and $\#_m(t^G) = t_j^G$, where $m = j$ if $j < i$ and $m = j - 1$ otherwise, i.e., intuitively, the part of t^G which corresponds to p_j equals t_j^G. Let $p_j \in P^- \setminus \{p_i\}$. Clearly, $(t_{j,0}^G, \tilde{t}_0) \in S_j^i$ holds for all $t_{j,0}^G \in T_{j,0}^G$ and all $\tilde{t}_0 \in \tilde{T}_{i,0}$ by construction of $\tilde{\mathcal{T}}_i$. Next, let $t_j^G \in T_j^G$ and let $\tilde{t} \in \tilde{T}_i$ be states such that $(t_j^G, \tilde{t}) \in S_j^i$ holds. Let $\iota \in 2^{I_j}$ and $\iota' \in 2^{I_i}$ such that both $\iota \cap I_i = \iota' \cap I_j$ and $\tilde{\ell}(\tilde{t}) \cap O_i^A = \iota' \cap O_i^A$ hold. Since $(t_j^G, \tilde{t}) \in S_j^i$, we have $\#_m(\#_2(\tilde{t})) = t_j^G$, where $m = j$ if $j < i$ and $m = j - 1$ otherwise, by construction of S_j^i. Thus, in particular $\ell_j^G(t^G) = \ell(\#_2(\tilde{t})) \cap O_j^G$ holds by definition of the parallel composition of finite-state transducers. By construction of $\tilde{\mathcal{T}}_i$, we have $\tilde{\ell}_i(\tilde{t}) \cap O_j^G = \ell(\#_2(\tilde{t})) \cap O_j^G$ since $O_i \cap O_j^G$ holds by definition of architectures and guarantee outputs. Therefore, $\tilde{\ell}(\tilde{t}) \cap O_j^G = \ell_j^G(t_j^G)$ follows. Furthermore, since $\tilde{\ell}_i(\tilde{t}) \cap O_i^A = \iota' \cap O_i^A$ holds by assumption and since condition (i) of the definition of augmented local strategies is satisfied as shown above, there exists a transition $(\tilde{t}, \iota', (t, M)) \in \tilde{\tau}_i$ for some $(t, M) \in \tilde{T}_i$. Thus, by construction of $\tilde{\mathcal{T}}_i$, there also exists a transition $(\#_1(\tilde{t}), \iota', t) \in \tau_i$, where τ_i is the transition relation of the finite-state transducer representing s_i. If there exists a transition $(t_j^G, \iota, t_j^{G'}) \in \tau_j^G$, then it follows with the construction of $\tilde{\mathcal{T}}_i$ and the fact that it is deterministic that we have $\#_m(t^{G'}) = t_j^{G'}$, where $M = \{t^{G'}\}$ and $m = j$ if $j < i$ and $m = j - 1$ otherwise. Therefore, by construction of S_j^i, we have $(t_j^{G'}, (t, M)) \in S_j^i$ as well. Hence, S_j^i satisfies the requirements stated in the definition of augmented local strategies and thus condition (ii) holds.

Lastly, we show that the augmented local strategies $\tilde{s}_i$ indeed form a solution of certifying synthesis. Let $\tilde{\mathcal{S}} = \langle \tilde{s}_1, \ldots, \tilde{s}_n \rangle$. By construction of $\tilde{s}_i$, we clearly have $\mathrm{local}(\tilde{s}_i) = \mathrm{restrict}(s_i, \mathcal{G}_i)$.

Let $\hat{\mathcal{S}} = \langle \text{local}(\tilde{s}_1), \dots, \text{local}(\tilde{s}_n)\rangle$. Hence, it follows with Lemma 4.7 that $(\hat{\mathcal{S}}, \mathcal{G})$ realizes φ. That is, for all $p_i \in P^-$, we have both $\hat{\mathcal{T}}_i \preceq \mathcal{T}_i^G$ and, for all $\gamma \in (2^{I_i})^\omega$, $\gamma' \in (2^{V \setminus V_i})^\omega$, either $comp(\text{local}(\tilde{s}_i), \gamma)$ is finite or $comp(\text{local}(\tilde{s}_i), \gamma) \cup \gamma' \models \varphi$ holds. Thus, by definition of certifying synthesis with augmented local strategies, $(\tilde{\mathcal{S}}, \mathcal{G})$ thus realizes φ as well. □

Therefore, it follows from Lemma 4.9 and Lemma 4.10 that utilizing augmented local strategies for certifying synthesis is sound and, under certain conditions, complete. In the following, we thus introduce an encoding of the search for augmented local strategies and guarantee transducers that satisfy the requirements of certifying synthesis into a SAT constraint system. We focus on synthesizing deterministic guarantee transducers that ensure observation determinism. As outlined above, such guarantee transducers do not necessarily exist for all system architectures. In the following sections, however, we introduce several optimizations of certifying synthesis, one of them being to allow for nondeterministic guarantee transducers as well as transducers that not ensure observation determinism.

4.4.4. Constraint System for Deterministic Certificates

Like for monolithic bounded synthesis [FS13], we encode the search for a solution of certifying synthesis of a certain size into a SAT constraint system. We reuse the concepts of run graphs and valid annotations (see Section 2.8.1). Therefore, we employ parts of the SAT constraint system for bounded synthesis of monolithic systems presented in [FFRT17]. In particular, the constraint system $\mathbb{C}_{\mathscr{A},B,\varphi}$ encoding certifying synthesis for some architecture $\mathscr{A}$, a vector B of size bounds for both augmented local strategies and guarantee transducers, and some LTL specification φ consists, intuitively, of n slightly modified copies of the SAT constraint system for monolithic bounded synthesis, one for each system process of $\mathscr{A}$. For each copy, we add variables encoding the guarantee transducers representing the certificates as well as constraints that ensure that the augmented local strategies and certificates indeed fulfill the requirements of certifying synthesis with augmented local strategies and guarantee transducers.

First, we encode the finite-state transducers $\mathcal{T}_i$ representing the augmented local strategies, the finite-state transducers $\mathcal{T}_i^G$ representing the certificates, the universal co-Büchi automata $\mathcal{A}_i$ representing the specifications φ_i, and the annotation function λ. Furthermore, we encode the simulation relations R_i that establish that $\mathcal{T}_i \preceq \mathcal{T}_i^G$ holds as well as the relations S_j^i that establish that $\mathcal{T}_i$ is an augmented local strategy.

- Finite-state transducer $\mathcal{T}_i = (T_i, T_{i,0}, \tau_i, \ell_i)$. We represent the transition relation τ_i by one Boolean variable $\tau^i_{t,\iota,t'}$ for each $t, t' \in T_i$ and $\iota \in 2^{I_i}$. Given $t, t' \in T_i$, $\iota \in 2^{I_i}$, and $o \in 2^{O_i}$, it holds that $\tau^i_{t,\iota,t'}$ is *true* if, and only if, $(t, \iota, t') \in \tau_i$ holds. Furthermore, we represent the labeling relation ℓ_i by one Boolean variable $o^i_{t,o}$ for each $t \in T_i$ and $o \in 2^{O_i \cup O_i^A}$. Given $t \in T_i$ and $o \in O_i \cup O_i^A$, it holds that $o^i_{t,o}$ is *true* if, and only if, $o \in \ell_i(t)$ holds.
- Finite-state transducer $\mathcal{T}_i^G = (T_i^G, T_{i,0}^G, \tau_i^G, \ell_i^G)$. We represent the transition relation τ_i^G by one Boolean variable $\tau^{G,i}_{t,\iota,t'}$ for each $t, t' \in T_i^G$ and $\iota \in 2^{I_i}$. Given $t, t' \in T_i^G$ and $\iota \in 2^{I_i}$, it holds that $\tau^{G,i}_{t,\iota,t'}$ is *true* if, and only if, $(t, \iota, t') \in \tau_i^G$ holds. Furthermore, we represent

the labeling relation ℓ_i^G by one Boolean variable $o_{t,o}^{G,i}$ for each $t \in T_i^G$ and $o \in 2^{O_i^G}$. Given $t \in T_i^G$ and $o \in O_i^G$, it holds that $o_{t,o}^{G,i}$ is *true* if, and only if, $o \in \ell_i^G(t)$ holds.

- Universal co-Büchi automaton $\mathcal{A}_i = (Q_i, q_{i,0}, \delta_i, F_i)$. We represent the transition relation δ_i by one propositional formula $\delta^i_{q,v,q'}$ for each $q, q' \in Q_i$ and $v \in 2^{V_i}$. Given $q, q' \in Q_i$ and $v \in 2^{V_i}$, it holds that $\delta^i_{q,v,q'}$ is *true* if, and only if, $(q, v, q') \in \delta_i$ holds.
- Annotation function $\lambda_i : T_i \times Q_i \rightarrow \mathbb{N} \cup \{\bot\}$. We split the encoding of λ_i into two parts, one focusing on the reachability of a state of the run graph of $\mathcal{T}_i$ and $\mathcal{A}_i$ and one focusing on the actual bound. We thus represent λ_i by one Boolean variable $\lambda^{\mathbb{B}}_{i,t,q}$ for each $t \in T_i$ and $q \in Q_i$ and one bit vector $\lambda^{\#}_{t,t,q}$ for each $t \in T_i$ and $q \in Q_i$. Given $t \in T_i$ and $q \in Q_i$, it holds that (i) $\lambda^{\mathbb{B}}_{i,t,q}$ is *true* if, and only if, $\lambda_i(t,q) \neq \bot$ and (ii) $\lambda^{\#}_{i,t,q}$ represents the binary encoding of the value $\lambda_i(t,q)$ if $\lambda_i(t,q) = k \neq \bot$.
- Simulation relation $R_i : T_i \times T_i^G$. We represent the simulation relation by one propositional formula $R^i_{t,u}$ for each $t \in T_i$ and $u \in T_i^G$. Given $t \in T_i$ and $u \in T_i^G$, it holds that $R^i_{t,u}$ is *true* if, and only if, $(t, u) \in R_i$ holds.
- Relation $S^i_j : T_j^G \times T_i$. We represent the relation by one propositions formula $S^{i,j}_{u,t}$ for each $u \in T_j^G$ and $t \in T_i$. Given $u \in T_j^G$ and $t \in T_i$, it holds that $S^{i,j}_{u,t}$ is *true* if, and only if, $(u, t) \in S^i_j$ holds.

Next, we present the SAT constraint system $\mathbb{C}_{\mathscr{A},B,\varphi}$ encoding certifying synthesis with augmented local strategies for an architecture $\mathscr{A}$, a vector B of size bounds, and an LTL formula φ with decomposition $\langle \varphi_1, \dots, \varphi_n \rangle$, where each φ_i is represented by a universal co-Büchi automaton $\mathcal{A}_i$ with $\mathcal{L}(\mathcal{A}_i) = \mathcal{L}(\varphi_i)$. We first present the SAT constraints for a single system process $p_i \in P^-$. Afterward, we assemble the constraints for the individual processes to the full constraint system $\mathbb{C}_{\mathscr{A},B,\varphi}$.

First, we encode the technical requirement that guarantee transducers must be both deterministic and complete, i.e., that there exists exactly one outgoing transition for every source state $u \in T_i^G$ and every input valuation $\iota \in 2^{I_i}$. The constraint is similar to the one encoding determinism and completeness of strategies in the SAT constraint system $\mathbb{C}^{\mathrm{SAT}}_{b,\varphi}$ for classical monolithic bounded synthesis [FS13, FFRT17].

$$\bigwedge_{u \in T_i^G} \bigwedge_{\iota \subseteq I_i} \left(\bigvee_{u' \in T_i^G} \tau^{G,i}_{u,\iota,u'} \wedge \bigwedge_{u' \in T_i^G} \bigwedge_{u'' \in T_i^G \setminus \{u'\}} \neg \left(\tau^{G,i}_{u,\iota,u'} \wedge \tau^{G,i}_{u,\iota,u''} \right) \right) \tag{4.1}$$

Second, we encode that an augmented local strategy needs to adhere to its own certificate, i.e., that $\mathcal{T}_i \preceq \mathcal{T}_i^G$ holds. For this, we explicitly encode the existence of a simulation relation $R : T \times T^G$ that establishes that $\mathcal{T}_i^G$ simulates $\mathcal{T}_i$. The constraint thus closely follows the definition of transducer simulation presented in Definition 4.5. Note that due to the Moore semantics of both augmented local strategies and guarantee transducers, both $\mathcal{T}_i$ and $\mathcal{T}_i^G$ only have a single initial state. Thus, instead of encoding that $(t_{i,0}, t^G_{i,0}) \in R_i$ holds for all $t_{i,0} \in T_{i,0}$ and all $t^G_{i,0} \in T^G_{i,0}$, we simply encode that $(t_{i,0}, t^G_{i,0}) \in R_i$ holds, where $t_{i,0}$ and $t^G_{i,0}$ represent the unique initial states of $\mathcal{T}_i$ and $\mathcal{T}_i^G$, respectively. Recall that transducers representing local strategies have a unique initial state, although they can be transition-incomplete due to the requirements on the finiteness and infiniteness of computations of local strategies (see Proposition 4.2). Since augmented local strategies satisfy the same properties, $\mathcal{T}_i$ has a unique initial state as well, although it can be transition-incomplete.

$$R^i_{t_{i,0},t^G_{i,0}} \wedge \bigwedge_{t\in T_i} \bigwedge_{u\in T_i^G} \left(R^i_{t,u} \rightarrow \left(\bigwedge_{o\in O_i^G} o^{G,i}_{u,o} \leftrightarrow o^i_{t,o} \wedge \bigwedge_{\iota\subseteq I_i} \bigwedge_{t'\in T_i} \left(\tau^i_{t,\iota,t'} \rightarrow \bigvee_{u'\in T_i^G} \left(\tau^{G,i}_{u,\iota,u'} \wedge R^i_{t',u'} \right)\right)\right)\right) \tag{4.2}$$

Third, we encode that $\mathcal{T}_i$ adheres to condition (i) of the definition of augmented local strategies. That is, we encode that for every state $t \in \mathcal{T}_i$ and every input $\iota \in 2^{I_i}$, there exists a transition with source state t for input ι if, and only if $\iota \cap O_i^A = \ell(t) \cap O_i^A$ holds. Moreover, we encode that $\mathcal{T}_i$ is deterministic, i.e., that for every state $t \in \mathcal{T}_i$ and every input $\iota \in 2^{I_i}$, there exists at most one successor state.

$$\bigwedge_{t\in T_i} \bigwedge_{\iota\subseteq I_i} \left(\left(\bigwedge_{o\in O_i^A} o \in \iota \leftrightarrow o^i_{t,o} \right) \leftrightarrow \bigvee_{t'\in T_i} \tau^i_{t,\iota,t'} \right) \wedge \bigwedge_{t'\in T_i} \bigwedge_{t''\in T_i\setminus\{t'\}} \neg \left(\tau^i_{t,\iota,t'} \wedge \tau^i_{t,\iota,t''} \right) \tag{4.3}$$

Next, we encode that $\mathcal{T}_i$ further adheres to condition (ii) of the definition of augmented local strategies. For this, we explicitly encode the existence of relations S^i_j that satisfy the requirements stated in the definition of augmented local strategies, i.e., in Definition 4.11. Similar to constraint Equation (4.2), i.e., the constraint encoding the existence of a simulation relation establishing that $\mathcal{T}_i \preceq \mathcal{T}_i^G$ holds, we make use of the Moore semantics of both augmented local strategies and guarantee transducers: instead of encoding that $(t^G_{j,0}, t_{i,0}) \in S^i_j$ holds for all $t^G_{j,0} \in T^G_{j,0}$ and all $t_{i,0} \in T_{i,0}$, we encode that $(t^G_{j,0}, t_{i,0}) \in S^i_j$ holds, where $t^G_{j,0}$ and $t_{i,0}$ represent the unique initial states of $\mathcal{T}_j^G$ and $\mathcal{T}_i$, respectively.

$$\begin{aligned} & S^{j,i}_{t^G_{j,0},t_{i,0}} \wedge \\ & \bigwedge_{u\in T_j^G} \bigwedge_{t\in T_i} \Bigg(S^{j,i}_{u,t} \rightarrow \Bigg(\bigwedge_{o\in O_i^A\cap O_j^G} o^{G,j}_{u,o} \leftrightarrow o^i_{t,o} \wedge \bigwedge_{\iota\subseteq I_j} \bigwedge_{\iota'\subseteq I_i} \Bigg(\iota \cap I_i = \iota' \cap I_j \\ & \qquad \wedge \bigwedge_{o\in O_i^A} o \in \iota' \leftrightarrow o^i_{t,o} \Bigg) \rightarrow \bigwedge_{u'\in T_j^G} \left(\tau^{G,j}_{u,\iota,u'} \rightarrow \bigvee_{t'\in T_i} \left(\tau^i_{t,\iota',t'} \wedge S^{j,i}_{u',t'} \right)\right)\Bigg)\Bigg) \end{aligned} \tag{4.4}$$

This constraint considers only a single other system process $p_j \in P^- \setminus V_i$, i.e., it only encodes the existence of a single relation S^i_j satisfying the requirements of local strategies. Hence, to ensure that there does not only exist a relation S^i_j for a single system process $p_j \in P^- \setminus \{p_i\}$ but for *all* other system processes, we use one copy of this constraint for each system process that is different from p_i, i.e., we use $\bigwedge_{p_j\in P^-\setminus\{p_i\}}$ (4.4).

Lastly, we encode that the run graph of the local strategy represented by $\mathcal{T}_i$ and the universal co-Büchi automaton $\mathcal{A}_i$ has a valid annotation. The constraint is similar to the one for classical monolithic bounded synthesis [FS13, FFRT17].

$$\lambda^{\mathbb{B}}_{i,t_{i,0},q_{i,0}} \wedge \bigwedge_{q \in Q_i} \bigwedge_{t \in T_i} \left(\lambda^{\mathbb{B}}_{i,t,q} \rightarrow \bigwedge_{q' \in Q_i} \bigwedge_{\iota \subseteq I_i} \left(\delta^i_{q,\iota,q',t} \rightarrow \bigwedge_{t' \in T_i} \left(\tau^i_{t,\iota,t'} \rightarrow \left(\lambda^{\mathbb{B}}_{i,t',q'} \wedge \lambda^{\#}_{i,t',q'} \triangleright_{q'} \lambda^{\#}_{i,t,q} \right) \right) \right) \right) \tag{4.5}$$

As in monolithic bounded synthesis, we use the notation $\delta^i_{q,\iota,q',t}$ to denote that there exists a transition in $\mathcal{A}_i$ from q to q' with $\iota \cup o$, where o is the set of output variables of $\mathcal{T}_i$ for every transition with source state t. That is, $\delta^i_{q,\iota,q',t}$ is syntactic sugar for

$$\delta^i_{q,\iota,q',t} := \bigwedge_{o \subseteq O_i} \delta^i_{q,\iota \cup o,q'} \wedge \bigwedge_{v \in O_i} v \in o \leftrightarrow o^i_{t,o}.$$

Although $\mathcal{T}_i$ is an augmented local strategy and thus outputs in every step a valuation of the variables in $O_i \cup O^A_i$, we only consider the outputs of process p_i, i.e., the variables in O_i in the definition of $\delta^i_{q,\iota,q',t}$ and thus in constraint (4.5) encoding the existence of a valid annotation of the run graph. Since condition (i) of the definition of augmented local strategies and thus constraint (4.3) ensures that augmented local strategies only produce well-defined traces, this is equivalent to considering the variables in $O_i \cup O^A_i$ while yielding a smaller constraint systems.

Combining all these constraints, we obtain the following constraint system $\mathbb{C}_{\mathscr{A},B,\varphi}$ for certifying synthesis with augmented local strategies and guarantee transducers:

$$\bigwedge_{p_i \in P^-} \left((4.1) \wedge (4.2) \wedge \left(\bigwedge_{p_j \in P^- \setminus \{p_i\}} (4.4) \right) \wedge (4.3) \wedge (4.5) \right)$$

Since the constraint system $\mathbb{C}_{\mathscr{A},B,\varphi}$ explicitly encodes the search for augmented local strategies and guarantee transducers that satisfy the requirements of certifying synthesis, it thus follows from the soundness and completeness of certifying synthesis with augmented local strategies, i.e., Lemma 4.9 and Lemma 4.10, that $\mathbb{C}_{\mathscr{A},B,\varphi}$ can be used to compositionally derive a solution for the synthesis task for a distributed system.

Theorem 4.3. *Let $\mathscr{A}$ be an architecture, let φ be an LTL formula, and let B be size bounds for the strategies and certificates. There is a SAT constraint system $\mathbb{C}_{\mathscr{A},B,\varphi}$ such that (i) if $\mathbb{C}_{\mathscr{A},B,\varphi}$ is satisfiable, then φ is realizable in $\mathscr{A}$, and (ii) if φ is realizable in $\mathscr{A}$ for the bounds B and additionally $prop(\varphi_i) \subseteq V_i$ holds for all $p_i \in P^-$ and, for all $p_i \in P^-$, observation determinism of the strategies of the other system processes can be ensured, then $\mathbb{C}_{\mathscr{A},B,\varphi}$ is satisfiable.*

Note that we build a *single* constraint system for the whole certifying synthesis task. That is, the augmented local strategies and certificates of the individual processes are not synthesized entirely independently. This is one of the main differences between our approach and the negotiation-based assume-guarantee synthesis algorithm [MMSZ20]. While this prevents separate synthesis tasks and thus parallelizability, it eliminates the need for a negotiation between the processes. Moreover, it allows for completeness of the synthesis algorithm under certain conditions on the architecture. Although the synthesis tasks for the individual system processes are not fully separated, the constraint system $\mathbb{C}_{\mathcal{A},B,\varphi}$ is in most cases still significantly smaller and easier to solve than the one of classical distributed synthesis.

In the following two sections, we present two further optimizations of certifying synthesis, which allow, in many cases, for even smaller constraint systems and, thus, for finding solutions for even more complex architectures and specifications. First, we reduce the number of certificates a process $p_i \in P^-$ needs to consider by identifying processes that are *relevant* for p_i's strategy. In this way, we reduce the number of processes for which we need to add a copy of constraint (4.4). Afterward, we permit nondeterminism in certificates, thus possibly reducing the minimal number of states of a guarantee transducer.

4.5. Computing Relevant Processes

In all variants of certifying synthesis presented in the previous sections and, in particular, in the SAT constraint system that encodes certifying synthesis with augmented local strategies and guarantee transducers, we consider, for each system process $p_i \in P^-$, the certificates of *all* other system processes when formulating p_i's local objective. In many cases, however, it suffices only to consider the certificates of a *subset* of the other system processes. Consider, for instance, the robots from the running example presented in Section 4.1 and suppose that, in addition to r_1 and r_2, there is a third robot r_3. The additional robot uses a different route through the factory and thus never passes the crossing. Therefore, r_3's behavior does not influence whether or not r_1 or r_2 can enter the crossing at a certain point in time. Thus, in particular, r_3's certificate does not need to be considered in r_1's and r_2's local objective to be realizable.

Therefore, we present an optimization of certifying synthesis that reduces the number of considered certificates in this section. For every system process $p_i \in P^-$, we compute a set $\mathcal{R}_i \subseteq P^- \setminus \{p_i\}$ of *relevant processes*. Certifying synthesis then only considers the certificates of the relevant processes: let φ be an LTL formula over atomic propositions V with decomposition $\langle \varphi_1, \dots, \varphi_n \rangle$, and let $\mathcal{S} = \langle s_1, \dots, s_n \rangle$, $\Psi = \langle \psi_1, \dots, \psi_n \rangle$, and $\mathcal{G} = \langle \mathcal{T}_1^G, \dots, \mathcal{T}_n^G \rangle$ be vectors of strategies, LTL certificates, and guarantee transducers for the system processes respectively. For every system process $p_j \in P^-$, we define $\Psi_j^{\mathcal{R}} := \{\psi_i \mid p_i \in \mathcal{R}_j\}$ and $\mathcal{G}_j^{\mathcal{R}} := \{\mathcal{T}_i^G \mid p_i \in \mathcal{R}_j\}$. For certifying synthesis with LTL certificates, we then require that $s_i \models \psi_i \land (\Psi_i^{\mathcal{R}} \rightarrow \varphi_i)$ holds for every $p_i \in P^-$. For certifying synthesis with guarantee transducers and complete strategies, both $\mathcal{T}_i \preceq \mathcal{T}_i^G$ and $s_i \models_{\mathcal{G}_i^{\mathcal{R}}} \varphi_i$ need to hold for every $p_i \in P^-$, where $\mathcal{T}_i$ is the finite-state transducer representing s_i. For certifying synthesis with local strategies and guarantee transducers, let $\hat{\mathcal{S}} = \langle \hat{s}_1, \dots, \hat{s}_n \rangle$ be a vector such that $\hat{s}_i$ is a local strategy for $p_i \in P^-$ with respect to $\mathcal{G}_i^{\mathcal{R}}$. We then require that, for all $p_i \in P^-$, we have $\hat{\mathcal{T}}_i \preceq \mathcal{T}_i^G$, where $\hat{\mathcal{T}}_i$ is the finite-state transducer representing $\hat{s}_i$, and, for all $\gamma \in (2^{I_i})^\omega$, $\gamma' \in (2^{V \setminus V_i})^\omega$, either $comp(\hat{s}_i, \gamma)$ is finite or $comp(\hat{s}_i, \gamma) \cup \gamma' \models \varphi$ holds. Similarly, for certifying synthesis with augmented local strategies and guarantee transducers, let $\tilde{\mathcal{S}} = \langle \tilde{s}_1, \dots, \tilde{s}_n \rangle$ be a vector such that $\tilde{s}_i$ is an augmented local strategy for $p_i \in P^-$ with respect to $\mathcal{G}_i^{\mathcal{R}}$. We then require that, for all $p_i \in P^-$, we have $\tilde{\mathcal{T}}_i \preceq \mathcal{T}_i^G$, where $\tilde{\mathcal{T}_i}$ is the finite-state transducer representing $\tilde{s}_i$, and, for all $\gamma \in (2^{I_i})^\omega$, $\gamma' \in (2^{V \setminus V_i})^\omega$, either $comp(\tilde{s}_i, \gamma)$ is finite or $comp(\tilde{s}_i, \gamma) \cup \gamma' \models \varphi$ holds. Then, we say that $(\mathcal{S}, \Psi)_\mathcal{R}$, $(\mathcal{S}, \mathcal{G})_\mathcal{R}$, $(\hat{\mathcal{S}}, \mathcal{G})_\mathcal{R}$, and $(\tilde{\mathcal{S}}, \mathcal{G})_\mathcal{R}$ realize φ.

The construction of the sets of relevant processes needs to preserve soundness and completeness of certifying synthesis. In the following, we introduce a syntactic definition of relevant processes that does so. It excludes processes from p_i's set of relevant processes $\mathcal{R}_i$ whose output variables do not occur in the subspecification φ_i:

Definition 4.14 (Relevant Processes)**.**
Let φ be an LTL formula over atomic propositions V with decomposition $\langle \varphi_1, \dots, \varphi_n \rangle$. Let $p_i \in P^-$ be a system process. The *relevant processes* $\mathcal{R}_i \subseteq P^- \setminus \{p_i\}$ of p_i are defined by $\mathcal{R}_i = \{p_j \in P^- \setminus \{p_i\} \mid O_j \cap prop(\varphi_i) \neq \emptyset\}$.

Intuitively, since $O_j \cap prop(\varphi_i) = \emptyset$ holds for a system process $p_j \in P^- \setminus (\mathcal{R}_i \cup \{p_i\})$, the LTL formula φ_i does not restrict the valuations of p_j's output variables. Thus, if an infinite sequence satisfies φ_i, then it does so for any valuations of the variables in O_j. Hence, p_j's guaranteed behavior does not influence the satisfiability of φ_i and thus p_i does not need to consider it.

Example 4.8. Consider the robots from the running example and, for simplicity, suppose that the robots do not have additional objectives φ_{add_i}. Hence, the objective of robot r_i is given by $\varphi_i = \varphi_{no\_crash} \land \varphi_{cross_i}$. Since both go_1 and go_2 occur in φ_{no_crash}, clearly r_1 is relevant for r_2 and vice versa. This meets our observation that none of the robots can realize both φ_{no_crash} and φ_{cross_i} without information about the other robot's behavior. Suppose that there exists a third robot r_3 with output variable go_3 that uses a different route through the factory and thus never uses the crossing. Hence, we do not need to adapt φ_{no_crash} to specify that no crash occurs. Then, go_3 neither occurs in φ_{no_crash} nor in φ_{cross_i} and thus r_3 is not relevant for r_1 or for r_2. This meets our observation that there exist strategies for r_1 and r_2 that realize $\varphi_{no\_crash} \land \varphi_{cross_1}$ and $\varphi_{no\_crash} \land \varphi_{cross_2}$, respectively, without the need for taking r_3's certificate into account. △

By definition of relevant processes, we have $\mathcal{R}_i \subseteq P^- \setminus \{p_i\}$ for every system process $p_i \in P^-$. Thus, in particular, both $\Psi_i^\mathcal{R} \subseteq \Psi_i$ and $\mathcal{G}_i^\mathcal{R} \subseteq \mathcal{G}_i$ hold for every $p_i \in P^-$. Hence, soundness of certifying synthesis with complete strategies when only considering relevant processes, follows from the corresponding results when considering all other system processes:

Lemma 4.11. *Let φ be an LTL formula over atomic propositions V with decomposition $\langle \varphi_1, \dots, \varphi_n \rangle$. Let $\mathcal{S} = \langle s_1, \dots, s_n \rangle$, $\Psi = \langle \psi_1, \dots, \psi_n \rangle$, and $\mathcal{G} = \langle \mathcal{T}_1^G, \dots, \mathcal{T}_n^G \rangle$ be vectors of strategies, LTL certificates, and guarantee transducers for the system processes, respectively. If either $(\mathcal{S}, \Psi)_\mathcal{R}$ realizes φ or $(\mathcal{S}, \mathcal{G})_\mathcal{R}$ realizes φ, then $s_1 \mid\mid \dots \mid\mid s_n \models \varphi$ holds.*

Proof. For every $p_j \in P^-$, we define $\Psi_j = \left\{\psi_i \mid p_i \in P^- \setminus \{p_j\}\right\}$, $\mathcal{G}_j = \left\{\mathcal{T}_i^G \mid p_i \in P^- \setminus \{p_j\}\right\}$, $\Psi_j^\mathcal{R} = \left\{\psi_i \mid p_i \in \mathcal{R}_j\right\}$, and $\mathcal{G}_j^\mathcal{R} = \left\{\mathcal{T}_i^G \mid p_i \in \mathcal{R}_j\right\}$. First, assume that $(\mathcal{S}, \Psi)_\mathcal{R}$ realizes φ. Then, $s_i \models \psi_i \land (\Psi_i^\mathcal{R} \rightarrow \varphi_i)$ holds for all $p_i \in P^-$. By construction of the relevant processes, we have $\mathcal{R}_i \subseteq P^- \setminus \{p_i\}$ and thus $\Psi_i^\mathcal{R} \subseteq \Psi_i$ follows. Hence, since $s_i \models \psi_i \land (\Psi_i^\mathcal{R} \rightarrow \varphi_i)$ holds, $s_i \models \psi_i \land (\Psi_i \rightarrow \varphi_i)$ follows with the semantics of conjunction and implication. Thus, $(\mathcal{S}, \Psi)$ realizes φ as well and therefore $s_1 \mid\mid \dots \mid\mid s_n \models \varphi$ follows with Theorem 4.1.

Second, assume that $(\mathcal{S}, \mathcal{G})_{\mathcal{R}}$ realizes φ. Then, we have both $s_i \models_{\mathcal{G}_i^{\mathcal{R}}} \varphi_i$ and $\mathcal{T}_i \preceq \mathcal{T}_i^G$ for all $p_i \in P^-$, where $\mathcal{T}_i$ is the finite-state transducer representing s_i. Hence, for all $\gamma \in (2^{I_i})^\omega$ and all $\gamma' \in (2^{V \setminus V_i})^\omega$, either $comp(s_i, \gamma) \cup \gamma' \models \varphi_i$ holds or we have $comp(s_i, \gamma) \cup \gamma' \notin \mathcal{V}_{\mathcal{G}_i^{\mathcal{R}}}$. Since $\mathcal{R}_i \subseteq P^- \setminus \{p_i\}$ holds, $\mathcal{G}_i^{\mathcal{R}} \subseteq \mathcal{G}_i$ follows. Therefore, it follows immediately with the definition of valid computations that, if $comp(s_i, \gamma) \cup \gamma' \notin \mathcal{V}_{\mathcal{G}_i^{\mathcal{R}}}$ holds, then $comp(s_i, \gamma) \cup \gamma' \notin \mathcal{V}_{\mathcal{G}_i}$ holds as well. Thus, we have both $s_i \models_{\mathcal{G}_i} \varphi_i$ and $\mathcal{T}_i \preceq \mathcal{T}_i^G$ for all $p_i \in P^-$ and therefore $(\mathcal{S}, \mathcal{G})$ realizes φ. Thus, $s_1 \mid\mid \ldots \mid\mid s_n \models \varphi$ follows with Theorem 4.2. □

Next, we consider the variants of certifying synthesis with transition-incomplete strategies, i.e., with local strategies and augmented local strategies. Intuitively, the fact that $\mathcal{R}_i \subseteq P^- \setminus \{p_i\}$ holds for every system process $p_i \in P^-$ again allows for concluding soundness: local strategies and augmented local strategies realize the specification for every input sequence on which they produce infinite traces. They produce infinite traces on input sequences that match the guaranteed behavior of the relevant processes and thus, in particular, they produce infinite traces on all input sequences that match the guaranteed behavior of all other system processes. Formally, however, we cannot simply use the local and augmented local strategies for certifying synthesis when considering all other system processes again since they do not adhere to the definition of local strategies with respect to all other guarantee transducers.

Hence, the soundness proofs for these types of certifying synthesis differ in their structure from those for certifying synthesis with complete strategies. First, we focus on certifying synthesis with local strategies. The main idea is to extend the local strategies with strategy extension as described in Definition 4.9 to complete strategies. These strategies build a solution to certifying synthesis with complete strategies, guarantee transducers, and relevant processes. Furthermore, the computations of their parallel composition coincide with the computations of the parallel composition of the local strategies. Thus, it follows with Lemma 4.11 that the parallel composition of the local strategies realizes the specification. Formally:

Lemma 4.12. *Let φ be an LTL formula over atomic propositions V with decomposition $\langle \varphi_1, \ldots, \varphi_n \rangle$. Let $\mathcal{G} = \langle \mathcal{T}_1^G, \ldots, \mathcal{T}_n^G \rangle$ be a vector of guarantee transducers for the system processes. For $p_j \in P^-$, let $\mathcal{G}_j^{\mathcal{R}} = \left\{ \mathcal{T}_i^G \mid p_i \in \mathcal{R}_j \right\}$. Let $\hat{\mathcal{S}} = \langle \hat{s}_1, \ldots, \hat{s}_n \rangle$ be a vector such that $\hat{s}_i$ is a local strategy for $p_i \in P^-$ with respect to $\mathcal{G}_i^{\mathcal{R}}$. If $(\hat{\mathcal{S}}, \Psi)_{\mathcal{R}}$ realizes φ, then $\hat{s}_1 \mid\mid \ldots \mid\mid \hat{s}_n \models \varphi$ holds.*

Proof. For $p_j \in P^-$, let $\mathcal{G}_j = \left\{ \mathcal{T}_i^G \mid p_i \in P^- \setminus \{p_i\} \right\}$. Let $\hat{\mathcal{T}}_i = (\hat{T}_i, \hat{T}_{i,0}, \hat{\tau}_i, \hat{\ell}_i)$ be the finite-state transducer representing $\hat{s}_i$. Assume that $(\hat{\mathcal{S}}, \Psi)_{\mathcal{R}}$ realizes φ. Then, $\hat{\mathcal{T}}_i \preceq \mathcal{T}_i^G$ holds for all $p_i \in P^-$. Furthermore, for all $p_i \in P^-$ and all $\gamma \in (2^{I_i})^\omega$, $\gamma' \in (2^{V \setminus V_i})^\omega$, either $comp(\hat{s}_i, \gamma)$ is finite or $comp(\hat{s}_i, \gamma) \cup \gamma' \models \varphi_i$ holds. Let $\mathcal{S} = \langle s_1, \ldots, s_n \rangle$ be a vector of complete strategies such that $s_i := \text{extend}(\hat{s}_i, \mathcal{T}_i^G)$ holds for all system processes $p_i \in P^-$, i.e., we use strategy extension as defined in Definition 4.9 to extend each $\hat{s}_i$ to a complete strategy s_i. Let $\mathcal{T}_i = (T_i, T_{i,0}, \tau_i, \ell_i)$ be the finite-state transducer s_i. We claim that $(\mathcal{S}, \mathcal{G})_{\mathcal{R}}$ realizes φ. Let $p_i \in P^-$.

First, we show that $\mathcal{T}_i \preceq \mathcal{T}_i^G$ holds. Note that the first part of the proof of Lemma 4.4, i.e., the part that proves transducer simulation there, does not make use of the fact that there is a local strategy *with respect to* $\mathcal{G}_i$. Hence, since $(\hat{\mathcal{S}}, \Psi)_{\mathcal{R}}$ realizes φ by assumption, we can show similarly hat $\mathcal{T}_i \preceq \mathcal{T}_i^G$ holds.

Second, we show that $s_i \models_{\mathcal{G}_i} \varphi_i$ holds. Let $\gamma \in (2^{I_i})^\omega$ and $\gamma' \in (2^{V \setminus V_i})^\omega$. If $comp(\hat{s}_i, \gamma)$ is infinite then $comp(\hat{s}_i, \gamma) \cup \gamma' \models \varphi_i$ holds. Furthermore, by Lemma 4.3, we have $comp(\hat{s}_i, \gamma) = comp(s_i, \gamma)$. Thus, $comp(s_i, \gamma) \cup \gamma' \models \varphi_i$ follows. Otherwise, i.e., if $comp(\hat{s}_i, \gamma)$ is finite, then we can show similar to the proof of Lemma 4.4 that $comp(s_i, \gamma) \cup \gamma' \models_{\mathcal{G}_i^\mathcal{R}} \varphi_i$ holds.

Hence, $(\mathcal{S}, \mathcal{G})_\mathcal{R}$ indeed realizes φ and therefore $s_1 \mid\mid \ldots \mid\mid s_n \models \varphi$ follows with Lemma 4.11. We can show that $Traces(\mathcal{T}_1 \mid\mid \ldots \mid\mid \mathcal{T}_n) = Traces(\hat{\mathcal{T}}_1 \mid\mid \ldots \mid\mid \hat{\mathcal{T}}_n)$ holds similar to the third part of the proof of Lemma 4.4. In particular, the only difference is as follows: for $p_i \in P^-$, let $\hat{s}_i'$ be the local strategy with respect to $\mathcal{G}_i$ used in the proof of Lemma 4.4. In the second case of the induction, we use that $\hat{s}_i'$ is a local strategy with respect to $\mathcal{G}_i$ to conclude that $|comp(\hat{s}_i', \sigma \cap I_i)| > k$ holds from the fact that $\sigma \in \mathcal{V}_{\mathcal{G}_i}$ holds. If $\sigma \in \mathcal{V}_{\mathcal{G}_i}$ holds, however, then it follows immediately from the construction of $\mathcal{G}_i^\mathcal{R}$ that $\sigma \in \mathcal{V}_{\mathcal{G}_i^\mathcal{R}}$ holds as well. Hence, we can also conclude that $|comp(\hat{s}_i, \sigma \cap I_i)| > k$ holds. Since the remainder of the proof is not specific to local strategies with respect to the guarantee transducers of all other system processes, we do not need to alter it for local strategies with respect to the guarantee transducers of all relevant processes. Hence, we have $Traces(\mathcal{T}_1 \mid\mid \ldots \mid\mid \mathcal{T}_n) = Traces(\hat{\mathcal{T}}_1 \mid\mid \ldots \mid\mid \hat{\mathcal{T}}_n)$ and thus $\hat{s}_1 \mid\mid \ldots \mid\mid \hat{s}_n \models \varphi$ follows. $\square$

For certifying synthesis with augmented local strategies, recall that, by Lemma 4.8, an augmented local strategy for a system process $p_i \in P^-$ with respect to a set $\mathcal{G}$ of guarantee transducers satisfies the same properties as a local strategy for p_i with respect to $\mathcal{G}$ with respect to finiteness and infiniteness of computations. Thus, as argued for the soundness of certifying synthesis with augmented local strategies when considering the guarantee transducers of all other system processes, it follows that the strategy local($\tilde{s}_i$), which is a copy of the augmented local strategy $\tilde{s}_i$ with respect to $\mathcal{G}$ that restricts the output to O_i, is a *local* strategy with respect to $\mathcal{G}$. Furthermore, simulation by the guarantee transducer for p_i is preserved by construction and, since augmented local strategies only produce well-defined traces, the computations of the augmented local strategy $\tilde{s}_i$ and local($\tilde{s}_i$) agree for all input sequences. Therefore, the computations of $\tilde{s}_1 \mid\mid \ldots \mid\mid \tilde{s}_n$ and local($\tilde{s}_1$) $\mid\mid \ldots \mid\mid$ local($\tilde{s}_n$) agree as well by Proposition 4.1. Thus, soundness of certifying synthesis with augmented local strategies and relevant processes follows from Lemmas 4.8 and 4.12, and the above observation.

Lemma 4.13. *Let φ be an LTL formula over atomic propositions V with decomposition $\langle \varphi_1, \ldots, \varphi_n \rangle$. Let $\mathcal{G} = \langle \mathcal{T}_1^G, \ldots, \mathcal{T}_n^G \rangle$ be a vector of guarantee transducers for the system processes. For $p_j \in P^-$, let $\mathcal{G}_j^\mathcal{R} = \{\mathcal{T}_i^G \mid p_i \in \mathcal{R}_j\}$. Let $\tilde{\mathcal{S}} = \langle \tilde{s}_1, \ldots, \tilde{s}_n \rangle$ be a vector such that $\tilde{s}_i$ is an augmented local strategy for $p_i \in P^-$ with respect to $\mathcal{G}_i^\mathcal{R}$. If $(\tilde{\mathcal{S}}, \Psi)_\mathcal{R}$ realizes φ, then $\tilde{s}_1 \mid\mid \ldots \mid\mid \tilde{s}_n \models \varphi$ holds.*

Next, we consider completeness. Recall that when considering the certificates of *all* other system processes in the previous sections, we showed that if $s_1 \mid\mid \ldots \mid\mid s_n \models \varphi$ holds, then there is a vector Ψ of LTL certificates and a vector $\mathcal{G}$ of guarantee transducers such that both $(\mathcal{S}, \Psi)$ and $(\mathcal{S}, \mathcal{G})$ realize φ, where $\mathcal{S} = \langle s_1, \ldots, s_n \rangle$. When considering only the certificates of relevant processes, however, we cannot prove this exact property: a strategy s_i for system process $p_i \in P^-$ may make use of a certificate of a system process $p_j \in P^- \setminus (\mathcal{R}_i \cup \{p_i\})$ outside of $\mathcal{R}_i$, i.e., it may violate its specification φ_i on an input sequence $\gamma \in (2^{I_i})^\omega$ that deviates from p_j's guaranteed behavior, although φ_i is satisfiable for this input. While s_i is not required to satisfy φ_i on this input since only $s_1 \mid\mid \ldots \mid\mid s_n \models \varphi$ needs to hold, a strategy for p_i that may only consider the certificates of relevant processes, in contrast, is. In this case, s_i does not satisfy the requirements of certifying synthesis when only considering relevant certificates.

However, we can show that if $s_1 \mid\mid \ldots \mid\mid s_n \models \varphi$ holds, then there are *some strategies* $s'_1, \ldots, s'_n$ such that we can construct certificates that, together with $\mathcal{S}' = \langle s'_1, \ldots, s'_n \rangle$, form a solution of certifying synthesis for φ. The main idea is to construct a strategy s'_i for the system processes $p_i \in P^-$ that behaves on *every input sequence* as s_i does on input sequences that can occur in the parallel composition of all strategies. Since the parallel composition of all strategies realizes φ by assumption, the strategies s'_i do so on all input sequences that match the relevant certificates. Note, however, that this construction requires a slightly more restrictive assumption than observation determinism: every process $p_i \in P^-$ needs to be able to observe all environment outputs or, if it does not, it needs to be able to observe all environment outputs that occur in its specification and it may not have any relevant processes. First, we show completeness of certifying synthesis with relevant processes for full strategies and guarantee transducers:

Lemma 4.14. *Let φ be an LTL formula over atomic propositions V with decomposition $\langle \varphi_1, \ldots, \varphi_n \rangle$. Let $s_1, \ldots, s_n$ be strategies for the system processes represented by finite-state transducers $\mathcal{T}_1, \ldots, \mathcal{T}_n$. Suppose that for all $p_i \in P^-$ either (i) $O_{env} \subseteq I_i$ holds, or (ii) we have $\mathcal{R}_i = \emptyset$ and $O^- \cap I_i = \emptyset$. If $s_1 \mid\mid \ldots \mid\mid s_n \models \varphi$ holds, then there exist vectors $\mathcal{S}'$, Ψ' of strategies and LTL certificates for the system processes such that $(\mathcal{S}', \Psi')_{\mathcal{R}}$ realizes φ.*

Proof. Let $\mathcal{S} = \langle s_1, \ldots, s_n \rangle$. For $p_j \in P^-$, let $\mathcal{T}_j = (T_j, T_{j,0}, \tau_j, \ell_j)$ and $\mathcal{G}_j = \{\mathcal{T}_i^G \mid p_i \in P^- \setminus \{p_j\}\}$. Assume that $s_1 \mid\mid \ldots \mid\mid s_n \models \varphi$ holds and let $\mathcal{T} = (T, T_0, \tau, \ell)$ be the transducer representing $s_1 \mid\mid \ldots \mid\mid s_n$, i.e., we have $\mathcal{T} = \mathcal{T}_1 \mid\mid \ldots \mid\mid \mathcal{T}_n$. Then, by Theorem 4.1, there exists a vector $\mathcal{G}$ of guarantee transducers for the system processes such that $(\mathcal{S}, \mathcal{G})$ realizes φ.

We construct strategies $s'_1, \ldots, s'_n$ represented by transducers $\mathcal{T}_i' = (T'_i, t'_{i,0}, \tau'_i, \ell'_i)$ as follows: for each system process $p_i \in P^-$ with $\mathcal{R}_i = \emptyset$ and $O^- \cap I_i = \emptyset$, let s'_i be a copy of s_i, i.e., the transducer $\mathcal{T}_i'$ representing s'_i is given by $\mathcal{T}_i' := (T_i, T_{i,0}, \tau, \ell)$. For each system process $p_i \in P^-$ with either $\mathcal{R}_i \neq \emptyset$ or $O^- \cap I_i \neq \emptyset$, let $\mathcal{T}_i' = (T'_i, t'_{i,0}, \tau'_0, \ell')$ be the transducer defined by

- $T'_i = T$,
- $T'_{i,0} = T_0$,
- $(t, \iota, t') \in \tau'_i$ if, and only if, $(t, \iota \cap O_{env}, t') \in \tau$ holds, and
- $(t, o) \in \ell'_i$ if, and only if, there exists some $o' \in 2^{O^-}$ with $o \cap O_i = o$ and $(t, o') \in \ell$.

Note that since either $\mathcal{R}_i \neq \emptyset$ or $O^- \cap I_i \neq \emptyset$ holds, we have $O_{env} \subseteq I_i$ by assumption. Thus, $\iota \cap O_{env}$ defines the valuation of all environment outputs and therefore τ'_i is well-defined. Let $\mathcal{S}' := \langle s'_1, \ldots, s'_n \rangle$. Furthermore, for each $p_i \in P^-$, let $\mathcal{T}_i^{G'}$ be the guarantee transducer that is a copy of $\mathcal{T}_i'$ which restricts the output to the guarantee outputs of p_i. Let $\mathcal{G}' := \langle \mathcal{T}_1^{G'}, \ldots, \mathcal{T}_n^{G'} \rangle$ and, for $p_j \in P^-$, let $\mathcal{G}'_{\mathcal{R},j} := \{\mathcal{T}_i^{G'} \mid p_i \in \mathcal{R}_j\}$. In the following, we show that $(\mathcal{S}', \mathcal{G}')_{\mathcal{R}}$ realizes φ, i.e., that both $s'_i \models_{\mathcal{G}'_{\mathcal{R},i}} \varphi_i$ and $\mathcal{T}_i' \preceq \mathcal{T}_i^{G'}$ hold for all $p_i \in P^-$. Let $p_i \in P^-$. It follows immediately from the construction of $\mathcal{T}_i^{G'}$ that $\mathcal{T}_i' \preceq \mathcal{T}_i^{G'}$ holds. Next, let $\gamma \in (2^{I_i})^\omega$, let $\gamma' \in (2^{V \setminus V_i})^\omega$, and let $\sigma := comp(s'_i, \gamma) \cup \gamma'$. If $\sigma \in Traces(\mathcal{T})$ holds, then, since $s_1 \mid\mid \ldots \mid\mid s_n \models \varphi$ by assumption, it follows from the construction of $\mathcal{T}$ that we have $\sigma \models \varphi$. Thus, by definition of specification decomposition and by the semantics of conjunction, $\sigma \models \varphi_i$ holds as well. Otherwise, we have $\sigma \notin \mathit{Traces}(\mathcal{T})$. We distinguish two cases:

First, suppose that $\mathcal{R}_i = \emptyset$ and $O^- \cap I_i = \emptyset$ hold. Then, s_i' is a copy of s_i by construction. Let $\sigma' \in \mathit{Traces}(\mathcal{T})$ be the trace of $\mathcal{T}$ such that $\sigma' \cap O_{env} = \sigma \cap O_{env}$ holds. Since $O^- \cap I_i = \emptyset$ holds by assumption, p_i cannot react to the outputs of any other system process and thus, in particular, $\sigma' \cap I_i = \sigma \cap I_i$ holds as well. Hence, $\sigma \cap O_i = \sigma' \cap O_i$ follows with the definition of computations. Furthermore, since $\mathcal{R}_i = \emptyset$ holds, we have $\mathit{prop}(\varphi_i) \cap O_j = \emptyset$ for all $p_j \in P^- \setminus \{p_i\}$ and thus $\mathit{prop}(\varphi_i) \subseteq O_{env} \cup V_i$ holds. Since σ and σ' agree on all variables in O_{env}, in I_i, and in O_i as shown above, we thus have $\sigma \cap \mathit{prop}(\varphi_i) = \sigma' \cap \mathit{prop}(\varphi_i)$. By construction, we have $\sigma' \in \mathit{Traces}(\mathcal{T})$ and therefore, since $s_1 \mid\mid \ldots \mid\mid s_n \models \varphi$ holds by assumption, we have $\sigma' \models \varphi$ and thus $\sigma' \models \varphi_i$. By definition, the satisfaction of φ_i is only affected by variables in $\mathit{prop}(\varphi_i)$. Thus, $\sigma \models \varphi_i$ follows and hence $\sigma \models \Psi'_{\mathcal{R},i} \rightarrow \varphi_i$ holds by the semantics of implication as well.

Second, suppose that either $\mathcal{R}_i \neq \emptyset$ or $O^- \cap I_i \neq \emptyset$ holds. Then, since $\sigma \cap V_i \in \mathit{Traces}(\mathcal{T}_i')$ by definition of σ, there exists, by construction of $\mathcal{T}_i'$, some trace $\sigma' \in \mathit{Traces}(\mathcal{T})$ such that $\sigma \cap O_{env} = \sigma' \cap O_{env}$ and $\sigma \cap O_i = \sigma' \cap O_i$ hold. Furthermore, since $s_1 \mid\mid \ldots \mid\mid s_n \models \varphi$ holds by assumption, $\sigma' \models \varphi$ and thus, in particular, $\sigma' \models \varphi_i$ follows. The satisfaction of φ_i is only affected by variables occurring in φ_i. Hence, if $\sigma \cap \mathit{prop}(\varphi_i) = \sigma' \cap \mathit{prop}(\varphi_i)$ holds, then we have $\sigma \models \varphi_i$ as well and thus $\sigma \models_{\mathcal{G}'_{\mathcal{R},i}} \varphi_i$ follows. Otherwise, σ and σ' disagree on some variable in $\mathit{prop}(\varphi_i)$. Since both $\sigma \cap O_{env} = \sigma' \cap O_{env}$ and $\sigma \cap O_i = \sigma' \cap O_i$ hold, σ and σ' disagree on a variable $v \in \mathit{prop}(\varphi_i) \cap \bigcup_{p_j \in P^- \setminus \{p_i\}} O_j$. By definition of relevant processes, we have $O_j \cap \mathit{prop}(\varphi_i) = \emptyset$ for all $p_j \in P^- \setminus (\mathcal{R}_i \cup \{p_i\})$ and thus $v \in \mathit{prop}(\varphi_i) \cap \bigcup_{p_j \in \mathcal{R}_i} O_j$ follows. Let $p_j \in \mathcal{R}_i$ such that $v \in O_j$ holds. Then, in particular, $\sigma \cap O_j \neq \sigma' \cap O_j$ holds. Since $\sigma' \in \mathit{Traces}(\mathcal{T})$ holds by construction, we have $\sigma' \cap V_j \in \mathit{Traces}(\mathcal{T}_j)$ by Proposition 4.1 and, in particular, $\sigma' \cap V_j \in \mathit{Traces}(\mathcal{T}_j, \sigma' \cap I_j)$. Since $\mathcal{T}_j$ is deterministic, $\sigma \cap V_j \notin \mathit{Traces}(\mathcal{T}_j, \sigma' \cap I_j)$ follows. If we have $\mathcal{R}_j = \emptyset$ and $O^- \cap I_j = \emptyset$, then, p_j's behavior is only affected by environment outputs. Since $\sigma \cap O_{env} = \sigma' \cap O_{env}$ holds as shown above, thus $\sigma \cap V_j \notin \mathit{Traces}(\mathcal{T}_j, \sigma \cap I_j)$ follows. By construction of $\mathcal{T}_j'$ and $\mathcal{T}_j^{G'}$, we then also have $\sigma \cap V_j \notin \mathit{Traces}(\mathcal{T}_j^{G'}, \sigma \cap I_j)$ and hence $\sigma \notin \mathcal{V}_{\mathcal{G}'_{\mathcal{R},j}}$ follows since we have $p_j \in \mathcal{R}_i$ by construction. Hence, $\sigma \models_{\mathcal{G}'_{\mathcal{R},j}} \varphi_i$ holds. Otherwise, suppose that $\sigma \cap V_j \in \mathit{Traces}(\mathcal{T}_j', \sigma \cap I_j)$ holds. Then, by construction of $\mathcal{T}_j'$, there exists some $\sigma'' \in \mathit{Traces}(\mathcal{T})$ such that $\sigma' \cap O_{env} = \sigma'' \cap O_{env}$ as well as $\sigma' \cap O_j = \sigma'' \cap O_j$ holds. Since $\sigma \cap O_{env} = \sigma' \cap O_{env}$ holds, we have $\sigma \cap O_{env} = \sigma'' \cap O_{env}$ as well. Since $\mathcal{T}$ is deterministic as shown above and since its set of input variables is given by O_{env} by construction, $\sigma = \sigma''$ follows. But then $\sigma \cap O_j = \sigma' \cap O_j$ holds, contradicting that σ and σ' differ on some output variable of p_j. Hence, $\sigma \cap V_j \notin \mathit{Traces}(\mathcal{T}_j', \sigma \cap I_j)$ holds. Thus, since $\mathcal{T}_j'$ is deterministic, we have $\sigma \cap V_j \neq \mathit{comp}(\mathcal{T}_j', \sigma \cap I_j)$. By construction of $\mathcal{T}_j^{G'}$, thus $\sigma \cap V_j \neq \mathit{comp}(\mathcal{T}_j^{G'}, \sigma \cap I_j)$ holds and hence $\sigma \cap O_j \neq \mathit{comp}(\mathcal{T}_j^{G'}, \sigma \cap I_j) \cap O_j$ holds as well. Therefore, $\sigma \notin \mathcal{V}_{\mathcal{G}'_{\mathcal{R},i}}$ and thus $\sigma \models_{\mathcal{G}'_{\mathcal{R},i}} \varphi_i$ follows. □

Recall that, as shown in the proof of Lemma 4.1, a guarantee transducer $\mathcal{T}_i^G$ for system process $p_i \in P^-$ can be translated into an LTL certificate ψ_i. Analogous to the proof of Lemma 4.1, we can conclude that whenever a strategy s_i for process $p_i \in P^-$ realizes a specification φ_i with respect to the set $\mathcal{G}_i^{\mathcal{R}}$ of guarantee transducers of the relevant processes of p_i, then s_i realizes the formula $\Psi_i^{\mathcal{R}} \rightarrow \varphi_i$ as well, where $\Psi_i^{\mathcal{R}}$ is the set of LTL certificates of the of the relevant processes of p_i constructed from the guarantee transducers. Hence, together with Lemma 4.14, it follows that certifying synthesis is also conditionally complete for LTL certificates when only considering the certificates of relevant processes:

Lemma 4.15. *Let φ be an LTL formula over atomic propositions V with decomposition $\langle \varphi_1, \dots, \varphi_n \rangle$. Let $s_1, \dots, s_n$ be strategies for the system processes represented by finite-state transducers $\mathcal{T}_1, \dots, \mathcal{T}_n$. Suppose that for all $p_i \in P^-$, either (i) $O_{env} \subseteq I_i$ holds, or (ii) we have $\mathcal{R}_i = \emptyset$ and $O^- \cap I_i = \emptyset$. If $s_1 \mid\mid \dots \mid\mid s_n \models \varphi$ holds, then there exist vectors $\mathcal{S}'$, Ψ of strategies and LTL certificates for the system processes such that $(\mathcal{S}', \Psi)_{\mathcal{R}}$ realizes φ.*

Next, recall that we can restrict complete strategies to local strategies with strategy restriction as defined in Definition 4.10. Thus, in particular, we can restrict the full strategy s_i for a system process $p_i \in P^-$ to the set $\mathcal{G}_i^{\mathcal{R}}$ of guarantee transducers of the relevant processes of p_i. By Lemma 4.5, the resulting strategy $\hat{s}_i := \text{restrict}(s_i, \mathcal{G}_i^{\mathcal{R}})$ is indeed a local strategy with respect to $\mathcal{G}_i^{\mathcal{R}}$. Furthermore, by construction, the transducer representing $\hat{s}_i$ is simulated by the guarantee transducer for p_i as long as the transducer representing the full strategy s_i is. Analogous to the proof of Lemma 4.7, we can thus show that if the full strategy s_i satisfies φ with respect to $\mathcal{G}_i^{\mathcal{R}}$ and if we have $prop(\varphi_i) \subseteq V_i$ holds, then $\hat{s}_i$ realizes φ_i on all inputs on which it produces infinite computations. Therefore, together with Lemma 4.14, it follows that certifying synthesis is also conditionally complete for local strategies when only considering the certificates of relevant processes:

Lemma 4.16. *Let φ be an LTL formula over atomic propositions V with decomposition $\langle \varphi_1, \dots, \varphi_n \rangle$. Let $s_1, \dots, s_n$ be strategies for the system processes represented by finite-state transducers $\mathcal{T}_1, \dots, \mathcal{T}_n$. Suppose that for all $p_i \in P^-$, either (i) $O_{env} \subseteq I_i$ holds, or (ii) we have $\mathcal{R}_i = \emptyset$ and $O^- \cap I_i = \emptyset$. If we have $s_1 \mid\mid \dots \mid\mid s_n \models \varphi$ and if $prop(\varphi_i) \subseteq V_i$ holds for all $p_i \in P^-$, then there exists a vector $\mathcal{G} = \langle \mathcal{T}_1^G, \dots, \mathcal{T}_m^G \rangle$ of guarantee transducers and a vector $\hat{\mathcal{S}} = \langle \hat{s}_1, \dots, \hat{s}_n \rangle$ such that $\hat{s}_i$ is a local strategy for $p_i \in P^-$ and $\mathcal{G}_i^{\mathcal{R}}$, where $\mathcal{G}_i^{\mathcal{R}} = \left\{ \mathcal{T}_j^G \mid p_j \in \mathcal{R}_i \right\}$, and such that $(\hat{\mathcal{S}}, \mathcal{G})_{\mathcal{R}}$ realizes φ.*

Lastly, recall that if observation determinism as formalized in Definition 4.13 is ensured, then we can build an augmented local strategy from a complete strategy similar to strategy restriction, yet, adding the associated outputs to the labeling function. In particular, similar to above, we can use restriction to the set $\mathcal{G}_i^{\mathcal{R}}$ of guarantee transducers of the relevant processes of p_i. Similar to the proof of Lemma 4.10, it follows that the resulting transducers indeed represent augmented local strategies. Furthermore, when restricting the transducers to the outputs of the corresponding process, we obtain transducers that are identical to those constructed with strategy restriction. Since we used these transducers for the proof of Lemma 4.16, they form a solution of certifying synthesis with local strategies and relevant processes. Hence, analogous as in the proof of Lemma 4.10, it follows that the augmented local strategies form a solution of certifying synthesis with augmented local strategies and relevant processes as well. Thus, conditional completeness follows. Note here that observation determinism is already ensured by the requirement that $O_{env} \subseteq I_i$ holds. Therefore we do not need to state it separately for such system processes $p_i \in P^-$.

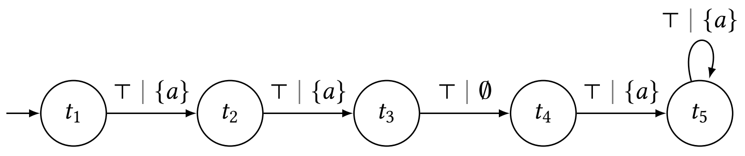
Lemma 4.17. *Let φ be an LTL formula over atomic propositions V with decomposition $\langle \varphi_1, \dots, \varphi_n \rangle$. Let $s_1, \dots, s_n$ be strategies for the system processes represented by finite-state transducers $\mathcal{T}_1, \dots, \mathcal{T}_n$. Suppose that for all $p_i \in P^-$, either (i) $O_{env} \subseteq I_i$ holds, or (ii) $\left\{\mathcal{T}_j \mid p_j \in P^- \setminus \{p_i\}\right\}$ is observation-deterministic for p_i and we have $\mathcal{R}_i = \emptyset$ and $O^- \cap I_i = \emptyset$. If $s_1 \mid\mid \dots \mid\mid s_n \models \varphi$ holds and if we have $prop(\varphi_i) \subseteq V_i$ for all $p_i \in P^-$, then there exist vectors $\mathcal{G} = \langle \mathcal{T}_1^G, \dots, \mathcal{T}_m^G \rangle$ and $\tilde{\mathcal{S}} = \langle \tilde{s}_1, \dots, \tilde{s}_n \rangle$ of guarantee transducers such that $\tilde{s}_i$ is a local strategy for $p_i \in P^-$ and $\mathcal{G}_i^{\mathcal{R}}$, where $\mathcal{G}_i^{\mathcal{R}} = \left\{\mathcal{T}_j^G \mid p_j \in \mathcal{R}_i\right\}$, and such that $(\tilde{\mathcal{S}}, \mathcal{G})_{\mathcal{R}}$ realizes φ.*

Therefore, it is sound and, under certain conditions, complete to consider only the certificates of relevant processes instead of the certificates of all other system processes in all variants of certifying synthesis that we introduced in the previous sections. Considering the certificates of only relevant processes can easily be integrated into the SAT constraint system from Section 4.4.4 that encodes the search for augmented local strategies and guarantee transducers satisfying the conditions of certifying synthesis. The only difference is that we do not require, for each $p_i \in P^-$, the existence of relations S_j^i establishing that the strategy is an augmented local strategy for *all other* system processes $p_j \in P^- \setminus \{p_i\}$ but only for the relevant processes $p_j \in \mathcal{R}_i$. In particular, we thus do not add $\bigwedge_{p_j \in P^- \setminus \{p_i\}}$ (4.4) for p_i to the SAT constraint system but $\bigwedge_{p_j \in \mathcal{R}_i}$ (4.4). This results in the following overall constraint system:

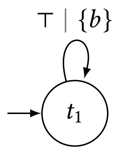
$$\bigwedge_{p_i \in P^-} \left((4.1) \land (4.2) \land \left(\bigwedge_{p_j \in \mathcal{R}_i} (4.4) \right) \land (4.3) \land (4.5) \right)$$

Correctness of the SAT constraint system under the stated conditions then follows immediately from the soundness and completeness results for certifying synthesis with augmented local strategies and relevant processes presented in this section, i.e., from Lemmas 4.13 and 4.17, as well as from Theorem 4.3. Note here that soundness of the SAT constraint system holds unconditionally, i.e., if the constraint system is satisfiable, then its solution defines augmented local strategies whose parallel composition realizes the given specification. Unconditional completeness, however, cannot be guaranteed: note that, as outlined in Section 4.4.1, moving from complete strategies with local satisfaction to local strategies makes the requirement that, for each process $p_i \in P^-$, all variables that are contained in p_i's subspecification are observable by p_i. We experienced, however, that this condition is often satisfied.

Furthermore, completeness of certifying synthesis with augmented local strategies and relevant processes and thus completeness of the SAT constraint system requires that every system process $p_i \in P^-$ can either observe all environment outputs or does not have relevant processes. This ensures that the guarantee transducers of other system processes are completely deterministic also from p_i's point of view, or it is not necessary to consider other processes' guarantees. While this limits the completeness result of the SAT constraint system to certain architectures and specifications, we experienced again that this condition is often satisfied. For instance, this is the case for all benchmarks considered in our experimental evaluation in Section 4.7. Furthermore, we did not encounter a benchmark so far, where certifying synthesis with relevant processes failed while classical certifying synthesis succeeded, even if the completeness condition on the architecture is violated. Hence, since the condition is only necessary



(a) Strategy s_1.



(b) Strategy s_2.

Figure 4.5.: Finite-state transducers representing the strategies s_1 and s_2 for the two system processes p_1 and p_2 from Example 4.9.

for completeness – soundness of certifying synthesis with relevant processes is guaranteed even if the condition is violated – trying to synthesize a solution with relevant processes only and only considering all other system processes if certifying synthesis with relevant processes fails still guarantees a correct solution.

4.6. Nondeterminism in Guarantee Transducers

In the previous sections, we focused on either LTL certificates or certificates represented by deterministic guarantee transducers. As outlined in Section 4.3.1, the former are not suitable for practical synthesis when aiming for integrating certifying synthesis into existing frameworks for constraint-based bounded synthesis. The latter, in contrast, can be integrated into such frameworks as presented in Section 4.4. While certifying synthesis with guarantee transducers is sound and complete as well (see Theorem 4.2), requiring the guarantee transducers to be *deterministic* can influence the conciseness of certificates.

Example 4.9. Consider a system with two system processes p_1 and p_2 with $O_1 = I_2 = \{a\}$ and $O_2 = I_1 = \{b\}$. Let $\varphi = a \leftrightarrow b \land a \land \bigcirc a \land \bigcirc\bigcirc a \land \bigcirc\bigcirc\bigcirc \neg a \land \bigcirc\bigcirc\bigcirc\bigcirc \square a$. Then, the decomposition of φ is given by $\langle \varphi_1, \varphi_2 \rangle$ with $\varphi_1 = \varphi$ and $\varphi_2 = a \leftrightarrow b$. To realize its specification φ_1, process p_1 only needs information about p_2's behavior in the very first time step. The behavior in all other time steps is irrelevant. Similarly, p_2 only needs information about p_1's behavior in the very first time step. Suppose that both processes guarantee to set their respective output variables to *true* in the first time step. Then an LTL certificate for p_1 could be given by $\psi_1 = a$ and an LTL certificate for p_2 by $\psi_2 = b$. These certificates only restrict the processes' behavior in the very first time step and is thus, in some sense, nondeterministic for the other time steps. Nevertheless, a strategy s_2 for p_2 that sets b to *true* in the very first time step realizes both $\psi_1 \rightarrow \varphi_2$ and ψ_2. A strategy s_1 for p_1 that sets a to *true* in the first time step and additionally ensures that the remaining parts of φ_1 are satisfied realizes both $\psi_2 \rightarrow \varphi_1$ and ψ_1. Examples of transducers representing such strategies are depicted in Figure 4.5.

When modeling certificates with deterministic finite-state transducers, however, the guarantee transducers need to uniquely determine the guaranteed behavior of the processes not only in the very first step but in all steps. Since guarantee transducer $\mathcal{T}_i^G$ for process p_i is required to simulate the transducer representing strategy s_i, it needs to explicitly spell out the behavior of s_1 regarding a in all time steps, resulting in a much less concise certificate. △

In this section, we, therefore, extend the notion of guarantee transducers with nondeterminism and adapt the notion of valid computation to nondeterministic guarantee transducers. Afterward, we define certifying synthesis with nondeterministic guarantee transducers and establish soundness and completeness. Lastly, we present a SAT constraint system for synthesizing augmented local strategies and nondeterministic guarantee transducers that satisfy the requirements of certifying synthesis. As part of the experimental evaluation of certifying synthesis presented in Section 4.7, we investigate the trade-off between having more concise nondeterministic certificates and the larger search space resulting from permitting multiple transitions and labels for the same state and input.

4.6.1. Synthesizing Nondeterministic Guarantee Transducers

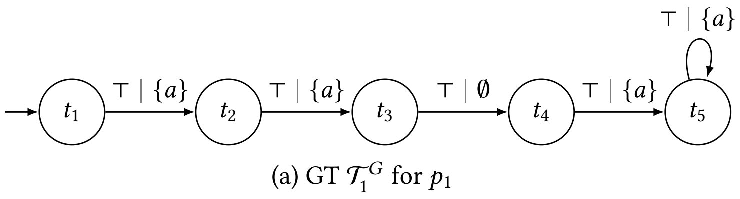
We model the certificate of a system process $p_i \in P^-$ with a complete and possibly nondeterministic finite-state Moore transducer $\mathcal{T}_i^G$, called *nondeterministic guarantee transducer* (NGT), over input variables I_i and guarantee output variables O_i^G. Since a nondeterministic guarantee transducer is complete, it produces *at least* one trace for every infinite input sequence $\gamma \in (2^{I_i})^\omega$, i.e., we have $|Traces(\mathcal{T}_i^G, \gamma)| \geq 1$. The concept of valid computations and valid histories for nondeterministic guarantee transducers is similar to the respective notion for deterministic guarantee transducers introduced in Section 4.3:

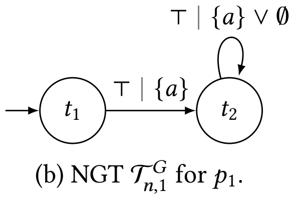
Definition 4.15 (Valid Computation and Valid History for NGTs)**.**
Let $\mathcal{P} \subseteq P^-$ be a finite set of system processes. Let $\mathcal{G}$ be a finite set of nondeterministic guarantee transducers, one for each of the processes in $\mathcal{P}$. An infinite sequence $\sigma \in (2^V)^\omega$ is called *valid computation* for $\mathcal{G}$ if, and only if $\sigma \cap O_i^G \in Traces(\mathcal{T}_i^G, \sigma \cap I_i) \cap O_i^G$ holds for all $\mathcal{T}_i^G \in \mathcal{G}$. The set of valid computations for $\mathcal{G}$ is denoted with $\mathcal{V}_\mathcal{G}$. A finite prefix $\rho \in (2^V)^*$ of length $k \geq 0$ of some valid computation $\sigma \in \mathcal{V}_\mathcal{G}$ is called *valid history* of length k for $\mathcal{G}$. The set of all valid histories of length k for $\mathcal{G}$ is denoted with $\mathcal{H}_\mathcal{G}^k$.

The notions of local satisfaction and local realization (see Definition 4.4) then carry over immediately when utilizing valid computations for nondeterministic guarantee transducers instead of valid computations for deterministic ones. We can thus define certifying synthesis with certificates represented by nondeterministic guarantee transducers similar to certifying synthesis with deterministic guarantee transducers (see Definition 4.6). The only difference is that we derive nondeterministic guarantee transducers instead of deterministic ones.

Example 4.10. Reconsider the system and the specification from Example 4.9. A deterministic guarantee transducer $\mathcal{T}_1^G$ for process p_1 is shown in Figure 4.6a, a nondeterministic one $\mathcal{T}_{n,1}^G$ in Figure 4.6b. Clearly, $\mathcal{T}_{n,1}^G$ is much more concise than $\mathcal{T}_1^G$.

The set of traces of $\mathcal{T}_{n,1}^G$ is given by $\{\rho \in (2^{\{a,b\}})^\omega \mid \rho_0 \cap \{a\} = \{a\}\}$, i.e., by the set of all sequences where a is set to *true* in the very first time step. Therefore, it follows immediately that strategy s_2 for process p_2 depicted in Figure 4.5b satisfies $\varphi_2 = a \leftrightarrow b$ with respect to $\mathcal{T}_{n,1}^G$, i.e., we have $s_2 \models_{\{\mathcal{T}_{n,1}^G\}} \varphi_2$. Furthermore, the relation $R = \{(t_1, t_1), (t_2, t_2), (t_3, t_2), (t_4, t_2), (t_5, t_2)\}$ establishes that $\mathcal{T}_1 \preceq \mathcal{T}_{n,1}^G$ holds, where $\mathcal{T}_1$ is the finite-state transducer representing strategy s_1





t_1
$\top \mid \{a\}$
t_2
$\top \mid \{a\} \vee \emptyset$
(b) NGT $\mathcal{T}_{n,1}^G$ for p_1.

Figure 4.6.: Deterministic and nondeterministic guarantee transducers $\mathcal{T}_1^G$ and $\mathcal{T}_{n,1}^G$ for process p_1 from Examples 4.9 and 4.10.

depicted in Figure 4.5a. A (deterministic) guarantee transducer $\mathcal{T}_2^G$ for p_2 that is a copy of the transducer $\mathcal{T}_2$ representing s_2 clearly ensures that both $\mathcal{T}_2 \preceq \mathcal{T}_2^G$ and $s_1 \models_{\{\mathcal{T}_2^G\}} \varphi_2$ hold. Hence, the strategies s_1 and s_2 from Figure 4.5 as well as the guarantee transducer $\mathcal{T}_2^G$ described above and the nondeterministic guarantee transducer depicted in Figure 4.6b constitute a solution of certifying synthesis with nondeterministic guarantee transducers. △

A valid computation for nondeterministic guarantee transducers must match *some* trace of every guarantee transducer of the other system processes. When requiring that a strategy s_i locally realizes a specification φ_i, we thus require that s_i satisfies φ_i on every input sequence that *can match* the behavior of the guarantee transducers in the sense that it matches some of the nondeterministic choices of the certificates. Employing nondeterministic guarantee transducers instead of deterministic ones in certifying synthesis thus, intuitively, makes local realization harder since a strategy needs to realize the specification irrespective of the actual nondeterministic choices of the certificates. Hence, the strategy might need to realize the specification, although it does not know to which concrete behavior the other processes commit.

Soundness of certifying synthesis with complete strategies and nondeterministic guarantee transducers thus follows from this observation since, intuitively, a solution $(\mathcal{S}, \mathcal{G})$ of certifying synthesis with nondeterministic guarantee transducers still ensures that every strategy s_i from $\mathcal{S}$ realizes the specification φ_i on all sequences that can occur in the interplay of all processes. Furthermore, since every guarantee transducer is, by definition, also a nondeterministic guarantee transducer, completeness follows from Theorem 4.2. Formally:

Theorem 4.4. *Let φ be an LTL formula over atomic propositions V with decomposition $\langle \varphi_1, \ldots, \varphi_n \rangle$. Let $\mathcal{S} = \langle s_1, \ldots, s_n \rangle$ be a vector of strategies for the system processes. Then, there exists a vector $\mathcal{G}$ of nondeterministic guarantee transducers for the system processes such that $(\mathcal{S}, \mathcal{G})$ realizes φ if, and only if, $s_1 \mid\mid \ldots \mid\mid s_n \models \varphi$ holds.*

Proof. Let $\mathcal{T}_i = (T_i, T_{i,0}, \tau_i, \ell_i)$ be the deterministic and complete finite-state transducer representing strategy s_i. Let $\mathcal{G} = \langle \mathcal{T}_1^G, \ldots, \mathcal{T}_n^G \rangle$ and, for $p_j \in P^-$, let $\mathcal{G}_j = \left\{ \mathcal{T}_i^G \mid p_i \in P^- \setminus \{p_j\} \right\}$. First, observe that, by definition, every guarantee transducer is a nondeterministic guarantee transducer as well. Thus, completeness of certifying synthesis with nondeterministic guarantee transducers follows immediately from Theorem 4.2. Next, suppose that there exists a vector $\mathcal{G} = \langle \mathcal{T}_1^G, \ldots, \mathcal{T}_n^G \rangle$ of nondeterministic guarantee transducers for the system processes such that $(\mathcal{S}, \mathcal{G})$ realizes φ. Then, for each $p_i \in P^-$, we have both $s_i \models_{\mathcal{G}_i} \varphi_i$ and

$\mathcal{T}_i \preceq \mathcal{T}_i^G$. Let $\sigma \in \mathit{Traces}(\mathcal{T}_1 \mid\mid \ldots \mid\mid \mathcal{T}_n)$. Then, by Proposition 4.1, we have $\sigma \cap V_i \in \mathit{Traces}(\mathcal{T}_i)$ for all $p_i \in P^-$. Hence, since $s_i \models_{\mathcal{G}_i} \varphi_i$ holds by assumption, for all $\gamma' \in (2^{V \setminus V_i})^\omega$, either $(\sigma \cap V_i) \cup \gamma' \models \varphi_i$ or $(\sigma \cap V_i) \cup \gamma' \notin \mathcal{V}_{\mathcal{G}_i}$ holds. Thus, in particular, for every $p_i \in P^-$, we have either $\sigma \models \varphi_i$ or $\sigma \notin \mathcal{V}_{\mathcal{G}_i}$. If $\sigma \notin \mathcal{V}_{\mathcal{G}_i}$ holds for some $p_i \in P^-$, however, then there exists some $p_j \in P^- \setminus \{p_j\}$ such that $\sigma \cap O_j \notin \mathit{Traces}(\mathcal{T}_i^G, \sigma \cap I_j) \cap O_j$ holds. As shown above, however, we have $\sigma \cap V_j \in \mathit{Traces}(\mathcal{T}_j)$. Furthermore, since $(\mathcal{S}, \mathcal{G})$ realizes φ by assumption, we have $\mathcal{T}_j \preceq \mathcal{T}_j^G$. Hence, $\sigma \cap V_j \in \mathit{Traces}(\mathcal{G}_j)$ follows with Proposition 4.1. By definition of traces, in particular $\sigma \cap V_j \in \mathit{Traces}(\mathcal{G}_j, \sigma \cap I_j)$ holds and therefore we have $\sigma \cap O_j \in \mathit{Traces}(\mathcal{T}_i^G, \sigma \cap I_j) \cap O_j$; contradicting that $\sigma \notin \mathcal{V}_{\mathcal{G}_i}$ holds due to p_j. Since we chose both $p_i \in P^-$ and $p_j \in P^- \setminus \{p_i\}$ arbitrarily, it thus follows that $\sigma \in \mathcal{V}_{\mathcal{G}_i}$ holds for all $p_i \in P^-$. Hence, it follows that $\sigma \models \varphi_i$ holds for all $p_i \in P^-$ and thus we have $\sigma \models \bigwedge_{p_i \in P^-} \varphi_i$. Therefore, by definition of specification decomposition and by the semantics of conjunction, $\sigma \models \varphi$ holds. □

To utilize existing bounded synthesis algorithms and frameworks, we introduced certifying synthesis with local strategies in Section 4.4. Since the definition of local strategies (see Definition 4.7) is based on valid histories and valid computations, it carries over to nondeterministic guarantees immediately when utilizing valid histories and valid computations for nondeterministic guarantee transducers instead of the versions for deterministic ones. Hence, we can define certifying synthesis with local strategies and certificates represented by nondeterministic guarantee transducers similar to certifying synthesis with local strategies deterministic guarantee transducers (see Definition 4.8). The only difference is that we derive nondeterministic guarantee transducers instead of deterministic ones.

Similar to the case with deterministic guarantee transducers, we can *restrict* complete strategies to local ones, and *extend* local strategies to full ones. Note that strategy restriction as defined in Definition 4.10 can already handle nondeterminism in guarantee transducers: the restricted strategy keeps track of the guarantee transducers of the other system processes in the second component of its state. For certain architectures, namely those where the current system process $p_i \in P^-$ cannot observe all input variables of the considered other processes in $\mathcal{P}$, the guaranteed behavior of the processes in $\mathcal{P}$ is already nondeterministic from p_i's point of view, although we only considered deterministic guarantee transducers in the previous parts of this chapter. Therefore, the definition of strategy restriction already uses sets of states in which the guarantee transducers can be in, and thus permitting actual nondeterminism in the guarantee transducers does not yield the need for change in the construction. Furthermore, the proofs of Lemmas 4.5 and 4.6 do not rely on the fact that guarantee transducers are deterministic. Thus, the results carry over to certifying synthesis with local strategies and nondeterministic guarantee transducers. Hence, completeness of certifying synthesis with local strategies follows exactly as for deterministic guarantee transducers.

Lemma 4.18. *Let φ be an LTL formula over atomic propositions V with decomposition $\langle \varphi_1, \ldots, \varphi_n \rangle$. Let $\mathcal{S} = \langle s_1, \ldots, s_n \rangle$ be a vector of strategies for the system processes. If $\mathit{prop}(\varphi_i) \subseteq V_i$ holds for all $p_i \in P^-$ and if $s_1 \mid\mid \ldots \mid\mid s_n \models \varphi$ holds, then there exist vectors $\mathcal{G} = \langle \mathcal{T}_1^G, \ldots, \mathcal{T}_n^G \rangle$ and $\hat{\mathcal{S}} = \langle \hat{s}_1, \ldots, \hat{s}_n \rangle$ of nondeterministic guarantee transducers and local strategies such that $\hat{s}_i$ is a local strategy for $p_i \in P^-$ with respect to $\mathcal{G}_i$, where $\mathcal{G}_i = \left\{ \mathcal{T}_j^G \mid p_j \in P^- \setminus \{p_i\} \right\}$, and such that $(\hat{\mathcal{S}}, \mathcal{G})$ realizes φ.*

However, strategy extension as defined in Definition 4.9, utilizes the fact that guarantee transducers are deterministic. The extended strategy for $p_i \in P^-$ keeps track of both p_i's local strategy and p_i's guarantee transducer. Since p_i can clearly always observe all of the inputs that are relevant for p_i's guarantee transducer, the construction does not take care of nondeterminism from p_i's point of view such as strategy restriction does. Thus, when using the same construction for strategy extension when considering nondeterministic guarantee transducers as when considering deterministic ones, the resulting transducer can be nondeterministic. Hence, since system strategies are required to be deterministic, it then does not represent a complete strategy. Note, however, that using the behavior of p_i's guarantee transducer to extend the local strategy instead of some random behavior is only necessary to ensure that the extended strategy is still simulated by the guarantee transducer. Whenever the extended strategy differs from the behavior of the local strategy, the considered sequence does not match the guarantee transducers of the other considered processes. Hence, by definition of local satisfaction, the strategy's computation does not need to satisfy the specification anyhow.

Therefore, we can alter the extension of the local strategy, i.e., the behavior of the extended strategy whenever the local strategy does not contain transition anymore, to obtain a deterministic strategy extension. As long as the extended strategy is simulated by the guarantee transducer, the requirements of certifying synthesis with complete strategies are still satisfied. In particular, we can thus pick one of the nondeterministic choices occurring in the nondeterministic guarantee transducer when building the strategy restriction. Intuitively, we thus resolve the transducer's nondeterminism while clearly maintaining the simulation. Then, the extended strategy is deterministic even when considering nondeterministic choices, and, as outlined above, the requirements of certifying synthesis with local strategies are still satisfied. In particular, the proof of Lemma 4.3 does not change at all since it only considers the part of the strategy extension that represents the local strategy, and the proof of Lemma 4.4 can be carried out analogously when utilizing the slightly altered version of strategy extension. Hence, soundness of certifying synthesis with local strategies follows exactly as for deterministic guarantee transducers.

Lemma 4.19. *Let φ be an LTL formula over atomic propositions V with decomposition $\langle \varphi_1, \dots, \varphi_n \rangle$. Let $\mathcal{G} = \langle \mathcal{T}_1^G, \dots, \mathcal{T}_n^G \rangle$ be a vector of nondeterministic guarantee transducers for the system processes and, for $p_j \in P^-$, let $\mathcal{G}_j = \{\mathcal{T}_i^G \mid p_i \in P^- \setminus \{p_j\}\}$. Let $\hat{\mathcal{S}} = \langle \hat{s}_1, \dots, \hat{s}_n \rangle$ be a vector of local strategies for the system processes such that $\hat{s}_i$ is a local strategy for $p_i \in P^-$ and $\mathcal{G}_i$. If $(\hat{\mathcal{S}}, \mathcal{G})$ realizes φ, then $\hat{s}_1 \mid\mid \dots \mid\mid \hat{s}_n \models \varphi$.*

Next, recall that, in order to practically identify valid computations, we augmented local strategies with associated output variables, i.e., the outputs of other processes that are also inputs of the current process, in Section 4.4.3. In particular, an augmented local strategy for process $p_i \in P^-$ is represented by a $(2^{I_i}, 2^{O_i \cup O_i^A})$-transducer, while a local strategy is represented by a $(2^{I_i}, 2^{O_i})$-transducer. Since system strategies are deterministic, this construction inherently enforces determinism of the guarantee transducers of the other processes and, in fact, even determinism from the current processes point of view, i.e., observation determinism. In the following, we thus slightly alter the definition of augmented local strategies. We represent an augmented local strategy $\tilde{s}_i$ for system process $p_i \in P^-$ as a deterministic $(2^{I_i}, 2^{O_i})$-transducer $\tilde{\mathcal{T}_i} = (\tilde{T}_i, \tilde{T}_{i,0}, \tilde{\tau}_i, \tilde{\ell}_i)$ and equip it with an additional labeling function $\ell_i^A : \tilde{T}_i \times O_i^A \rightarrow \{\top, \bot, ?\}$ that determines for each state $t \in T$ and each associated output $v \in O_i^A$ whether it is definitely *true* in t (encoded with $\top$), definitely *false* in t (encoded with $\bot$) or whether it can be both *true* and *false* in t (encoded with ?). For all $t \in \tilde{T}_i$ and all $\iota \in 2^{I_i}$, we then require that there is some $t' \in \tilde{T}_i$ with $(t, \iota, t') \in \tilde{\tau}_i$ if, and only if, it holds for all $v \in O_i^A$ that (i) if $v \in \iota$, then we have $\ell^A(t, v) \subseteq \{\top, ?\}$ and (ii) if $v \notin \iota$, then we have $\ell^A(t, v) \subseteq \{\bot, ?\}$. Furthermore, we again require the existence of relations S_j^i for all other considered processes p_j. Consequently, the definition of these relations is adapted in the straightforward manner to work with the additional labeling ℓ_i^A instead of the labeling $\tilde{\ell}_i$ restricted to the associated outputs.

We can then show analogously to the proof of Lemma 4.8 that an augmented local strategy as defined above satisfies the properties of local strategies for nondeterministic guarantee transducers regarding finiteness and infiniteness of computations. Hence, in particular, the transducer representing the augmented local strategy is a local strategy as well and therefore soundness of certifying synthesis with augmented local strategies and nondeterministic guarantee transducers follows, as in the deterministic case, from the respective soundness result for certifying synthesis with local strategies.

Lemma 4.20. *Let φ be an LTL formula over atomic propositions V and let $\langle \varphi_1, \dots, \varphi_n \rangle$ be its decomposition. Let $\mathcal{G} = \langle \mathcal{T}_1^G, \dots, \mathcal{T}_n^G \rangle$ be a vector of nondeterministic guarantee transducers for the system processes and, for $p_j \in P^-$, let $\mathcal{G}_j = \{\mathcal{T}_i^G \mid p_i \in P^- \setminus \{p_j\}\}$. Let $\tilde{\mathcal{S}} = \langle \tilde{s}_1, \dots, \tilde{s}_n \rangle$ be a vector of augmented local strategies for the system processes such that $\tilde{s}_i$ is an augmented local strategy for $p_i \in P^-$ with respect to $\mathcal{G}_i$. Let $\tilde{\mathcal{T}_i}$ be the finite-state transducer representing $\tilde{s}_i$. If, for all $p_i \in P^-$, both $\tilde{s}_i \models \varphi_i$ and $\tilde{\mathcal{T}_i} \preceq \mathcal{T}_i^G$ hold, then $\tilde{s}_1 \mid\mid \dots \mid\mid \tilde{s}_n \models \varphi$.*

Furthermore, recall that the proof of completeness of certifying synthesis with augmented local strategies and deterministic guarantee transducers, i.e., the proof of Lemma 4.10, constructs augmented local strategies similar to the construction of strategy restriction. The proof relies on the assumption that the strategies and the guarantee transducers are observation-deterministic, which yields that in the construction, the second part of the state of the transducer $\tilde{\mathcal{T}_i}$ representing the augmented local strategy is always a singleton. This, however, is only needed to ensure that, for every state of $\tilde{\mathcal{T}_i}$, there exists a *unique* labeling to obtain a deterministic transducer. Since the labeling of augmented local strategies as introduced in Section 4.4.3 contains the valuations of the associated outputs, we thus require a unique valuation of associated outputs. Hence, if the second part of the states of $\tilde{\mathcal{T}_i}$ would not be a singleton, we might obtain contradicting valuations of associated outputs. Outsourcing the associated outputs to an additional labeling function now enables to also handle non-singleton sets of states of the parallel composition of the guarantee transducers in the construction of the augmented local strategy without losing determinism. Since strategy extension already allows for nondeterminism in the guarantee transducers as outlined above, we can use the same construction and define the additional labeling ℓ^A in the straightforward manner. Note that the valuation of the associated outputs in the labeling of a state of the augmented local strategy does not influence whether or not it realizes a specification. Thus, we can show analogously to the proof of Lemma 4.10 that the augmented local strategies constructed in this way indeed form a solution of certifying synthesis with augmented local strategies and nondeterministic guarantee transducers, thus proving completeness. Observe that the new definition of augmented local strategies allows for dropping the requirement that the strategies are observation-deterministic.

Lemma 4.21. *Let φ be an LTL formula over atomic propositions V and let $\langle \varphi_1, \dots, \varphi_n \rangle$ be its decomposition. Let $s_1, \dots, s_n$ be strategies for the system processes represented by finite-state transducers $\mathcal{T}_1, \dots, \mathcal{T}_n$. If we have $prop(\varphi_i) \subseteq V_i$ for all $p_i \in P^-$, and if $s_1 \mid\mid \dots \mid\mid s_n \models \varphi$ holds, then there exists a vector $\mathcal{G} = \langle \mathcal{T}_1^G, \dots, \mathcal{T}_n^G \rangle$ of nondeterministic guarantee transducers for the system processes and a vector $\tilde{\mathcal{S}} = \langle \tilde{s}_1, \dots, \tilde{s_2} \rangle$ such that $\tilde{s}_i$ is an augmented local strategy for $p_i \in P^-$ with respect to $\mathcal{G}_i$, where $\mathcal{G}_i = \left\{ \mathcal{T}_j^G \mid p_j \in P^- \setminus \{p_i\} \right\}$, such that $(\tilde{\mathcal{S}}, \mathcal{G})$ realizes φ.*

Hence, when altering the definition of augmented local strategies as presented above, augmented local strategies can again be used to identify valid computations. Furthermore, certifying synthesis is both sound and complete when considering nondeterministic guarantee transducers. Therefore, we encode certifying synthesis with augmented local strategies and nondeterministic guarantee transducers into a SAT constraint system in the following to practically synthesize strategies and certificates in the following section.

4.6.2. Constraint System for Nondeterministic Certificates

To encode the search for augmented local strategies and nondeterministic guarantee transducers that adhere to the requirements of certifying synthesis into a SAT constraint system, we reuse most of the constraint system for the deterministic case presented in Section 4.4.4. We only need to adapt a few variable encodings and constraints. To incorporate the slight change in the definition of augmented local strategies, we first adapt the encoding of the labeling function of the finite-state transducer $\mathcal{T}_i = (T_i, T_{i,0}, \tau_i, \ell_i)$ representing the augmented local strategy as follows: we represent ℓ_i by one Boolean variable $o_{t,o}^{i}$ for each $t \in T_i$ and $o \in 2^{O_i}$. Given $t \in T_i$ and $o \in 2^{O_i}$, it holds that $o_{t,o}^{i}$ is *true* if, and only if, $\ell_i(t) = o$ holds. Thus, since an augmented local strategy is now a $(2^{I_i}, 2^{O_i})^\omega$-transducer instead of a $(2^{I_i}, 2^{O_i \cup O_i^A})$-transducer, we only have Boolean variables for the outputs of p_i.

Furthermore, we introduce variables to encode the additional labeling function ℓ_i^A: we represent ℓ_i^A by *two* Boolean variables $o_{t,o}^{i,\top}$ and $o_{t,o}^{i,\bot}$ for each $t \in T_i$ and $o \in O_i^A$. Given $t \in T_i$ and $o \in 2^{O_i^A}$, it holds that $o_{t,o}^{i,x}$ is *true* if, and only if, we have $\ell^A(t, o) \in \{x, ?\}$, where $x \in \{\top, \bot\}$. Thus, intuitively, $o_{t,o}^{i,\top}$ encodes that o can be set to *true* in state t, i.e., either it is definitely *true* or it can be both *true* and *false*. Similarly, $o_{t,o}^{i,\bot}$ encodes that o can be set to *false* in state t. Hence, if both $o_{t,o}^{i,\top}$ and $o_{t,o}^{i,\bot}$ are *true*, then there is no unique valuation of o in t.

Next, we adapt the constraints to the new variable encodings. Constraint (4.1) encodes determinism and completeness of guarantee transducers. Hence, we remove the conjunct that ensures that there is at most one outgoing transition for every state and every valuation of input variables; resulting in the following constraint:

$$\bigwedge_{u \in T_i^G} \bigwedge_{\iota \subseteq I_i} \bigvee_{u' \in T_i^G} \tau_{u,\iota,u'}^{G,i} \tag{4.6}$$

Constraint (4.2) encodes the existence of a simulation relation that establishes that the transducer representing the augmented local strategy for $p_i \in P^-$ is simulated by the guarantee transducer for p_i. Since the simulation is only defined on the output variables of p_i and not on its associated outputs, the constraint does not need to be changed.

Constraint (4.3) encodes condition (i) of the definition of augmented local strategies as well as determinism of the strategy. Hence, we adapt the conjunct for condition (i) to match the formalization in the slightly altered definition of augmented local strategies when considering nondeterministic guarantee transducers:

$$\begin{aligned}\bigwedge_{t\in T_i}\bigwedge_{\iota\subseteq I_i}\Bigg(\bigwedge_{o\in O_i^A}\left(o\in\iota\rightarrow o_{t,o}^{i,\top}\right)\wedge\left(o\notin\iota\rightarrow o_{t,o}^{i,\bot}\right)\leftrightarrow\bigvee_{t'\in T_i}\tau^i_{t,\iota,t'}\Bigg)\\ \wedge\bigwedge_{t'\in T_i}\bigwedge_{t''\in T_i\setminus\{t'\}}\neg\left(\tau^i_{t,\iota,t'}\wedge\tau^i_{t,\iota,t''}\right)\end{aligned}\tag{4.7}$$

Constraint (4.4) encodes condition (ii) of the definition of augmented local strategies, i.e., the existence of relations S_j^i for all other considered system processes p_j. Hence, we again need to adapt it to the slightly altered version of augmented local strategies for nondeterministic guarantee transducers:

$$\begin{aligned}S^{j,i}_{t^G_{j,0},t_{i,0}}\wedge\bigwedge_{u\in T_j^G}\bigwedge_{t\in T_i}\Bigg(S^{j,i}_{u,t}\rightarrow\Bigg(\bigwedge_{o\in O_i^A\cap O_j^G}\left(o\in o^{G,j}_{u,o}\rightarrow o^{i,\top}_{t,o}\right)\wedge\left(o\notin o^{G,j}_{u,o}\rightarrow o^{i,\bot}_{t,o}\right)\\ \wedge\bigwedge_{\iota\subseteq I_j}\bigwedge_{\iota'\subseteq I_i}\Bigg(\bigwedge_{o\in O_i^A}\left(o\in\iota'\rightarrow o^{i,\top}_{t,o}\right)\wedge\left(o\notin\iota'\rightarrow o^{i,\bot}_{t,o}\right)\\ \wedge\,\iota\cap I_i=\iota'\cap I_j\Bigg)\Bigg)\rightarrow\bigwedge_{u'\in T_j^G}\Bigg(\tau^{G,j}_{u,\iota,u'}\rightarrow\bigvee_{t'\in T_i}\left(\tau^i_{t,\iota',t'}\wedge S^{j,i}_{u',t'}\right)\Bigg)\Bigg)\end{aligned}\tag{4.8}$$

Lastly, constraint (4.5) encodes the existence of a valid annotation of the run graph of the universal co-Büchi automaton $\mathcal{A}_i$ representing the specification φ_i and the strategy $\mathcal{T}_i$. Since the constraint only considers the outputs of p_i, the change in the encoding of the labeling function does not affect this constraint. Hence, we do not need to alter it.

Combining all these constraints, we obtain the following constraint system $\mathbb{C}_{\mathscr{A},B,\varphi}$ for certifying synthesis with augmented local strategies and guarantee transducers:

$$\bigwedge_{p_i\in P^-}\Bigg((4.6)\wedge(4.2)\wedge\Bigg(\bigwedge_{p_j\in P^-\setminus\{p_i\}}(4.8)\Bigg)\wedge(4.7)\wedge(4.5)\Bigg)$$

Recall that the new definition of augmented local strategies allows for dropping the requirement that the strategies allow for observation determinism while still ensuring completeness.

Thus, redefining augmented local strategies immediately enables practical certifying synthesis when considering architectures that do not allow for observation determinism. Thus, we can use the constraint system introduced in this section also for synthesizing augmented local strategies and *deterministic* guarantee transducers that adhere to the requirements of certifying synthesis but violate observation determinism.

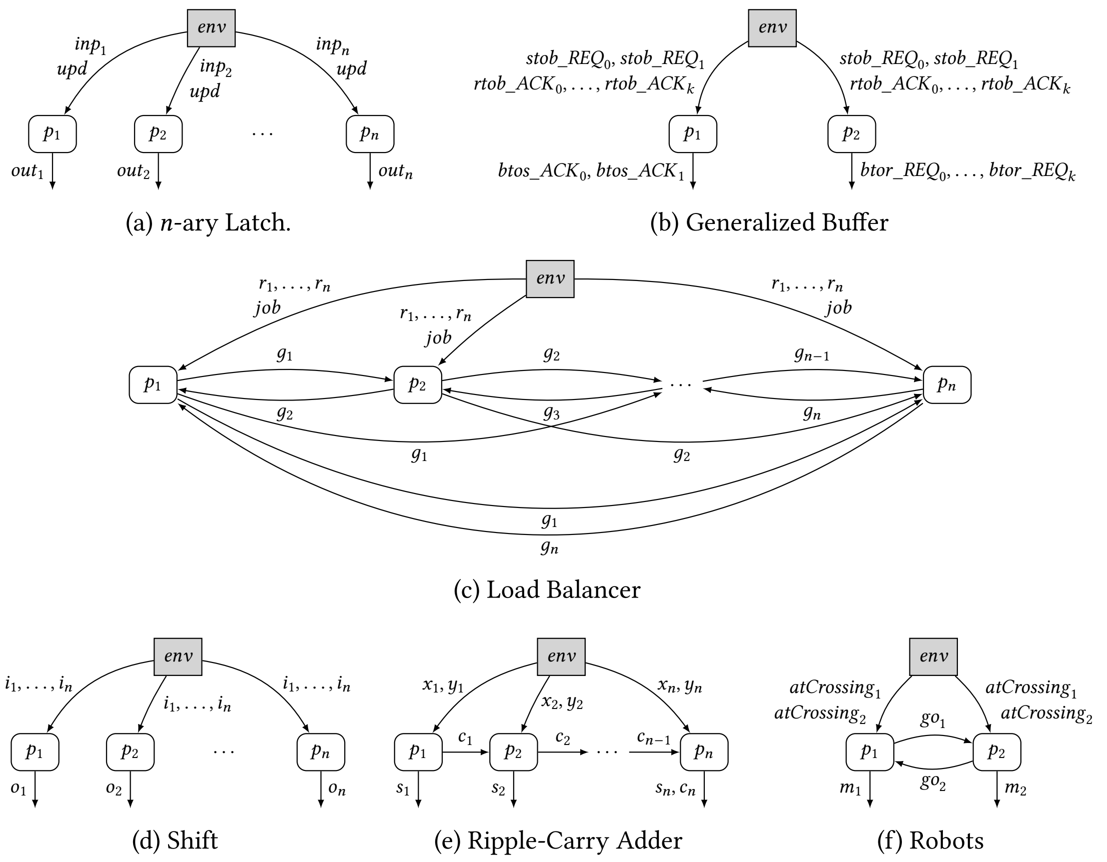
4.7. Experimental Evaluation

We have implemented certifying synthesis with augmented local strategies and both deterministic and nondeterministic guarantee transducers. Our implementation expects an LTL formula and its decompositions as well as the system architecture and the bounds on the strategy and certificate sizes as input. Our implementation extends BoSy [FFT17], a bounded synthesis tool for monolithic systems, to certifying synthesis for distributed systems. In particular, we extend and adapt BoSy's SAT encoding [FFRT17] as described in Sections 4.4.4 and 4.6.2, respectively. We evaluate our implementation in two lines of experiments. First, we compare certifying synthesis with deterministic guarantee transducers to two different synthesis approaches for distributed systems. Second, we compare the performance of our implementation of certifying synthesis with deterministic guarantee transducers to the one of certifying synthesis with nondeterministic guarantee transducers.

4.7.1. Distributed Synthesis

We compare our implementation of certifying synthesis with deterministic guarantee transducers to two extensions of BoSy [FFT17]: a non-compositional one for distributed systems [Bau17], and one for synthesizing remorsefree dominant strategies separately, implementing the compositional synthesis algorithm presented in [DF14]. We used a machine with a 3.1 GHz Dual-Core Intel Core i5 processor and 16 GB of RAM, and a timeout of 60 minutes. We use the SMT encoding of the non-composition distributed version of BoSy since the other ones either do not support most of our architectures (QBF), or cause memory errors frequently (SAT). Since the running times of the underlying SMT solver vary immensely, we report on the average running time over ten runs. Synthesizing dominant strategies separately is incomplete; thus, we cannot report on results for all benchmarks. We could not compare our algorithm to the iterative distributed synthesis tool Agnes [MMSZ20] since it currently is restricted to two-process architectures with safety or deterministic Büchi objectives. It thus does not support most of our system architectures and specifications.

First, we compare the performance of the implementations in terms of their running time on five different scalable benchmarks. Four of them, the *n-ary latch*, the *generalized buffer*, the *load balancer*, and *shift*, stem from the annual synthesis competition SyntComp [BEJ14, JBB+17b, JBB+15, JBB+16, JB16, JBB+17a, JBC+19, JPA+22] and the fifth one describes a ripple-carry adder. The latch is parameterized in the number of bits, the generalized buffer in the number of senders, the load balancer in the number of servers, and the shift in the number of inputs. The ripple-carry adder is parameterized in the number of bits.


env
inp_1, upd
inp_2, upd
inp_n, upd
p_1 p_2 ... p_n
out_1 out_2 out_n
$stob\_REQ_0$, $stob\_REQ_1$, $rtob\_ACK_0, \dots, rtob\_ACK_k$
$btos\_ACK_0$, $btos\_ACK_1$
$btor\_REQ_0, \dots, btor\_REQ_k$
$r_1, \dots, r_n$, job
g_1 g_2 g_3 g_{n-1} g_n
$i_1, \dots, i_n$
o_1 o_2 o_n
x_1, y_1 x_2, y_2 x_n, y_n
c_1 c_2 c_{n-1}
s_1 s_2 s_n, c_n
$atCrossing_1$, $atCrossing_2$
go_1 go_2
m_1 m_2

(a) n-ary Latch. (b) Generalized Buffer

(c) Load Balancer

(d) Shift (e) Ripple-Carry Adder (f) Robots

Figure 4.7.: System architectures of all considered benchmarks.

The system architectures are depicted in Figure 4.7. For the specifications of the n-ary latch, the generalized buffer, the load balancer, and the shift, we refer to the benchmark descriptions of the synthesis competition [JBC+19]. The *ripple-carry adder* adds two bit vectors, both with n bits. The inputs are the very first carry bit c_{in} and the bits of the two bit vectors, $x_0, \dots, x_{n-1}$ and $y_0, \dots, y_{n-1}$. The outputs are the sum bits $s_0, \dots, s_{n-1}$ as well as the carry bits $c_0, \dots, c_{n-1}$. The specification φ is then given by $\varphi := \varphi_{init} \land \bigwedge_{0<i<n} \varphi_i$, where

$$\begin{aligned}
\varphi_{init} :=& \square (\bigcirc c_0 \leftrightarrow ((x_0 \land y_0) \lor (c_{in} \land ((x_0 \land \neg y_0) \lor (\neg x_0 \land y_0)))))\\
&\land \square (\bigcirc s_0 \leftrightarrow ((x_0 \land \neg y_0 \land \neg c_{in}) \lor (\neg x_0 \land y_0 \land \neg c_{in})\\
&\qquad\qquad \lor (\neg x_0 \land \neg y_0 \land c_{in}) \lor (x_0 \land y_0 \land c_{in})))\\
\varphi_i :=& \square (\bigcirc c_i \leftrightarrow ((x_i \land y_i) \lor (c_{i-1} \land ((x_i \land \neg y_i) \lor (\neg x_i \land y_i)))))\\
&\land \square (\bigcirc s_i \leftrightarrow ((x_i \land \neg y_i \land \neg c_{i-1}) \lor (\neg x_i \land y_i \land \neg c_{i-1})\\
&\qquad\qquad \lor (\neg x_i \land \neg y_i \land c_{i-1}) \lor (x_i \land y_i \land c_{i-1})))
\end{aligned}$$

The *robots* benchmarks describes the robots from the running example presented in Section 4.1. The specification has two parameters, n_1 and n_2. The additional objectives of the robots state

Table 4.1.: Results on scalable benchmarks. Reported is the parameter and the running time (in seconds) for certifying synthesis with deterministic guarantee transducers, distributed BoSy, and dominant strategy synthesis. The timeout is 60 minutes.

Benchmark	Parameter	Certifying Synthesis	Distributed BoSy	Dominant Strategies
n-ary Latch	2	**0.89**	41.26	4.75
	3	**0.91**	TO	6.40
	4	**0.92**	TO	8.46
	5	**0.94**	TO	10.74
	6	**12.26**	TO	13.89
	7	105.69	TO	**15.06**
Generalized Buffer	1	**1.20**	6.59	5.23
	2	**2.72**	3012.51	10.53
	3	**122.09**	TO	961.60
Load Balancer	1	**0.98**	1.89	2.18
	2	**1.64**	2.39	–
Shift	2	**1.10**	1.99	4.76
	3	**1.13**	4.16	7.04
	4	**1.14**	TO	11.13
	5	**1.29**	TO	13.68
	6	**2.20**	TO	16.01
	7	**9.01**	TO	16.08
	8	71.89	TO	**19.38**
Ripple-Carry Adder	1	**0.878**	1.83	–
	2	**2.09**	36.84	–
	3	**106.45**	TO	–

that robot r_i needs to visit the machine it is responsible for in every n_i-th step, indicated with the additional output variable m_i. The full specification φ of the benchmark is then given by $\varphi := \varphi_{no\_crash} \land \bigwedge_{1 \leq i \leq 2} \left(\varphi_{cross_i} \land \varphi_{add_i}\right)$, where

$$\varphi_{add_i} := m_i \land \square \left(m_i \rightarrow \left(\bigcirc \neg m_i \land \bigcirc^2 \neg m_i \land \ldots \bigcirc^{n_i - 1} \neg m_i \land \bigcirc^{n_i} m_i\right)\right)$$

and where $\bigcirc^x$ is syntactic sugar for applying the $\bigcirc$-operator x-times.

The results of the experiments are shown in Table 4.1. Since remorsefree dominance only makes implicit assumptions on the behavior of the other processes, dominance-based composition synthesis [DF14] does not always succeed. For instance, there are no independent remorsefree dominant strategies for the load balancer and the ripple-carry adder. While certifying synthesis performs better than the dominance-based compositional synthesis algorithm for the generalized buffer, the overhead of synthesizing explicit certificates becomes clear for the latch and the shift benchmarks: for larger parameters, synthesizing remorsefree dominant strategies outperforms certifying synthesis. However, the implicit assumptions do not encapsulate the required interface between the processes, and thus they do not increase the understandability of the system's interconnections.

For the latch, the generalized buffer, the ripple-carry adder, and the shift benchmark, certifying synthesis clearly outperforms the extension of BoSy to distributed systems [Bau17]. For these benchmarks, the latter does not terminate within 60 minutes for many parameters, while certifying synthesis solves the tasks in less than 13 seconds. Here, a process does not need to know the full behavior of the relevant processes. Thus, the certificates are notably smaller than the strategies. For instance, a process of the ripple-carry adder only needs information about the carry bit of the previous process; the sum bit is irrelevant. In contrast, the load balancer requires the certificates to contain the full behavior of the processes. Thus, the benefit of the compositional approach lies solely in the specification decomposition. This advantage suffices to produce a solution faster than distributed BoSy. However, for other benchmarks with full certificates, the overhead of synthesizing certificates dominates the benefit of specification decomposition for larger parameters, showcasing that certifying synthesis is particularly beneficial if a small interface between the processes exists.

Next, we consider the parameterized version of the running example from Section 4.1 describing two robots in a factory. It is parameterized in the size of the additional objectives φ_{add_i} of the robots r_1 and r_2. The robot benchmark is designed such that the interface stays small for all parameters. Thus, it demonstrates the advantage of abstracting away irrelevant behavior. We scale φ_{add_i}, while φ_{safe} and φ_{cross_i} are not changed: the parameter k_i denotes that r_i needs to visit a machine in every k_i-th step. The results in terms of strategy sizes in the number of states and running times in seconds are shown in Table 4.2.

Certifying synthesis clearly outperforms distributed BoSy on all instances. The size of the solutions of certifying synthesis only depends on the parameter of the respective robot and the size of the other robot's certificate. For all parameters, the certificates are of size two: the additional scalable requirements do not affect the other robot and thus do not constitute guarantee outputs. Hence, the certificate only needs to contain information about the robot's behavior at crossings, which can be encoded in a two-state transducer. The size of the solution with distributed BoSy, in contrast, depends on the parameters for *both* robots. Therefore, the solution sizes and, thus, the running times do not grow in parallel for certifying synthesis and distributed BoSy. The solution size, however, clearly has a great impact on the running time as both the constraint system and the search space grow significantly when increasing the solution size. Hence, this line of experiments demonstrates that certifying synthesis is highly beneficial for specifications where small certificates exist. This directly corresponds to the existence of a small interface between the processes of the system. Hence, bounding the size of the certificates indeed guides the synthesis procedure in finding solutions fast.

4.7.2. Deterministic vs. Nondeterministic Certificates

In a third line of experiments, we compared our implementations of certifying synthesis with deterministic guarantee transducers to our implementation of certifying synthesis with nondeterministic guarantee transducers. First, we synthesized solutions with nondeterministic guarantee transducers for the five benchmarks presented in Table 4.1, i.e., for the four SyntComp

Table 4.2.: Results for the running example. Reported are the parameters, strategy sizes, and running times (in seconds) for certifying synthesis and distributed BoSy. The timeout is 60 minutes.

Parameter	Strategy Size		Running Time	
	Cert. Synth.	Dist. BoSy	Cert. Synth.	Dist. BoSy
2, 3	2, 6	6	**1.59**	2.91
2, 4	2, 4	4	**1.18**	2.43
2, 5	2, 10	10	**3.97**	299.11
2, 6	2, 6	6	**1.40**	3.25
2, 7	2, 14	14	**76.32**	TO
2, 8	2, 8	8	**2.47**	5.28
2, 9	2, 18	18	**1832.53**	TO
2, 10	2, 10	10	**7.78**	106.34
3, 4	6, 4	12	**1.44**	TO
3, 5	6, 10	30	**32.83**	TO
3, 6	6, 6	6	**2.04**	3.43
3, 7	6, 14	42	**373.90**	TO
3, 8	6, 8	24	**8.82**	TO
3, 9	6, 18	18	TO	TO
3, 10	6, 10	30	**30.92**	TO
4, 5	4, 10	20	**11.66**	TO
4, 6	4, 6	12	**2.04**	TO
4, 7	4, 14	28	**221.17**	TO
4, 8	4, 8	8	**3.28**	6.06
4, 9	4, 18	36	**2911.26**	TO
4, 10	4, 10	20	**7,93**	TO
5, 6	10, 6	30	**26.16**	TO
5, 7	10, 14	35	TO	TO
5, 8	10, 8	40	**26.164**	TO
5, 9	10, 18	45	TO	TO
5, 10	10, 10	10	**89.87**	335.98

benchmarks n-ary latch, generalized buffer, load balancer, and shift as well as the ripple-carry adder. For none of these benchmarks, permitting nondeterminism has a size advantage on the guarantee transducers. That is, the minimal size of nondeterministic guarantee transducers that satisfy the requirements of certifying synthesis is also the minimal size of deterministic guarantee transducers. In contrast to deterministic guarantee transducers, the running times vary widely when synthesizing nondeterministic guarantee transducers. Most likely, permitting nondeterminism increases the degree of freedom, and thus the possibility for the underlying SAT solver to "take a wrong path" during solving, yielding the varying running times. Hence, we consider the average running time over ten runs. For smaller parameters, the running times of certifying synthesis with nondeterministic guarantee transducers are similar to those with deterministic ones. For larger parameters, the overhead increases. For the 7-ary latch, for

Table 4.3.: Comparison of certifying synthesis with deterministic and nondeterministic GTs. Reported is the parameter, the strategy sizes, and the running time (in seconds). The timeout is 60 minutes.

Parameter	Strategy Size		Running Time	
	deterministic	nondeterministic	deterministic	nondeterministic
3	6, 6	4, 3	1.30	**1.03**
4	4, 4	4, 2	1.17	**0.96**
5	10, 10	6, 3	31.67	**1.25**
6	6, 6	6, 2	4.35	**1.01**
7	14, 14	8, 3	TO	**1.23**
8	8, 8	8, 2	TO	**1.34**
9	18, 18	10, 3	TO	**1.91**
10	10, 10	10, 2	TO	**1.30**
11	22, 22	12, 3	TO	**3.44**
12	12, 12	12, 2	TO	**3.34**
13	26, 26	14, 3	TO	**13.88**
14	14, 14	14, 2	TO	**10.52**
15	30, 30	16, 3	TO	**30.48**
16	16, 16	16, 2	TO	**28.86**
17	34, 34	18, 3	TO	**398.56**
18	18, 18	18, 2	TO	**168.19**
19	38, 38	20, 3	TO	**299.80**
20	20, 20	20, 2	TO	**428.82**

instance, we have an overhead of 12% in the running time. For the generalized buffer with three senders, synthesizing nondeterministic guarantee transducers takes 11% more time than synthesizing deterministic ones. For the shift-benchmark with eight inputs, the overhead in the running time over the deterministic approach is 23%.

To analyze the advantage of permitting nondeterminism in guarantee transducers, we consider a benchmark with two processes, where, similar to Example 4.9 in Section 4.6, the nondeterministic guarantee transducers for process p_1 stays small, while the size of the deterministic guarantee transducers increases with the parameter. In particular, the benchmark is designed such that the guarantee transducer of process p_2 always consists of two states, irrespective of the parameter and whether we synthesize deterministic or nondeterministic certificates. The certificate sizes for p_1, however, differ for different parameters and different kinds of guarantee transducers. While a nondeterministic guarantee transducer with two states suffices for all parameters, the size of the deterministic guarantee transducer grows with the parameter. In fact, for parameter n, the deterministic guarantee transducer consists of n states. Due to the larger deterministic certificate, the strategy sizes for both processes p_1 and p_2 grow fast when considering deterministic guarantee transducers. The detailed results in terms of strategy size and running time are shown in Table 4.3.

Permitting nondeterminism has a clear benefit on the running time. With deterministic guarantee transducers, certifying synthesis does not terminate within one hour from parameter

$n = 7$ on, while we still synthesize a solution with nondeterministic guarantee transducers in less than eight minutes up to parameter $n = 20$. The fact that not only the certificate sizes but also the strategy sizes increase when considering deterministic guarantee transducers has a great impact on this significant difference. For benchmarks where only the certificate sizes differ, the running times do not differ as much. Often, however, large certificates yield an increase in the strategy size as well. Hence, the experiment demonstrates again that certifying synthesis is particularly beneficial when solutions with small certificates exist.

4.8. Summary

We have presented a sound and complete synthesis algorithm that reduces the complexity of distributed synthesis by decomposing the global system specification into local requirements on the individual processes. It synthesizes additional certificates that capture a certain behavior a system process commits to. The certificates then form an assume-guarantee contract, allowing a process to rely on the other processes to not deviate from the guaranteed behavior formalized in their certificate. The certificates increase the understandability of the system and the solution since the certificates capture which agreements the processes have to establish. Moreover, the certificates form a contract between the processes: the synthesized strategies can be exchanged safely as long as the new strategy still complies with the contract, i.e., as long as it does not deviate from the certificate, enabling modularity.

We have introduced two representations of the certificates, as LTL formulas and as deterministic finite-state transducers. For the latter, we presented an encoding of the search for strategies and certificates into a SAT constraint-solving problem. Furthermore, we extended the representation of certificates with finite-state transducers with nondeterminism, allowing for significantly smaller certificates than with deterministic transducers for certain specifications. We presented how the SAT constraint system needs to be changed to permit nondeterminism. Moreover, we have introduced a technique for reducing the number of certificates that a process needs to consider by determining relevant processes. We have implemented the certifying synthesis algorithm based on the two SAT encodings for certificates represented by deterministic and nondeterministic finite-state transducers. We compared it to two extensions of the synthesis tool BoSy to distributed systems. Furthermore, we analyzed the advantage of permitting nondeterminism in the certificates represented by finite-state transducers. The results clearly show the advantage of compositional approaches as well as of guiding the synthesis procedure by bounding the size of the certificates. For benchmarks where small interfaces between the processes exist, certifying synthesis outperforms the other algorithms significantly. If no solution with small interfaces exists, the overhead of certifying synthesis is small. Permitting nondeterminism can reduce the strategy and certificate sizes significantly.

Part II.

Monolithic Systems

Chapter 5

System Decomposition for Assumption-free Winning Strategies

In this chapter, we transfer classical compositional concepts to reactive synthesis of monolithic systems. We present a *modular synthesis algorithm* for monolithic systems that, given a specification and a decomposition of the single system process into several subspecifications, performs separate synthesis tasks for the components induced by the subspecifications. Afterward, the synthesized strategies are combined into a single strategy.

The main challenge in such compositional monolithic synthesis approaches is to decompose the single system process into several components, for which synthesis subtasks can be performed. For distributed systems, the system processes naturally serve as these components. For monolithic systems, in contrast, suitable decomposition techniques that preserve realizability and unrealizability of the initial monolithic synthesis task are necessary. Consequently, a crucial part of modular synthesis is to determine which parts of the specification depend on each other in the sense that considering them individually violates either soundness or completeness of modular synthesis. Once the dependencies have been identified, independent components for sound and complete modular synthesis can be constructed from them.

We present a criterion, the so-called *independent sublanguages criterion*, for subspecifications that ensures soundness and completeness of modular synthesis. It determines for a given set of subspecifications whether or not they are dependent, i.e., whether or not separate synthesis tasks for them will preserve realizability or unrealizability of the initial specification. More precisely, it determines whether or not it holds that (i) whenever the initial specification is realizable, the individual synthesis tasks for the subspecifications will succeed, and (ii) whenever the initial specification is unrealizable, at least one of the individual synthesis tasks will fail. The criterion is purely language-based, i.e., it is defined on the language of the specification. Hence, it is agnostic of the specification formalism used for synthesis.

Lifting the independent sublanguages criterion to the temporal logic level, we introduce an approximate independence criterion for specifications given as LTL formulas. It allows for determining whether or not subspecifications are independent on the LTL level, i.e., without considering the respective languages. It is approximate in the sense that it may conclude that two subspecifications are dependent although independent solutions can be synthesized.

Thus, utilizing the LTL independence criterion might result in coarser decompositions than the independent sublanguages criterion. Nevertheless, the criterion is sound. That is, whenever two subspecifications indeed depend on each other, the LTL dependence criterion will conclude dependence. Vaguely inspired by work on more scalable LTL model checking [DR18], the LTL independence criterion is based on a syntactic analysis of the LTL formula. Intuitively, it labels variables of the system as dependent if they occur in the same conjunct of the specification: the valuations of the variables occurring in some conjunct φ_i of an LTL specification $\varphi = \varphi_1 \wedge \ldots \wedge \varphi_m$ may influence the satisfaction of φ_i and thus also the satisfaction of φ. The criterion then concludes the dependence of two subspecifications if they contain dependent variables.

Since the LTL independence criterion heavily relies on analyzing the conjuncts of the specification of the entire system, it often unnecessarily concludes dependence of processes for specifications given in the common assume-guarantee form $\varphi = \bigwedge_{i=1}^{\ell} \varphi_i \rightarrow \bigwedge_{j=1}^{m} \psi_j$, where the formulas φ_i are assumptions and the formulas ψ_j are guarantees. When rewriting such a formula into conjunctive form, every guarantee is equipped with all guarantees, yielding dependence between all variables occurring in some assumption and the considered guarantee. However, not all assumptions are necessary for the realizability of all conjuncts. Hence, dependencies between conjuncts of the specification can sometimes be prevented.

Therefore, we present an optimization of the LTL independence criterion that utilizes so-called *assumption dropping*. Before checking the dependence of variables by analyzing the conjuncts of the specification, the optimized LTL independence criterion analyzes, for every conjunct of the guarantees, the assumptions of the specification and drops those that do not influence the satisfaction of the considered conjunct. We introduce criteria for dropping assumptions for LTL specifications both in *strict* assume-guarantee form and in *non-strict* assume-guarantee form, i.e., specifications that consist of several assume-guarantee conjuncts.

Modular synthesis can then immediately utilize the LTL independence criterion – and its variants for assume-guarantee specification – for applying classical compositional approaches to reactive synthesis. It determines independent subspecifications of the initial specifications using the independence criterion. Then, the algorithm synthesizes strategies for the subspecifications separately and composes the results. The soundness of the criterion ensures that both the synthesis subtasks and the composition succeed as long as the system specification is realizable. Otherwise, modular synthesis produces a counterstrategy.

In an experimental evaluation, we utilize the LTL decomposition algorithm as a preprocessing technique for modular synthesis with the two monolithic synthesis tools BoSy [FFT17] and Strix [MSL18]. We evaluate our approach on the publicly available benchmarks of the annual synthesis competition SyntComp [BEJ14]. For all benchmarks, the decomposition algorithm terminates in less than 26 milliseconds. Thus, even for non-decomposable specifications the overhead of decomposition is negligible. Both BoSy and Strix increase the number of successfully synthesized benchmarks when running modularly and decrease their running time significantly for many benchmarks for which multiple components have been identified, showcasing the advantage of compositional techniques over classical ones. The developers of the synthesis tool ltlsynt [MC18] recognized the potential of our modular synthesis approach based on LTL decomposition to be a game changer for reactive monolithic synthesis and integrated it into their newest release [RSDP22], which successfully competed in SyntComp 2022 [SYN22].

Furthermore, we demonstrate that smart contract specifications can be decomposed into independent parts describing the contract's control flow and the effects of function calls. This results gave rise to the development of an efficient synthesis tool that fully automatically constructs Solidity code from temporal control flow specifications [FHKP23].

Publications and Structure. This chapter is based on work published in the proceedings of the *13th NASA Formal Methods Symposium* [FGP21a], and in the *Innovations in Systems and Software Engineering Journal* [FGP22] as well as the extended version [FGP21b] of the former publication. First ideas for modular synthesis and the independence criteria have been published in Gideon Geier's Bachelor's thesis [Gei20], which was advised by the author of this thesis. The concrete formulation of the criterion and, thus, all results and proofs have been adapted by the author of this thesis for the conference and journal publications. Moreover, the author of this thesis converted the strategy formalism to finite-state transducers for consistency in this thesis. Section 5.7 is additionally based on work published in the following preprint [FHKP23].

This chapter is structured as follows. First, we introduce the modular synthesis algorithm and show its soundness and completeness for proper decompositions. In Section 5.2, we formalize the suitability of a decomposition by introducing the independent sublanguages criterion. We prove soundness and completeness of modular synthesis for all decompositions that adhere to the criterion. Afterward, in Section 5.3, we lift the language-based criterion to the temporal logic level by introducing the approximate LTL independence criterion. We show soundness and completeness of modular synthesis for all decompositions that respect the LTL independence criterion. Moreover, we present a decomposition algorithm for LTL specifications that computes decompositions that adhere to the criterion. Addressing the impreciseness of LTL decomposition for formulas in assume-guarantee form, we introduce criteria for eliminating assumptions for individual guarantees while preserving realizability and unrealizability in Sections 5.4 and 5.5. In the former section, we focus on formulas in a strict assume-guarantee form. Afterward, we extend it to a non-strict form in the latter section, thus allowing for LTL formulas that consist of several assume-guarantee conjuncts. For both variants, we prove soundness and completeness of modular synthesis. Furthermore, we present decomposition algorithms for LTL specifications in the respective forms that identify droppable assumptions and decompose the specification according to these results. In Section 5.6, we present an experimental evaluation of the performance of our algorithms. Lastly, in Section 5.7, we demonstrate the applicability of our decomposition algorithms to the domain of smart contracts.

5.1. Modular Monolithic Synthesis

In this section, we introduce a modular synthesis algorithm for monolithic systems that, given a suitable decomposition algorithm, decomposes the single system process into independent components and performs individual synthesis tasks for them. It then recomposes the obtained solutions to either a strategy or a counterstrategy for the entire system. The algorithm thus transfers classical concepts of compositionality from distributed systems, such as, for instance, described in Part I of this thesis, to monolithic systems.

Algorithm 5.1: Modular Synthesis for Monolithic Systems

```
   Input: spec: Specification, I: List Variable, O: List Variable
   Output: realizable: Bool, s: Strategy
1  components ← decompose(spec, I, O)
2  subStrategies ← []: List Strategy
3  foreach (compSpec, cInp, cOut) ∈ components do
4      (cRealizable, cStrategy) ← synthesize(compSpec, cInp, cOut)
5      if cRealizable then
6          subStrategies.append(cStrategy)
7      else
8          counterstrategy ← extendCounterstrategy(cStrategy, I, O)
9          return (false, counterstrategy)
10 strategy ← compose(subStrategies)
11 return (true, strategy)
```

Algorithm 5.1 describes the modular synthesis approach for monolithic systems. It expects sets of input and output variables of the system as well as the system specification as input. First, the single system process is decomposed using an adequate decomposition algorithm (line 1). The result is a list of *components*. A component $c = (L_c, I_c, O_c)$ of the system consists of a *component specification* L_c of the initial specification and a *component interface*, represented by sets I_c and O_c of input and output variables. Intuitively, the component interface captures the inputs and outputs of the system that occur in the component's specification L_c. Thus, we have $L_c \subseteq (2^{I_c \cup O_c})^\omega$, $I_c \subseteq I$, and $O_c \subseteq O$. A system decomposition is then defined as follows:

Definition 5.1 (System Decomposition)**.**
Let I and O be sets of input and output variables with $I \cap O = \emptyset$ and let $V = I \cup O$. A *decomposition* $\mathbb{D}$ of (I, O) is a vector $\mathbb{D} = \langle c_1, \dots, c_n \rangle$ of n components with $c_i = (L_i, I_i, O_i)$ and $V_i = I_i \cup O_i$ such that (i) $I_i \cap O_i = \emptyset$ holds for all $c_i \in \mathbb{D}$ and (ii) $V = \bigcup_{1 \leq i \leq n} V_i$.

In contrast to the setting for distributed systems discussed in Part I, we require every component to only observe input variables of the whole system. A component is thus not able to react to the output variables of another component. Therefore, we do not restrict strategies to be representable with Moore transducers in this chapter, as determinism and completeness of the parallel composition of component strategies can also be ensured for Mealy transducers as long as certain restrictions, which we discuss in detail in Section 5.2, are satisfied.

Deriving independent components of the single process of a monolithic system and thus finding an adequate decomposition algorithm is *the* crucial aspect of modular synthesis. This section, however, focuses on presenting the modular synthesis algorithm itself. Therefore, we assume that *some* correct decomposition algorithm is used. In the remaining sections of this chapter, we then present how independent components can be computed.

Once the system is decomposed, modular synthesis performs individual synthesis tasks for all of the system's components and stores the results (lines 3 to 4 of Algorithm 5.1). A synthesis result consists of two parts: a Boolean variable depicting whether the component specification is realizable for the component and a strategy. If the component specification is realizable (line 5), then, by construction, the strategy realizes the component specification, and thus it can be used as a part of a strategy for the entire system. Hence, we store the strategy (line 6). If, for some component, the component specification is unrealizable (line 7), however, the strategy returned by the synthesis procedure is a *counterstrategy*. Intuitively, a counterstrategy is a strategy of the components' environment that prevents the satisfaction of the component specification no matter how the system, in this case, the component, behaves. Formally, we represent a counterstrategy as a transducer, the so-called *counterstrategy transducer*:

Definition 5.2 (Counterstrategy Transducer)**.**
Let I and O be finite sets of input and output variables with $I \cap O = \emptyset$ and let $V = I \cup O$. Let $L \subseteq (2^V)^\omega$ be a language. If L is unrealizable, then there is a *counterstrategy transducer* for L. A counterstrategy transducer is a deterministic and complete finite-state $(2^O, 2^I)$-transducer $\mathcal{T}^c$ such that $Traces(\mathcal{T}^c) \subseteq (2^V)^\omega \setminus L$ holds. If L is unrealizable for Mealy transducers, then there exists a Moore counterstrategy transducer for L. If L is unrealizable for Moore transducers, then there exists a Mealy counterstrategy transducer for L.

Modular synthesis checks whether some of the synthesis subtasks are unrealizable. If so, it extends the corresponding counterstrategy to a counterstrategy for the whole system and returns it (lines 8 to 9 of Algorithm 5.1). Intuitively, a component's counterstrategy violates the full system specification as well since it violates the parts of it that affect the component whose component specification was unrealizable. A counterstrategy for the whole system, however, needs to be defined on all variables of the system, not only the ones occurring in the component. Since the system variables outside of the violating component do not affect the violation of the component specification, by definition of components, the component counterstrategy can be extended to counterstrategy for the whole system by, intuitively, ignoring output variables outside the component and assigning an arbitrary valuation to input variables outside the component. Formally, we first define the extension of transducers in general and then tailor it to counterstrategy extension afterward:

Definition 5.3 (Transducer Extension)**.**
Let I, I_1 and O, O_1 be finite sets of input and output variables with $I \cap O = \emptyset$, $I_1 \subseteq I$, and $O_1 \subseteq O$. Let $\mathcal{T}_1 = (T_1, T_{1,0}, \tau_1, \ell_1)$ be a finite-state $(2^{I_1}, 2^{O_1})$-transducer representing a strategy. We construct a $(2^I, 2^O)$-transducer $\mathcal{T} = (T, T_0, \tau, \ell)$ from $\mathcal{T}_1$ as follows:

- $T = T_1$,
- $T_0 = T_{1,0}$, and
- $(t, \iota, t') \in \tau$ if, and only if, $(t, \iota \cap I_1, t') \in \tau_1$ holds, and
- $(t, \iota, o) \in \ell$ if, and only if $(t, \iota, o \cap O_1) \in \ell_1$ and $o \cap (O \setminus O_1) = pick(2^{O \setminus O_1})$.

where $pick(M)$ picks one element from set M.

Suppose that some deterministic and complete $(2^{I_1}, 2^{O_1})$-transducer $\mathcal{T}_1$ realizes some language $L_1 \subseteq (2^{I_1 \cup O_1})^\omega$. Then, it follows that the extended transducer $\mathcal{T}$ constructed according to Definition 5.3 realizes a language $L \subseteq (2^{I \cup O})^\omega$ as well as long as L is a part of L_1, i.e., as long as every word that lies in L_1 also lies in L:

Lemma 5.1. *Let I, I_1 and O, O_1 be finite sets of input and output variables with $I \cap O = \emptyset$, $I_1 \subseteq I$, and $O_1 \subseteq O$. Let $V = I \cup O$ and $V_1 = I_1 \cup O_1$. Let $L \subseteq (2^V)^\omega$ be a language. Let $L_1 \subseteq (2^{V_1})^\omega$ with $L_1 \subseteq \{\sigma \cap V_1 \mid \sigma \in L\}$. Let L_1 be realizable and let $\mathcal{T}_1$ be a transducer realizing L_1. The extended transducer $\mathcal{T}$ constructed as in Definition 5.3 represents a strategy that realizes L.*

Proof. By construction, $\mathcal{T}$ is a $(2^I, 2^O)$-transducer. Determinism and completeness of $\mathcal{T}$, as well as finiteness of $\mathcal{T}$'s set of states, follows immediately from the construction of $\mathcal{T}$ and the fact that $\mathcal{T}_1$ is a deterministic and complete finite-state transducer. Note that it is crucial for determinism of $\mathcal{T}$ to utilize the function *pick*, which chooses a single valuation of the variables outside of V_1. Furthermore, by the definition of transducer extension, $\mathcal{T}$ neither introduces additional transitions with respect to $\mathcal{T}_1$ nor changes the valuation of the variables in V_1 defined by the labeling relation of $\mathcal{T}_1$. Therefore, in particular, the semantics does not change when extending a transducer, i.e., if $\mathcal{T}_1$ has Mealy semantics, then so does $\mathcal{T}$ and if $\mathcal{T}_1$ has Moore semantics, then so does $\mathcal{T}$. Thus, it remains to show that $\mathit{Traces}(\mathcal{T}) \subseteq L$ holds.

Let $\sigma \in \mathit{Traces}(\mathcal{T})$ be a trace of $\mathcal{T}$. Then, by construction of $\mathcal{T}$ and by the definition of traces, there exists some trace $\sigma' \in \mathit{Traces}(\mathcal{T}_1)$ of $\mathcal{T}_1$ such that for all points in time $k \geq 0$, we have $\sigma_k \cap I_1 = \sigma'_k \cap I_1$ and $\sigma_k \cap O_1 = \sigma'_k \cap O_1$. Therefore, $\sigma_k \cap V_1 = \sigma'_k \cap V_1$ follows for all $k \geq 0$ with the definition of V_1. Since $\mathcal{T}_1$ is a $(2^{I_1}, 2^{O_1})$-transducer, we have $\sigma' \in (2^{V_1})^\omega$and thus $\sigma \cap V_1 = \sigma'$ follows. By assumption, L_1 is realizable and $\mathcal{T}_1$ realizes L_1. Therefore, $\mathit{Traces}(\mathcal{T}_1) \subseteq L_1$ holds. Since $\sigma' \in \mathit{Traces}(\mathcal{T}_1)$ holds, σ' thus realizes L_1, i.e., we have $\sigma' \in L_1$. Hence, since $\sigma \cap V_1 = \sigma'$ holds, $\sigma \cap V_1 \in L_1$ follows. By assumption, we have $L_1 \subseteq \{\sigma \cap V_1 \mid \sigma \in L\}$. Hence, $\sigma \in L$ follows. Since we chose $\sigma \in \mathit{Traces}(\mathcal{T})$ arbitrarily, $\sigma \in L$ follows for all $\sigma \in \mathit{Traces}(\mathcal{T})$. Hence, $\mathit{Traces}(\mathcal{T}) \subseteq L$ holds and therefore $\mathcal{T}$ realizes L. □

Based on transducer extension and the above realization result, we can now formalize the extension of a counterstrategy for a component to a counterstrategy for the whole system. The *counterstrategy extension* of a counterstrategy $(2^{O_1}, 2^{I_1})$-transducer $\mathcal{T}_1^c$ for a language $L_1 \subseteq (2^{I_1 \cup O_1})^\omega$ to sets $I \supseteq I_1$ and $O \supseteq O_1$ of input and output variables, respectively, is the $(2^O, 2^I)$-transducer obtained by extending $\mathcal{T}_1^c$ with transducer extension, i.e., according to Definition 5.3, to the sets I and O. Since $\mathcal{T}_1^c$ is a counterstrategy transducer for L_1, all of its traces violate L_1, i.e., we have $\mathit{Traces}(\mathcal{T}_1^c) \subseteq (2^{I_1 \cup O_1})^\omega \setminus L_1$. Hence, it follows with Lemma 5.1 that the counterstrategy extension $\mathcal{T}$ is a counterstrategy transducer for a language $L \in (2^{I \cup O})^\omega$ if $((2^{I_1 \cup O_1})^\omega \setminus L_1) \subseteq \left\{\sigma \cap V_1 \mid \sigma \in ((2^{I \cup O})^\omega \setminus L)\right\}$ holds. Consequently, we obtain that $\mathcal{T}$ is a counterstrategy transducer for L if $\{\sigma \cap (I_1 \cup O_1) \mid \sigma \in L\} \subseteq L_1$ holds.

Corollary 5.1. *Let I, I_1 and O, O_1 be finite sets of input and output variables with $I \cap O = \emptyset$, $I_1 \subseteq I$, and $O_1 \subseteq O$. Let $V = I \cup O$ and $V_1 = I_1 \cup O_1$. Let $L \subseteq (2^V)^\omega$ be a language. Let $L_1 \subseteq (2^{V_1})^\omega$ with $\{\sigma \cap V_1 \mid \sigma \in L\} \subseteq L_1$. Let L_1 be unrealizable and let $\mathcal{T}_1^c$ be a respective counterstrategy transducer. The counterstrategy extension $\mathcal{T}^c$ of $\mathcal{T}_1^c$ is a counterstrategy transducer for L.*

Hence, if one of the synthesis tasks of modular synthesis fails due to unrealizability of the component specification, then we construct a counterstrategy transducer for the entire system with counterstrategy extension. As long as the decomposition algorithm ensures that the component specifications are parts of the initial system specification in the sense that every word that lies in the component specification also lies in the initial specification, Corollary 5.1 allows for concluding that the resulting strategy is indeed a counterstrategy for the full system and the initial specification.

Otherwise, i.e., if all individual synthesis tasks in modular synthesis succeed, then derived component strategies all have been stored in the list `subStrategies`. We then compose the stored component strategies according to Definition 2.12, i.e., by building the parallel composition of the transducers representing the component strategies (lines 10 and 11 of Algorithm 5.1). By construction, the resulting finite-state transducer represents a strategy for the entire system. The decomposition algorithm is then required to ensure that the parallel composition of the component strategies indeed satisfies the full system specification.

The soundness and completeness of modular synthesis clearly depend on the employed decomposition algorithm. In the following, we describe how soundness and completeness of modular synthesis can be violated due to an incorrect decomposition algorithm. First, realizability and unrealizability of the initial synthesis task might not be preserved. Hence, a component specification might be unrealizable in the respective component, although the initial specification is realizable for the whole system. Vice versa, all component specifications might be realizable in their respective components, although the initial specifications is unrealizable. Second, even if realizability and unrealizability are preserved, the computed strategy or counterstrategy might not be suitable for the entire system. More precisely, the counterstrategy extension of a component's counterstrategy might not be a counterstrategy for the initial specification, or the parallel composition of the component strategies might violate the full specification. Furthermore, already the composition of the component strategies might fail if the synthesized component strategies can be contradictory.

In the remainder of this chapter, we thus focus on developing suitable decomposition algorithms that ensure soundness and completeness of modular monolithic synthesis and thus avoid the pitfalls outlined above. First, we derive a criterion for the dependence of component specifications such that every decomposition that adheres to the criterion can be safely used for modular synthesis. Afterward, we introduce an algorithm that, based on the independence criterion, computes suitable decompositions from system specifications.

5.2. Language-based Independence Criterion

As a first step toward an algorithm for decomposing a monolithic system into several components that ensures soundness and completeness of modular monolithic synthesis, we present a language-based sound and complete criterion for determining whether or not *subspecifications* of the initial specification depend on each other in this section. The subspecifications then induce independent components of the single system process. With the goal of compositional synthesis in mind, we intuitively consider subspecifications to be dependent if either synthesizing strategies for the respective components separately does not preserve realizability or unrealizability of the original monolithic synthesis task or if deriving a strategy or counterstrategy, respectively, from the results of the synthesis subtasks fails.

Whether or not strategies for components can be synthesized separately depends on three requirements: (i) *non-contradictory composability* of the subresults, i.e., of the strategies for the individual components, (ii) *realization* of the full system specification by the parallel composition of the subresults as well as *violation* of the full system specification by the extension of a counterstrategy, and (iii) *equirealizability* of the initial specification and the component specifications. Clearly, requirement (i) is a necessary condition for requirement (ii). Note that these three conditions match the possible pitfalls of modular synthesis outlined in the previous section: if the three conditions are satisfied, all of the pitfalls are eradicated.

In this section, we formalize these three requirements for soundness and completeness of modular synthesis and study in which cases they are satisfied. We present a language-based criterion that, if satisfied, ensures that all three requirements are satisfied for a given specification and decomposition. Furthermore, we show that for specifications and decompositions satisfying the criterion, the synthesis task for the initial specification can indeed be split into separate synthesis tasks for the components.

We formalize the three requirements as follows. Let L be a language defining the specification of the overall system, and let $\mathbb{D}$ be a decomposition consisting of k components. Let $\mathcal{T}_1, \dots, \mathcal{T}_k$ be deterministic and complete finite-state transducers representing the subresults of modular synthesis. Note that these transducers can be both strategy transducers and counterstrategy transducers, depending on the realizability of the respective synthesis subtask. Equirealizability is given, i.e., requirement (iii) is satisfied, if L is realizable for the whole system, if, and only, if all component specifications are realizable for their respective components and consequently if, and only if, all transducers $\mathcal{T}_1, \dots, \mathcal{T}_k$ are strategy transducers. If all synthesis subtasks succeeded, i.e., if all transducers represent component strategies, then they are composable, i.e., they satisfy requirement (i), if their parallel composition $\mathcal{T}_1 \mid\mid \dots \mid\mid \mathcal{T}_k$ represents a strategy for the full system. As defined in Section 2.6.1, their parallel composition thus needs to have a finite number of states and needs to be both deterministic and complete. Lastly, the subresults form a solution of the initial synthesis task, i.e., they satisfy requirement (ii), if $\mathcal{T}_1 \mid\mid \dots \mid\mid \mathcal{T}_k$ realizes L if all transducers represent component strategies, and, if one of the transducers represents a counterstrategy, its extension to the entire system is a counterstrategy for L.

Note that, if all of the transducers $\mathcal{T}_1, \dots, \mathcal{T}_k$ represent component strategies, finiteness of the set of states of the parallel composition $\mathcal{T}_1 \mid\mid \dots \mid\mid \mathcal{T}_k$ follows immediately from the definition of the parallel composition of transducers (see Definition 2.12) and the fact that the transducers $\mathcal{T}_1, \dots, \mathcal{T}_k$ all have a finite number of states. Furthermore, determinism of $\mathcal{T}_1 \mid\mid \dots \mid\mid \mathcal{T}_k$ is ensured by the construction of components. Intuitively, the parallel composition of two deterministic transducers $\mathcal{T}_1$ and $\mathcal{T}_2$ can only be nondeterministic if some transition of $\mathcal{T}_1$ depends on the output of $\mathcal{T}_2$ or vice versa. By construction, however, the inputs of a component are the inputs of the overall system that occur in the component specification, and the component outputs are the outputs of the overall system occurring in the component specification. Hence, the sets of inputs of $\mathcal{T}_i$ and outputs of $\mathcal{T}_{3-i}$ are disjoint for $i \in \{1, 2\}$ and therefore a transition of $\mathcal{T}_i$ cannot depend on the output of $\mathcal{T}_{3-i}$. Formally, we show determinism of $\mathcal{T}_1 \mid\mid \mathcal{T}_2$ as follows:

Lemma 5.2. *Let I and O be finite sets of input and output variables with $I \cap O = \emptyset$ and let $V = I \cup O$. Let V_1 and V_2 be finite sets such that $V_1, V_2 \subseteq V$ holds. Let $I_1 = I \cap V_1$, $I_2 = I \cap V_2$, $O_1 = O \cap V_1$, and $O_2 = O \cap O_2$. Let $\mathcal{T}_1$ be a deterministic finite-state $(2^{I_1}, 2^{O_1})$-transducer and let $\mathcal{T}_2$ be a deterministic finite-state $(2^{I_2}, 2^{O_2})$-transducer. Then, $\mathcal{T}_1 \mid\mid \mathcal{T}_2$ is deterministic.*

Proof. Let $\mathcal{T}_1 = (T_1, T_{1,0}, \tau_1, \ell_1)$, $\mathcal{T}_2 = (T_2, T_{2,0}, \tau_2, \ell_2)$, and $\mathcal{T}_1 \mid\mid \mathcal{T}_2 = (T, T_0, \tau, \ell)$. Since both $\mathcal{T}_1$ and $\mathcal{T}_2$ are deterministic by assumption, we have $|T_{1,0}| \leq 1$ and $|T_{2,0}| \leq 1$ and thus, by definition of transducer composition, $|T_0| \leq 1$ holds. Let $(u, v), (u', v') \in T$, let $\iota \in 2^I$, and let $o \in 2^O$ such that $((u, v), \iota, (u', v')) \in \tau$ and $((u, v), \iota, o) \in \ell$ hold. Then, by construction of τ, there exist $o_1 \in 2^{O_1}$ and $o_2 \subseteq 2^{O_2}$ with $(u, \iota_1, o_1) \in \ell_1$ and $(v, \iota_2, o_2) \in \ell_1$ such that $(u, \iota_1, u') \in \tau_1$ and $(v, \iota_2, v') \in \tau_2$ hold, where $\iota_1 := (\iota \cup o_2) \cap I_1$ and $\iota_2 = (\iota \cup o_1) \cap I_2$. Furthermore, we have $(u, \iota_1, o \cap O_1) \in \ell_1$ and $(v, \iota_2, o \cap O_2) \in \ell_1$ as well. Since $\mathcal{T}_1$ and $\mathcal{T}_2$ are deterministic by assumption, u' and v' are the only successor states of u and v in $\mathcal{T}_1$ and $\mathcal{T}_2$ for input ι_1 and ι_2, respectively. Moreover, o_1 and o_2 are the only outputs of $\mathcal{T}_1$ and $\mathcal{T}_2$ in u and v for input ι_1 and ι_2, respectively. Thus, in particular, $o \cap O_1 = o_1$ and $o \cap O_2 = o_2$ hold. By construction of I_1, I_2, O_1, and O_2, we have $I_i \subseteq I$ and $O_i \subseteq O$ for $i \in \{1, 2\}$. Thus, since $I \cap O = \emptyset$ holds, it follows that $I_i \cap O_j = \emptyset$ holds for $i, j \in \{1, 2\}$. In particular, $(\iota \cup o_{3-i}) \cap I_i = \iota \cap I_i$ holds for $i \in \{1, 2\}$. Hence, u' and v' are the only successors of u and v in $\mathcal{T}_1$ and $\mathcal{T}_2$ for $\iota \cap I_1$ and $\iota \cap I_2$, respectively, and therefore (u', v') is the only successor state of (u, v) in $\mathcal{T}_1 \mid\mid \mathcal{T}_2$ for input ι. Furthermore, o_1 and o_2 are the only outputs of $\mathcal{T}_1$ and $\mathcal{T}_2$ in u and v, for $\iota \cap I_1$ and $\iota \cap I_2$, respectively, and therefore, since $o \cap O_1 = o_1$ and $o \cap O_2 = o_2$ hold, o is the only output of (u, v) in $\mathcal{T}_1 \mid\mid \mathcal{T}_2$ for input ι. Thus, $\mathcal{T}$ is deterministic. □

Therefore, a criterion that, if satisfied, ensures that all three requirements of modular synthesis are met only needs to ensure that (i) if all subresults represent component strategies, then the parallel composition of the subresults is *complete*, (ii) if all component strategies represent component strategies, then their parallel compositions *realizes* the initial specification and, otherwise, the extension of the counterstrategy is a *counterstrategy* for the entire system and the initial specification, and (iii) the initial specification is *equirealizable* to the component specifications. In the remainder of this section, we define two conditions on the component specifications – or, more precisely, on the components consisting of a specification and an interface – which, if satisfied, ensure that the three requirements are satisfied. These conditions, *non-contradictory languages* and *independent sublanguages*, then allow for defining an independence criterion that can be utilized for defining a decomposition algorithm that ensures soundness and completeness of modular synthesis.

5.2.1. Non-contradictory Languages

We first consider the requirement of non-contradictory composability of the component strategies obtained from the synthesis subtasks in modular synthesis. Recall that composability requires the parallel composition of the transducers representing the substrategies to have a finite number of states and to be both deterministic and complete. As shown in the previous section, finiteness of the number of states as well as determinism follows from the definition of the parallel composition of transducers as well as the definition of component interfaces. Thus, it remains to consider the *completeness* of the parallel composition of the transducers.

Intuitively, the parallel composition of two deterministic and complete transducers can only be incomplete if, for some input sequence, they produce output sequences that are contradictory in the sense that they do not agree on shared output variables. Since the transducers represent the individual component strategies of modular synthesis, they realize the respective component specifications, and hence all traces produced by them satisfy the component specifications. Therefore, we can formulate conditions on the traces of the transducers representing the subresults as conditions on the *languages of the component specifications.* To relate the component specification languages, we define the *composition of languages.* Intuitively, the composition of two languages L and L' combines words contained in L and L' that agree on shared variables. Formally, we define language composition as follows:

Definition 5.4 (Language Composition)**.**
Let V_1 and V_2 be finite alphabets. Let $L_1 \subseteq (2^{V_1})^\omega$ and $L_2 \subseteq (2^{V_2})^\omega$ be languages. Their *parallel composition* $L_1 \mid\mid L_2 \subseteq (2^{V_1 \cup V_2})^\omega$ is defined by

$$L_1 \mid\mid L_2 := \{\sigma \cup \sigma' \mid \sigma \in L_1 \land \sigma' \in L_2 \land \sigma \cap V_1 = \sigma' \cap V_1\}\,.$$

Note that two infinite words $\sigma \in L_1$ and $\sigma' \in L_2$ that do not agree on shared variables are not contained in the parallel composition $L_1 \mid\mid L_2$ of two languages L_1 and L_2. Then, σ and σ' define contradicting valuations of some shared variable at some point in time. Thus, in order to combine σ and σ', one would need to choose one of the valuations of this variable to obtain a well-defined infinite word. However, then the parallel composition $L_1 \mid\mid L_2$ may contain a word that, restricted to the variables of one of the languages, does not lie in this language. More precisely, if we choose the valuation defined by σ for the composition of σ and σ', then $L_1 \mid\mid L_2$ may contain a word $\sigma'' \in (2^{V_1 \cup V_2})^\omega$, namely the composition of σ and σ', such that $\sigma'' \cap V_2 \notin L_2$ holds. To avoid this, we, therefore, pose the restriction that combined words need to agree on shared variables. Indeed, every word in $L_1 \mid\mid L_2$ is then, restricted to the respective variables of the language, contained in both L_1 and L_2:

Proposition 5.1. *Let V_1 and V_2 be finite alphabets. Let $L_1 \subseteq (2^{V_1})^\omega$ and $L_2 \subseteq (2^{V_2})^\omega$ be languages. Then, for every $\sigma \in L_1 \mid\mid L_2$, we have $(\sigma \cap V_1) \in L_1$ and $(\sigma \cap V_2) \in L_2$.*

Language composition is a first step toward formulating a language-based condition that ensures the non-contradictory composability of subresults of modular synthesis. By requiring that, for every input sequence $\gamma \in (2^I)^\omega$ of the overall system, there exists a word $\sigma \in L_1 \mid\mid L_2$ in the parallel composition of L_1 and L_2 that agrees with γ on the input variables of the system, we ensure that for every input sequence, there exist matching words in L_1 and L_2 which agree on shared variables and are thus non-contradictory.

However, while this requirement already ensures non-contradictory composability of words produced by *some* strategies realizing L_1 and L_2 for every input sequence of the entire system, it does not ensure non-contradictory composability of all strategies realizing L_1 and L_2 yet. The languages L_1 and L_2 might allow for several output sequences for an input sequence, i.e., given an input sequence $\gamma \in (2^I)^\omega$, there might exist two words $\sigma_1, \sigma_1' \in L_1$ and two words $\sigma_2, \sigma_2' \in L_2$ which all agree with γ on input variables. As long as one combination of these output sequences is non-contradictory in the sense that the words agree on shared variables, then this combination is contained in $L_1 \mid\mid L_2$. For instance, $\sigma_1 \cup \sigma_2 \in L_1 \mid\mid L_2$ but $\sigma_1' \cup \sigma_2' \notin L_1 \mid\mid L_2$ might hold. Since one of the combinations is contained in $L_1 \mid\mid L_2$, the intuitive requirement presented above is satisfied for input sequence γ. Nevertheless, not all strategies realizing L_1 and L_2 are composable since the strategies producing the output sequences whose combination does not lie in $L_1 \mid\mid L_2$, i.e., the strategies producing σ_1' and σ_2', respectively, for input sequence γ, do not produce composable output sequences for γ. Therefore, we need to strengthen the above requirement to also account for the case where the sublanguages allow for several output sequences for a given input sequence. Intuitively, we can formulate non-contradictory composability of the subresults in terms of language composition by requiring that all words in the sublanguages L_1 and L_2 that agree on shared input variables need to constitute a word in the parallel composition $L_1 \mid\mid L_2$ of L_1 and L_2. Formally:

Definition 5.5 (Non-contradictory Languages).
Let I and O be finite sets of input and output variables with $I \cap O = \emptyset$ and let $V = I \cup O$. Let $V_1, V_2 \subseteq V$ be finite sets of variables. Let $I_1 = I \cap V_1$ and $I_2 = I \cap V_2$. Let $L_1 \subseteq (2^{V_1})^\omega$ and $L_2 \subseteq (2^{V_2})^\omega$ be languages. Then, L_1 and L_2 are called *non-contradictory* if, and only if

$$\forall \sigma \in L_1.\, \forall \sigma' \in L_2.\, (\sigma \cap I_2 = \sigma' \cap I_1) \rightarrow \sigma \cup \sigma' \in L_1 \mid\mid L_2.$$

Completeness of the parallel composition of the transducers representing the subresults of modular synthesis follows immediately if the component specifications constitute non-contradictory language. Recall that, intuitively, the parallel composition $\mathcal{T}_1 \mid\mid \mathcal{T}_2$ of two deterministic and complete transducers $\mathcal{T}_1$ and $\mathcal{T}_2$ realizing languages L_1 and L_2, respectively, can only be incomplete if there exist traces of $\mathcal{T}_1$ and $\mathcal{T}_2$ that agree on shared inputs and thus should be combined into a trace of $\mathcal{T}_1 \mid\mid \mathcal{T}_2$ but that do not agree on shared outputs, preventing composability of the outputs at some point in time. If L_1 and L_2 are non-contradictory, however, all traces of $\mathcal{T}_1$ and $\mathcal{T}_2$ lie in L_1 and L_2, respectively. Therefore, all traces of $\mathcal{T}_1$ and $\mathcal{T}_2$ that agree on shared inputs also agree on shared outputs by definition of non-contradictory languages, preventing incompleteness. Formally, completeness of $\mathcal{T}_1 \mid\mid \mathcal{T}_2$ can be shown as follows:

Lemma 5.3. *Let I and O be finite sets of input and output variables with $I \cap O = \emptyset$ and let $V = I \cup O$. Let V_1 and V_2 be finite sets with $V_1, V_2 \subseteq V$. Let $L_1 \subseteq (2^{V_1})^\omega$ and $L_2 \subseteq (2^{V_2})^\omega$ be realizable languages. Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be deterministic and complete finite-state transducers realizing L_1 and L_2, respectively. If L_1 and L_2 are non-contradictory, then $\mathcal{T}_1 \mid\mid \mathcal{T}_2$ is complete.*

Proof. Let $\mathcal{T}_1 = (T_1, T_{1,0}, \tau_1, \ell_1)$, $\mathcal{T}_2 = (T_2, T_{2,0}, \tau_2, \ell_2)$, and $\mathcal{T}_1 \mid\mid \mathcal{T}_2 = (T, T_0, \tau, \ell_2)$. Let $I_1 = I \cap V_1$, $I_2 = I \cap V_2$, $O_1 = O \cap V_1$, and $O_2 = O \cap V_2$. Let $(u, v) \in T$ and $\iota \in 2^I$ be some state of and some and input valuation of $\mathcal{T}_1 \mid\mid \mathcal{T}_2$, respectively. Since both $\mathcal{T}_1$ and $\mathcal{T}_2$ are complete by assumption, there exist $u' \in T_1$, $v' \in T_2$, $o_1 \in 2^{O_1}$, and $o_2 \in 2^{O_2}$ such that $(u, \iota \cap I_1, u') \in \tau_1$ and $(v, \iota \cap I_2, v') \in \tau_2$ as well as $(u, \iota \cap I_1, o_1) \in \ell_1$ and $(v, \iota \cap I_2, o_2) \in \ell_2$ hold. By assumption, $I \cap O = \emptyset$ holds and thus, in particular, we have $I_i \cap O_j = \emptyset$ for all $i, j \in \{1, 2\}$ by construction of I_1, I_2, O_1, and O_2. Thus, it follows with the definition of the parallel composition of finite-state transducers that $((u, v), \iota, (u', v')) \in \tau$ holds and therefore $\mathcal{T}_1 \mid\mid \mathcal{T}_2$ is transition-complete.

It remains to show that $\mathcal{T}_1 \mid\mid \mathcal{T}_2$ is also labeling-complete and thus, in particular, that there exists some $o \in 2^O$ such that $((u, v), \iota, o) \in \ell$ holds. Since we assume without loss of generality that all states of a finite-state transducer occur in some of its paths (see Section 2.6.1), there exists a path π of $\mathcal{T}_1 \mid\mid \mathcal{T}_2$ that visits (u, v). Let $\gamma \in (2^I)^\omega$ such that $\pi \in \mathit{Paths}(\mathcal{T}_1 \mid\mid \mathcal{T}_2, \gamma)$ holds and let $\sigma \in \mathit{Traces}(\mathcal{T}_1 \mid\mid \mathcal{T}_2, \gamma)$ be the trace corresponding to π. Similar to the second part of the proof of Lemma 2.2, we can thus show that there exist paths $\pi^1 \in \mathit{Paths}(\mathcal{T}_1, \gamma^1)$ and $\pi^2 \in \mathit{Paths}(\mathcal{T}_2, \gamma^2)$ of $\mathcal{T}_1$ and $\mathcal{T}_2$ for inputs sequences $\gamma^1 \in (2^{I_1})^\omega$, $\gamma^2 \in (2^{I_2})^\omega$ such that both $\#_1(\pi^1_k) = i$ and $\#_1(\pi^2_k)$ hold for some point in time $k \geq 0$. Furthermore, since $I_i \cap O_j = \emptyset$ holds for all $i, j \in \{1, 2\}$ as observed above, it follows that $\gamma^i = \gamma \cap I_i$ holds for all $i \in \{1, 2\}$. Let $\sigma' \in \mathit{Traces}(\mathcal{T}_1, \gamma \cap I_1)$ and $\sigma'' \in \mathit{Traces}(\mathcal{T}_2, \gamma \cap I_2)$ be the traces corresponding to π^1 and π^2, respectively. By definition of traces, we have $\sigma \cap I = \gamma$ as well as $\sigma^i \cap I_i = \gamma \cap I_i$ for $i \in \{1, 2\}$ and therefore $(\sigma' \cap I_1) \cap I_2 = (\sigma'' \cap I_2) \cap I_1$ follows. Furthermore, we have $\sigma^i \in (2^{V_i})^\omega$ for $i \in \{1, 2\}$ and thus $\sigma^i \cap I_i = \sigma \cap I$ holds by construction of I_i. Hence, $(\sigma' \cap I) \cap I_2 = (\sigma'' \cap I) \cap I_1$ holds and therefore, since $I_1, I_2 \subseteq I$ holds by construction, we have $\sigma' \cap I_2 = \sigma'' \cap I_1$. By assumption, $\mathcal{T}_i$ realizes L_i and thus, in particular, $\sigma^i \in L_i$ holds for all $i \in \{1, 2\}$. Furthermore, L_1 and L_2 are non-contradictory languages by assumption and therefore $\sigma' \cup \sigma'' \in L_1 \mid\mid L_2$ follows immediately from the definition of non-contradictory languages. Hence, $\sigma' \cap V_2 = \sigma'' \cap V_1$ holds by definition of language composition. Thus, in particular, we have $\sigma' \cap O_2 = \sigma'' \cap O_1$ since $O_i \subseteq V_i$ holds for $i \in \{1, 2\}$ by construction. Therefore, $(\sigma' \cap O_1) \cap O_2 = (\sigma'' \cap O_2) \cap O_1$ follows and thus, since $\sigma'_k \cap O_1 = o_1$ and $\sigma''_k \cap O_2 = o_2$ hold by construction of σ' and σ'', we have $o_1 \cap O_2 = o_2 \cap O_1$. Hence, there exists some $o \in 2^{O_1 \cup O_2}$ such that $o_1 \cup o_2 = o$ holds since $o_1 \in 2^{O_1}$ and $o_2 \in 2^{O_2}$ do not define contradictory valuations of shared output variables. Since $I_i \cap O_j = \emptyset$ holds for all $i, j \in \{1, 2\}$ as observed above and since both $(u, \iota \cap I_1, o_1) \in \ell_1$ and $(v, \iota \cap I_2, o_2) \in \ell_2$ hold by construction, it follows from the definition of the parallel composition of finite-state transducers that there is some $o \in 2^{O_1 \cup O_2}$ such that $((u, v), \iota, o) \in \tau$ holds. Therefore, $\mathcal{T}_1 \mid\mid \mathcal{T}_2$ is labeling-complete as well. □

Hence, Lemmas 5.2 and 5.3 together allow for concluding that the parallel composition of deterministic and complete finite-state transducers is deterministic and complete as well and has a finite number of states as long as the languages that the individual transducers realize are non-contradictory. Therefore, composability of the subresults in modular synthesis, i.e., requirement (i) for soundness and completeness of modular synthesis, can be ensured by requiring the languages of the component specifications to be non-contradictory.

5.2.2. Independent Sublanguages

Next, we consider the remaining two requirements, i.e., (ii) realization of the full system specification by the parallel composition of the subresults if all subresults represent component strategies and violating of the full system specification by the extension of a component's counterstrategy for all possible system behaviors otherwise, and (iii) equirealizability of the initial specification and the subspecifications. We first focus on requirement (ii). Note that the requirement of non-contradictory composability of the subresults is a necessary condition for the first part of requirement (ii). If the subresults cannot be composed, they cannot realize the system specification. Consequently, the language-based conditions that ensure that the parallel composition of the subresults realizes the system specification is closely related to the concept of non-contradictory languages.

By construction of the modular synthesis approach, the subresults are component strategies that realize the respective component specification if all synthesis subtasks succeed. Hence, in this case, the synthesized component strategies produce only computations that lie in the language of the respective component specification. We thus pose the condition that the composition of the languages of the component specifications is exactly the language of the whole system specification, i.e., that the languages of the subspecifications are so-called *sublanguages*. Two languages $L_1 \subseteq (2^{V_1})^\omega$ and $L_2 \subseteq (2^{V_2})^\omega$ are called sublanguages of a language $L \subseteq (2^{V_1 \cup V_2})^\omega$ if, and only if, $L_1 \mid\mid L_2 = L$ holds.

Note that, in general, the fact that two languages L_1 and L_2 are sublanguages of another language L does not necessarily allow for concluding that the parallel composition of strategies s_1 and s_2 realizing L_1 and L_2, respectively, realizes L. There can be input sequences γ, γ' for s_1 and s_2 that agree on shared input variables but on which s_1 and s_2 produce computations $comp(s_1, \gamma)$ and $comp(s_2, \gamma')$, respectively, that do not agree on shared output variables. By definition of language composition, $L_1 \mid\mid L_2$ then does not contain a word that agrees with $\gamma \cup \gamma'$ on the input variables. If L does not contain such a word either, L is unrealizable, and thus, in particular, the parallel composition of s_1 and s_2 does not realize L. Yet, L_1 and L_2 can be sublanguages of L. This situation occurs if s_1 and s_2 produce computations that do not agree on shared outputs, i.e., if s_1 and s_2 are not composable. Hence, we further need to ensure that L_1 and L_2 are also non-contradictory. Consequently, we define *independent sublanguages* as follows:

Definition 5.6 (Independent Sublanguages)**.**
Let I and O be finite sets of input and output variables with $I \cap O = \emptyset$ and let $V = I \cup O$. Let V_1 and V_2 be finite sets with $V_1, V_2 \subseteq V$. Let $L_1 \subseteq (2^{V_1})^\omega$, $L_2 \subseteq (2^{V_2})^\omega$, and $L \subseteq (2^{V_1 \cup V_2})^\omega$ be languages. Then, L_1 and L_2 are called *independent sublanguages* of L if, and only if, L_1 and L_2 are non-contradictory and $L_1 \mid\mid L_2 = L$ holds.

If the languages of the component specifications are independent sublanguages of the initial specification, it follows immediately that their parallel composition is a strategy for the full system that realizes the initial specification. Since the languages of the component specifications are non-contradictory, the parallel composition of the component strategies is deterministic and complete by Lemmas 5.2 and 5.3. Furthermore, every trace of the parallel compositions is, restricted to the variables of the respective component, a trace of every component strategy by Lemma 2.2. Since the component strategies realize the component specifications and since the component specifications are sublanguages of the initial specification, the parallel composition of the component strategies realizes the initial specification. Formally:

Lemma 5.4. *Let I and O be finite sets of input and output variables with $I \cap O = \emptyset$ and let $V = I \cup O$. Let V_1 and V_2 be finite sets such that $V_1, V_2 \subseteq V$ holds. Let $L \subseteq (2^V)^\omega$ be a language. Let $L_1 \subseteq (2^{V_1})^\omega$ and $L_2 \subseteq (2^{V_2})^\omega$ be realizable languages. Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be deterministic and complete finite-state transducers realizing L_1 and L_2, respectively. If L_1 and L_2 are independent sublanguages of L, then $\mathcal{T}_1 \mid\mid \mathcal{T}_2$ realizes L.*

Proof. Since $\mathcal{T}_1$ and $\mathcal{T}_2$ are deterministic and complete finite-state transducers realizing L_1 and L_2, respectively, and since L_1 and L_2 are independent and thus non-contradictory by assumption, it follows with Lemmas 5.2 and 5.3 that $\mathcal{T}_1 \mid\mid \mathcal{T}_2$ is deterministic and complete as well. Hence, it remains to show that $\mathit{Traces}(\mathcal{T}_1 \mid\mid \mathcal{T}_2) \subseteq L$ holds. Let $\sigma \in \mathit{Traces}(\mathcal{T}_1 \mid\mid \mathcal{T}_2)$. By definition of the parallel composition of transducers, $\mathcal{T}_1 \mid\mid \mathcal{T}_2$ is a $(2^{(I_1 \cup I_2) \setminus (O_1 \cup O_2)}, 2^{O_1 \cup O_2})$-transducer and thus $\mathit{Traces}(\mathcal{T}_1 \mid\mid \mathcal{T}_2) \subseteq (2^{V_1 \cup V_2})^\omega$ holds. Since $\mathcal{T}_1 \mid\mid \mathcal{T}_2$ is complete, it produces infinite traces only. Furthermore, since $I \cap O = \emptyset$ and both $V_1 \subseteq V$ and $V_2 \subseteq V$ hold by assumption, $\sigma \cap V_1 \in \mathit{Traces}(\mathcal{T}_1)$ and $\sigma \cap V_2 \in \mathit{Traces}(\mathcal{T}_2)$ follows with Lemma 2.2. Since $\mathcal{T}_1$ and $\mathcal{T}_2$ realize L_1 and L_2, respectively, we have $(\sigma \cap V_1) \in L_1$ and $(\sigma \cap V_2) \in L_2$. Let $\sigma' := \sigma \cap V_1$ and $\sigma'' := \sigma \cap V_2$. By construction, σ' and σ'' agree on shared variables. Hence, by definition of language composition, $\sigma' \cup \sigma'' \in L_1 \mid\mid L_2$ holds. Furthermore, we have $\sigma = \sigma' \cup \sigma''$ since $\sigma \in (2^{V_1 \cup V_2})^\omega$ holds and since σ' and σ'' agree on shared variables. Consequently, $\sigma \in L_1 \mid\mid L_2$ holds. Since L_1 and L_2 are independent sublanguages of L by assumption, we have $L_1 \mid\mid L_2 = L$ and therefore $\sigma \in L$ follows. Since we chose the trace $\sigma \in \mathit{Traces}(\mathcal{T}_1 \mid\mid \mathcal{T}_2)$ arbitrarily, we have $\mathit{Traces}(\mathcal{T}_1 \mid\mid \mathcal{T}_2) \subseteq L$. Therefore, $\mathcal{T}_1 \mid\mid \mathcal{T}_2$ indeed realizes L. □

Hence, Lemma 5.4 together with Lemmas 5.2 and 5.3 allows for concluding that, if all synthesis subtasks in modular synthesis succeed and thus produce finite-state transducers representing component strategies, the subresults, i.e., the separately synthesized component strategies, are composable without contradiction and that their parallel composition realizes the initial specification as long as the languages of the component specifications are independent sublanguages of the initial specification, i.e., as long as they are non-contradictory and their composition is precisely the language of the initial specification. Therefore, requiring the languages of the component specifications in modular synthesis to be independent sublanguages of the initial specification ensures that both requirement (i) and the first part of requirement (ii) for soundness and completeness of modular synthesis are satisfied.

Furthermore, if two languages $L_1 \in (2^{V_1})^\omega$ and $L_2 \in (2^{V_2})^\omega$ are independent sublanguages of a language $L \in (2^V)^\omega$, then it follows from the definition of language composition that every word that lies in L also lies, restricted to the variables of the respective sublanguage, in L_i for $i \in \{1, 2\}$. Therefore, Corollary 5.1 allows for concluding that the counterstrategy extension according to Definition 5.3 of a counterstrategy for one of the languages L_1 and L_2 is a counterstrategy for the language L. Formally:

Lemma 5.5. *Let I and O be finite sets of input and output variables with $I \cap O = \emptyset$ and let $V = I \cup O$. Let V_1 and V_2 be finite sets such that $V_1, V_2 \subseteq V$ holds. Let $L \subseteq (2^V)^\omega$, $L_1 \subseteq (2^{V_1})^\omega$, and $L_2 \subseteq (2^{V_2})^\omega$ be languages. Suppose that language L_i is unrealizable for some $i \in \{1, 2\}$ and let $\mathcal{T}_1^c$ be a counterstrategy transducer for L_i. If L_1 and L_2 are independent sublanguages of L, then the counterstrategy extension $\mathcal{T}^c$ of $\mathcal{T}_i^c$ to I and O is a counterstrategy for L.*

Proof. Let $\sigma \in L$ be some word that lies in L. Since L_1 and L_2 are independent sublanguages of L by assumption, particularly $L_1 \mid\mid L_2 = L$ holds. Thus, we have $\sigma \in L_1 \mid\mid L_2$ as well. Hence, we have, in particular, $(\sigma \cap V_i) \in L_i$ by Proposition 5.1. Since we chose the word $\sigma \in L$ arbitrarily, $\{\sigma \cap V_i \mid \sigma \in L\} \subseteq L_i$ follows. Hence, by Corollary 5.1, the counterstrategy extension $\mathcal{T}^c$ constructed according to Definition 5.3 is a counterstrategy transducer for L. □

Thus, if a synthesis subtask in modular synthesis fails and a counterstrategy is computed, then the extension of this counterstrategy to the full system is a counterstrategy for the initial specification, i.e., it violates the initial specification for all possible system behavior, as long as the component specifications are (independent) sublanguages of the initial specification. Therefore, requiring the languages of the component specifications in modular synthesis to be independent sublanguages of the initial specification further ensures that the second part of requirement (ii) for soundness and completeness of modular synthesis is satisfied.

In the following, we consider requirement (iii) for soundness and completeness of modular synthesis, i.e., equirealizability of the initial specification and all component specifications. Relying on the previous results introduced for requirement (ii), we show that equirealizability is insured if the languages of the component specifications are independent sublanguages of the initial specification. Particularly, it then follows immediately from Lemma 5.4 that the initial specification is realizable if the component specifications are. If, in contrast, one of the component specifications is unrealizable, then Lemma 5.5 allows for concluding unrealizability of the initial specification. Formally:

Lemma 5.6. *Let I and O be finite sets of input and output variables with $I \cap O = \emptyset$ and let $V = I \cup O$. Let V_1 and V_2 be finite sets such that $V_1, V_2 \subseteq V$. Let $L \subseteq (2^V)^\omega$, $L_1 \subseteq (2^{V_1})^\omega$, and $L_2 \subseteq (2^{V_2})^\omega$ be languages. If L_1 and L_2 are independent sublanguages of L, then L is realizable if, and only if, both L_1 and L_2 are realizable.*

Proof. First, let both languages L_1 and L_2 be realizable. Then, there exist deterministic and complete finite-state transducers $\mathcal{T}_1$ and $\mathcal{T}_2$ realizing L_1 and L_2, respectively. By Lemma 5.4, their parallel composition $\mathcal{T}_1 \mid\mid \mathcal{T}_2$ realizes L since L_1 and L_2 are independent sublanguages of L by assumption. Therefore, L is realizable. Second, let L_i be unrealizable for some $i \in \{1,2\}$. Then, there exists a finite-state counterstrategy $(2^{V_i \cap O}, 2^{V_i \cap I})$-transducer $\mathcal{T}_i^c$ for L_i. We extend $\mathcal{T}_i^c$ to a $(2^O, 2^I)$-transducer $\mathcal{T}^c$ as described in Definition 5.3. By Lemma 5.5, the transducer $\mathcal{T}^c$ represents a counterstrategy transducer for L. Therefore, L is unrealizable. □

Hence, requiring the languages of the component specifications in modular synthesis to be independent sublanguages of the initial specification also ensures that requirement (iii) is satisfied. Therefore, together with Lemmas 5.2 to 5.5 it follows that all three requirements for soundless and completeness of modular synthesis are satisfied if the languages of the component specifications are independent sublanguages of the initial specification.

5.2.3. Independence Criterion

Utilizing the language-based conditions introduced above, namely non-contradictory languages and sublanguages, joined in the notion of independent sublanguages, we can now state the language-based independence criterion for modular synthesis of monolithic systems. Recall that a component $c = (L_c, I_c, O_c)$ consists of a subspecification, here represented by a language L_c, and sets of component inputs I_c and component outputs O_c. Note that, by definition of components, $I_c \cap O_c = \emptyset$ holds and that L_c is a specification over component inputs and outputs, i.e., $L_c \subseteq (2^{V_c})^\omega$, where $V_c = I_c \cup O_c$. We define the independence of a decomposition as follows:

Definition 5.7 (Independence Criterion)**.**
Let I and O be finite sets of input and output variables with $I \cap O = \emptyset$ and let $V = I \cup O$. Let $\mathbb{D} = \langle c_1, \dots, c_n \rangle$ be a decomposition of (I, O) with $c_i = (L_i, I_i, O_i)$ for $c_i \in \mathbb{D}$. Let $L \subseteq (2^V)^\omega$ be a language. Then, $\mathbb{D}$ is called an *independent decomposition* if, and only if, $L_1, \dots, L_n$ are independent sublanguages of L.

In the previous sections, we have shown that if the languages component specifications in modular synthesis are independent sublanguages of the initial specification, then (i) if all component specifications are realizable, then component strategies realizing them are composable, (ii) the parallel composition of the component strategies realizes the initial specification if all component specifications are realizable and, otherwise, the extension of a component's counterstrategy is a counterstrategy for the initial specification, and (iii) the initial specification and all component specifications are equirealizable. These three properties of the components and their specifications eradicate the pitfalls in modular synthesis that can prevent soundness and completeness. Therefore, modular synthesis as described in Algorithm 5.1 is sound and complete if the decomposition adheres to the language-based independence criterion:

Theorem 5.1 (Soundness and Completeness)**.** *Let $\mathcal{A}$ be a monolithic architecture with input variables I and output variables O. Let $L \subseteq (2^{I \cup O})^\omega$ be a language. Suppose that Algorithm 5.1 utilizes a decomposition algorithm producing an independent decomposition $\mathbb{D}$ according to the language-based independence criterion. If modular synthesis returns* (`true`, $\mathcal{T}$) *on input L, I, O, then $\mathcal{T}$ realizes L. If it returns* (`false`, $\mathcal{T}^c$)*, then $\mathcal{T}^c$ is a counterstrategy transducer for L.*

Proof. First, suppose that modular synthesis returns (`true`, $\mathcal{T}$) on input L, I, O. Then, none of the component specifications of the components in $\mathbb{D}$ is unrealizable as otherwise the return statement in line 11 in Algorithm 5.1 is not reached. By assumption, the decomposition algorithm produces a decomposition that adheres to the language-based independence criterion. Hence, the component specifications are independent sublanguages of the initial specification. From recursively applying Lemma 5.6 it thus follows that L is realizable as well. Furthermore, since the component strategies are deterministic and complete finite-state transducers by construction, it follows from Lemma 5.4 that the parallel composition of the component strategies realizes L. Since $\mathcal{T}$ is the parallel composition of all component strategies, $\mathcal{T}$ realizes L.

Second, suppose that modular synthesis returns (`false`, $\mathcal{T}^c$) on input L, I, O. Then, there exists a component $c_i \in \mathbb{D}$ with unrealizable component specification L_i. Since the decomposition algorithm produces decompositions that adhere to the independence criterion by assumption, the component specifications are independent sublanguages of the initial specification. Thus, from recursively applying Lemma 5.6, it follows that L is unrealizable as well. Moreover, we obtain with Lemma 5.5 that extending the counterstrategy transducer $\mathcal{T}_i^c$ for L_i as described in Definition 5.3, which is returned by Algorithm 5.1, is a counterstrategy transducer $\mathcal{T}^c$ for L. $\square$

The language-based independence criterion from Definition 5.7 thus allows for characterizing decompositions of a monolithic system that ensure soundness and completeness of modular synthesis. Hence, as long as a decomposition algorithm produces components that are independent according to the language-based independence criterion, i.e., as long as the component specifications are independent sublanguages of the initial system specification, the decomposition algorithm can be safely used in modular synthesis. In the following section, we introduce an approximative variant of the independent sublanguages criterion that is specifically tailored to specifications given as LTL formulas.

5.3. Independent LTL Specifications

In this section, we lift the language-based independence criterion to specifications given as LTL formulas. Hence, we formulate the notion of independence directly on the LTL specification and not on its language. This allows for determining whether or not a decomposition ensures soundness and completeness of modular synthesis without computing the language of the LTL specification. Afterward, we present a decomposition algorithm for LTL specifications that produces decompositions that, by construction, adhere to the LTL independence criterion. Thus, when utilizing this decomposition algorithm in modular synthesis for LTL specifications, soundness and completeness are guaranteed.

The LTL independence criterion is vaguely inspired by Dureja and Rozier's work on more scalable LTL model checking [DR18]. They introduce a preprocessing algorithm for model checking that analyzes dependencies between the properties that need to be checked. For instance, they search for dependencies of the form $\varphi_1 \rightarrow \varphi_2$, which allows them to cancel the model checking task for φ_2 if the one for φ_1 succeeded. We lift the idea of analyzing dependencies from model checking to synthesis. However, due to the different nature of compositional model checking and synthesis, the dependency analysis in our approach differs inherently from the one presented in [DR18] in both their goal and their realization.

For ease of presentation, we assume in the remainder of this chapter that the language $\mathcal{L}(\varphi)$ of an LTL formula φ over atomic propositions V reaches over $\mathit{prop}(\varphi) \subseteq V$, i.e., the atomic propositions that actually occur in φ, instead of the full set V. Hence, $\mathcal{L}(\varphi) \subseteq (2^{\mathit{prop}(\varphi)})^\omega$ holds. Note that such a language can easily be extended to a language $L \subseteq (2^V)^\omega$ that ranges over the full set of atomic propositions: $L = \left\{\sigma \in (2^V)^\omega \mid \sigma \cap \mathit{prop}(\varphi) \in \mathcal{L}(\varphi)\right\}$. Furthermore, a transducer that realizes $\mathcal{L}(\varphi)$ can easily be extended to a transducer realizing L using transducer extension (see Definition 5.3), i.e., by, intuitively, ignoring inputs in $V \setminus \mathit{prop}(\varphi)$ and choosing arbitrary valuations for outputs in $V \setminus \mathit{prop}(\varphi)$. Since clearly $\mathcal{L}(\varphi) = \{\sigma \cap L \mid \sigma \in L\}$ holds by construction of L, it follows with Lemma 5.1 that the extended transducer realizes L.

5.3.1. Syntactic LTL Independence

The main idea of the notion of independence for LTL specifications is to analyze the initial LTL formula as well as the LTL component specifications syntactically in order to determine whether or not a decomposition is independent. This eradicates the need of computing the language of the LTL specification to determine whether or not a decompositions is independent and thus, in particular, no checks for language containment are needed.

For the syntactic specification analysis, we focus on *conjuncts* of the LTL formula representing the initial specification because of convenient properties of their semantics. Given a conjunctive

LTL formula $\varphi = \varphi_1 \wedge \ldots \wedge \varphi_m$ with m conjuncts, it follows from the semantics of conjunction that the languages $\mathcal{L}(\varphi_1), \ldots, \mathcal{L}(\varphi_m)$ of the conjuncts of φ form sublanguages of the language $\mathcal{L}(\varphi)$ of the initial specification φ. Formally:

Lemma 5.7. *Let $\varphi = \varphi_1 \wedge \varphi_2$ be an LTL formula. Then, $\mathcal{L}(\varphi_1) \mid\mid \mathcal{L}(\varphi_2) = \mathcal{L}(\varphi)$.*

Proof. First, let $\sigma \in \mathcal{L}(\varphi)$ be some infinite word satisfying φ. For all $i \in \{1, 2\}$, the satisfaction of conjunct φ_i of φ only depends on the variables occurring in φ_i and thus on the variables in the set $prop(\varphi_i)$. Thus, by the semantics of conjunction, $\sigma \cap prop(\varphi_i) \in \mathcal{L}(\varphi_i)$ holds for all $i \in \{1, 2\}$. Since clearly $(\sigma \cap prop(\varphi_1)) \cap prop(\varphi_2) = (\sigma \cap prop(\varphi_2)) \cap prop(\varphi_1)$ holds, the sequences $\sigma \cap prop(\varphi_1)$ and $\sigma \cap prop(\varphi_2)$ do not define contradictory valuations for shared variables and therefore $(\sigma \cap prop(\varphi_1)) \cup (\sigma \cap prop(\varphi_2)) \in \mathcal{L}(\varphi_1) \mid\mid \mathcal{L}(\varphi_2)$ follows. Furthermore, since we have $prop(\varphi_1) \cup prop(\varphi_2) = prop(\varphi)$ by definition of φ, we have $\sigma = (\sigma \cap prop(\varphi_1)) \cup (\sigma \cap prop(\varphi_2))$. Therefore, $\sigma \in \mathcal{L}(\varphi_1) \mid\mid \mathcal{L}(\varphi_2)$ follows. Since we chose the word $\sigma \in \mathcal{L}(\varphi)$ satisfying φ arbitrarily, we have $\mathcal{L}(\varphi) \subseteq \mathcal{L}(\varphi_1) \mid\mid \mathcal{L}(\varphi_2)$.

Next, let $\sigma \in \mathcal{L}(\varphi_1) \mid\mid \mathcal{L}(\varphi_2)$. Then, there are infinite words $\sigma' \in \mathcal{L}(\varphi_1)$ and $\sigma'' \in \mathcal{L}(\varphi_2)$ with $\sigma' \cap prop(\varphi_2) = \sigma'' \cap prop(\varphi_1)$ and $\sigma = \sigma' \cup \sigma''$. Thus, in particular both $\sigma \cap prop(\varphi_1) \in \mathcal{L}(\varphi_1)$ and $\sigma \cap prop(\varphi_2) \in \mathcal{L}(\varphi_2)$ hold. Therefore, $\sigma \in \mathcal{L}(\varphi_1 \wedge \varphi_2)$ follows immediately with the semantics of conjunction and thus $\sigma \in \mathcal{L}(\varphi)$ holds. Since we chose the word $\sigma \in \mathcal{L}(\varphi_1) \mid\mid \mathcal{L}(\varphi_2)$ arbitrarily, $\mathcal{L}(\varphi) \subseteq \mathcal{L}(\varphi_1) \mid\mid \mathcal{L}(\varphi_2) \subseteq \mathcal{L}(\varphi)$ follows. □

Recall that the language-based independence criterion introduced in Section 5.2 further requires that the languages of the component specifications are non-contradictory. In terms of LTL formulas, non-contradictoriness of the languages of subspecifications can be guaranteed if the formulas do not share output variables:

Lemma 5.8. *Let I and O be finite sets of input and output variables with $I \cap O = \emptyset$. Let V_1 and V_2 be finite sets with $V_1, V_2 \subseteq I \cup O$. Let φ_1 and φ_2 be LTL formulas over atomic propositions V_1 and V_2, respectively. If $prop(\varphi_1) \cap prop(\varphi_2) \subseteq I$ holds, then $\mathcal{L}(\varphi_1)$ and $\mathcal{L}(\varphi_2)$ are non-contradictory.*

Proof. Let $I_1 = I \cap prop(\varphi_1)$, $I_2 = I \cap prop(\varphi_2)$, $O_1 = O \cap prop(\varphi_1)$, and $O_2 = O \cap prop(\varphi_2)$ Let $\sigma \in \mathcal{L}(\varphi_1)$ and $\sigma' \in \mathcal{L}(\varphi_2)$ be infinite words satisfying φ_1 and φ_2, respectively. Since $prop(\varphi_1) \cap prop(\varphi_2) \subseteq I$ holds by assumption, σ and σ' do not share output variables and hence $\sigma \cap O_2 = \sigma' \cap O_1$ follows. If additionally $\sigma \cap I_2 = \sigma' \cap I_1$ holds, then σ and σ' thus agree on all shared variables, i.e., $\sigma \cap prop(\varphi_2) = \sigma' \cap prop(\varphi_1)$. Hence, by definition of language composition, $\sigma \cup \sigma' \in \mathcal{L}(\varphi_1) \mid\mid \mathcal{L}(\varphi_2)$ and therefore $\mathcal{L}(\varphi_1)$ and $\mathcal{L}(\varphi_2)$ are non-contradictory. □

Thus, if the conjuncts of an LTL specifications for the whole system do not share output variables, then their languages are independent sublanguages according to Definition 5.6. They are sublanguages of the initial specification by Lemma 5.7, and they are non-contradictory by Lemma 5.8. We can thus lift the language-based independence criterion from Definition 5.7 to specifications given as LTL formulas by requiring the component specifications to be conjuncts of the specification for the whole system and to not share output variables. It then follows immediately from Lemma 5.7 and Lemma 5.8 that a decomposition that ensures that the component specifications are the conjuncts of the initial specification and if they do not share output variables is independent according to the language-based independence criterion:

Theorem 5.2. *Let I and O be finite sets of input and output variables with $I \cap O = \emptyset$. Let φ be an LTL specification over atomic propositions $I \cup O$. Let $\mathbb{D} = \langle c_1, \dots, c_n \rangle$ be a decomposition of (I, O) with $c_i = (\psi_i, I_i, O_i)$ for all $c_i \in \mathbb{D}$. If $\varphi = \bigwedge_{1 \leq i \leq n} \psi_i$ and if $O_i \cap O_j = \emptyset$ holds for all $i \neq j$ with $1 \leq i, j \leq n$, then the decomposition $\mathbb{D}' = \langle c'_1, \dots, c'_n \rangle$ with $c'_i = (\mathcal{L}(\psi_i), I_i, O_i)$ for all $c'_i \in \mathbb{D}'$ is independent according to the language-based independence criterion.*

Therefore, we also call decompositions of LTL formulas that satisfy the above syntactic properties *syntactically independent.* Since syntactical independence implies language-based independence, we can utilize the results from Section 5.2 to reason about such syntactic decompositions of specifications given as LTL formulas. In particular, it follows from Theorem 5.1 that modular synthesis is sound and complete for decompositions of LTL specifications that ensure that the components do not share output variables and that their specifications are the conjuncts of the initial specification:

Corollary 5.2 (Soundness and Completeness)**.** *Let $\mathcal{A}$ be a monolithic architecture with input variables I and output variables O. Let φ be an LTL formula over atomic proposition $I \cup O$. Suppose that Algorithm 5.1 utilizes a decomposition algorithm producing a decomposition $\mathbb{D} = \langle c_1, \dots, c_n \rangle$ with $c_i = (\psi_i, I_i, O_i)$ for all $c_i \in \mathbb{D}$ such that $\varphi = \bigwedge_{1 \leq i \leq n} \psi_i$ and $O_i \cap O_j = \emptyset$ holds for all $i \neq j$ with $1 \leq i, j \leq n$. If modular synthesis returns* (`true`, $\mathcal{T}$) *on input L, I, O, then $\mathcal{T}$ realizes L. If it returns* (`false`, $\mathcal{T}^c$)*, then $\mathcal{T}^c$ is a counterstrategy transducer for L.*

There exist decompositions of LTL formulas that satisfy the language-based independence criterion but not the syntactic independence criterion. Consider, for instance, for the LTL formula $\varphi = \square o_1 \land \square((o_1 \lor \neg o_1) \rightarrow o_2)$, where both o_1 and o_2 are output variables. The languages $\mathcal{L}(\square o_1)$ and $\mathcal{L}(\square o_2)$ form independent sublanguages of $\mathcal{L}(\varphi)$ since the conjunct $\square((o_1 \lor \neg o_1) \rightarrow o_2)$ is equivalent to $\square o_2$. Syntactic independence, in contrast, does not take the equivalence into account. Therefore, it would not detect that o_1 and o_2 are independent.

However, as LTL-based independence is, in contrast to language-based independence, a purely syntactic criterion, it is easy to determine whether or not a decomposition is syntactically independent. In particular, no language containment check is required. Hence, syntactic LTL independence can be seen as a simple and efficient approximation of language-based independence. In the following, we present a decomposition algorithm for specifications given as LTL formulas that produces syntactically independent decompositions only.

5.3.2. LTL Decomposition Algorithm

A decomposition algorithm for LTL formulas that produces syntactically independent decompositions only needs to construct components such that the component specifications (i) are conjuncts of the initial specification and (ii) do not share output variables. Hence, we present a decomposition algorithm that determines which conjuncts of an LTL formula share output variables and group them into component specifications. The component specifications are thus conjuncts of conjuncts of the initial specification, and therefore the component specifications can be seen as conjuncts of the initial formula as well. To determine which conjuncts of an LTL formula $\varphi = \varphi_1 \land \ldots \land \varphi_m$ share output variables, we build the *dependency graph* of the conjuncts of φ based on the output variables occurring in the conjuncts:

Algorithm 5.2: LTL Decomposition Algorithm

```
Input: $\varphi$: LTL, I: List Variable, O: List Variable
Output: components: List (LTL, List Variable, List Variable)
1  $\varphi \leftarrow$ rewrite($\varphi$)
2  formulas $\leftarrow$ removeTopLevelConjunction($\varphi$)
3  dependencyGraph $\leftarrow$ buildDependencyGraph($\varphi$, O)
4  cc $\leftarrow$ dependencyGraph.connectedComponents()
5  subspecs $\leftarrow$ [|cc|+1]: List LTL    // LTL list of length |cc|+1, initialized with true
6  foreach $\varphi_i \in$ formulas do
7      propositions $\leftarrow$ getPropositions($\varphi_i$)
8      foreach ($\psi$,vars) $\in$ zip(subspecs, cc ++ [I]) do
9          if propositions $\cap$ vars $\neq \emptyset$ then
10             $\psi \leftarrow \psi \wedge \varphi_i$
11             break
12 components $\leftarrow$ []: buildComponents(subspecs)
13 return components
```

Definition 5.8 (Conjunct Dependency Graph)**.**
Let I and O be finite sets of input and output variables with $I \cap O = \emptyset$. Let $\varphi = \varphi_1 \wedge \ldots \wedge \varphi_m$ be an LTL formula over atomic propositions $I \cup O$. The *conjunct dependency graph* $\mathcal{D}_\varphi = (\mathcal{V}, \mathcal{E})$ of φ is defined by $\mathcal{V} = O$ and $(a, b) \in \mathcal{E}$ if, and only if, $a \neq b$ and both $a \in prop(\varphi_i)$ and $b \in prop(\varphi_i)$ hold for some $1 \leq i \leq m$.

Hence, the output variables represent the nodes of the conjunct dependency graph $\mathcal{D}_\varphi$. Intuitively, two output variables $a, b \in O$ are connected in $\mathcal{D}_\varphi$ if they occur in the same conjunct of φ. Therefore, two output variables $a, b \in O$ that are contained in the same connected component of $\mathcal{D}_\varphi$ depend on each other in the sense that they either occur in the same conjunct of φ or that they occur in conjuncts that are "connected" by other output variables. Hence, to ensure that component specifications do not share output variables, conjuncts containing a or b need to be assigned to the same component specification. If the output variables of two conjuncts are contained in different connected components of $\mathcal{D}_\varphi$, however, they are not linked via conjuncts. Therefore, they can be assigned to different components while still ensuring that the components are independent according to the LTL independence criterion. Hence, strategies for them can be synthesized separately.

Algorithm 5.2 describes how an LTL formula φ with inputs I and outputs O can decomposed into independent components. First, the formula is rewritten in conjunctive form (line 1). This can be done by, e.g., applying distributivity and pushing temporal operators inwards whenever possible. The rewriting is done in order to maximize the number of top-level conjuncts since the LTL decomposition algorithm only decomposes specifications at conjunctions. Note, however, that the rewriting step is not necessary for the correctness of Algorithm 5.2; it only allows for finding more fine-grained decompositions. Next, the rewritten formula is split into its conjuncts (line 2) which serve as potential subspecifications. Then, the dependency graph of the conjuncts is built (line 3) and its connected components are computed (line 4). The former follows a simple implementation of the construction of the conjunct dependency graph based on Definition 5.8. The latter employs a standard depth-first search (DFS)-based algorithm for finding connected components such as the one described by Hopcroft and Tarjan [HT73].

The LTL decomposition algorithm then proceeds with computing subspecifications that do not share output variables from the list of conjuncts (lines 5 to 11). It iterates through all conjuncts φ_i of φ (line 6) and computes its propositions $\mathit{prop}(\varphi_i)$ (line 7). It then considers all pairs of already computed subspecifications ψ and variable sets `vars` of either a connected component or all input variables (line 8). Note here that given two lists l and l' of length $|l|$ and $|l'|$, respectively, `zip` returns a list l'' of tuples of elements of l and l' such that $l''_j := (l_j, l'_j)$ holds for all $1 \leq j \leq \min\{|l|, |l'|\}$, where l_j denotes the j-th entry of a list l. If the considered variable set `vars` shares variables with $\mathit{prop}(\varphi_i)$, then φ_i is added as a conjunct to the currently considered subspecification and the iteration over the pairs is stopped (lines 9 to 11). Hence, the two foreach-loops successively construct and refine subspecifications for all connected components of the conjunct dependency graph $\mathcal{D}_\varphi$ of φ as well as all inputs by adding all conjuncts containing propositions that lie in the respective connected component or that are an input variable. We also consider the set of all input variables to ensure that conjuncts of φ that do not contain output variables are contained in at least one subspecification. By construction of the conjunct dependency graph, every conjunct can only share variables with a single connected component. Yet, since we also consider all input variables, a conjunct can share variables with both a connected component and the input variables. To ensure that every conjunct is contained in at most one subspecification, we thus employ a break statement (line 11).

Lastly, Algorithm 5.2, defines the components according to the computed subspecifications (line 12). That is, for each subspecification ψ_i, it computes the propositions $\mathit{prop}(\psi_i)$ of ψ_i and, based on these, the inputs and outputs of the component. The component inputs I_i are the input variables occurring in ψ_i, i.e., $I_i = \mathit{prop}(\psi_i) \cap I$, and the component outputs O_i are the output variables occurring in ψ_i, i.e., $O_i = \mathit{prop}(\psi_i) \cap O$. The component is then defined by (ψ_i, I_i, O_i) After computing the full components from the subspecifications, the LTL decomposition algorithm returns the list of components (line 13).

Example 5.1. Consider the LTL formula $\varphi = \square(i_1 \rightarrow o_1) \wedge \Diamond o_2 \wedge (o_3 \,\mathcal{U}(i_2 \wedge o_2))$, where $I = \{i_1, i_2\}$ and $O = \{o_1, o_2, o_3\}$. It is in conjunctive form and thus does no need to be rewritten. It consists of the three conjuncts $\varphi_1 = \square(i_1 \rightarrow o_1)$, $\varphi_2 = \Diamond o_2$, and $\varphi_3 = o_3 \,\mathcal{U}(i_2 \wedge o_2)$. The conjunct dependency graph has three nodes o_1, o_2, and o_3 and there exists an edge between o_2 and o_3 due to conjunct φ_3. Hence, the connected components are given by $\{o_1\}$ and $\{o_2, o_3\}$. The only conjunct that shares variables with the connected component $\{o_1\}$ is φ_1 and thus the subspecification for $\{o_1\}$ is given by $\psi_1 = \square(i_1 \rightarrow o_1)$. Both φ_2 and φ_3 share variables with $\{o_2, o_3\}$ and therefore the subspecification for the second component is given by $\psi_2 = \Diamond o_2 \wedge (o_3 \,\mathcal{U}(i_2 \wedge o_2))$. This results in the two components $c_1 = (\psi_1, \{i_1\}, \{o_1\})$ and $c_2 = (\psi_2, \{i_2\}, \{o_2, o_3\})$. △

Algorithm 5.2 then indeed computes a decomposition of (I, O) that is syntactically independent, i.e., that ensures that the components do not share output variables and that their specifications are the conjuncts of the initial specification:

Lemma 5.9. *Let $\mathcal{A}$ be a monolithic architecture with input variables I and output variables O. Let $V = I \cup O$. Let φ be an LTL formula over atomic propositions V. Suppose that Algorithm 5.2 terminates with a decomposition $\mathbb{D} = \langle c_1, \dots, c_n \rangle$ for input φ, I, O with $c_i = (\psi_i, I_i, O_i)$ for all $c_i \in \mathbb{D}$. Then, $\varphi = \bigwedge_{1 \leq i \leq n} \psi_i$ and $O_i \cap O_j = \emptyset$ holds for all $i \neq j$ with $1 \leq i, j \leq n$.*

Proof. Observe that rewriting φ into conjunctive form does not alter its language. Thus, we use φ and the rewritten formula synonymously in the following. Algorithm 5.2 builds the conjunct dependency graph $\mathcal{D}_\varphi$ according to Definition 5.8 and computes its connected components. The algorithm then iterates over all conjuncts of φ and assigns them to the subspecifications mapped to connected components and the set of input variables they share variables with. Thus, since the output variables O constitute the nodes of $\mathcal{D}_\varphi$, it follows that every conjunct of φ is added to *at least* one subspecification. Hence, since the conjuncts of φ are added to a subspecification by adding them as a conjunct, $\varphi = \bigwedge_{1 \leq i \leq n} \psi_i$ follows with the semantics of conjunction.

Next, we show that every conjunct of φ is added to *at most* one subspecification to ensure disjointness of the output variables of the components. Clearly, every output variable is contained in exactly one connected component since the output variables constitute the set of nodes of $\mathcal{D}_\varphi$. Moreover, by definition of $\mathcal{D}_\varphi$, all output variables that are contained in the same conjunct of φ are contained in the same connected component. Thus, every conjunct can only share variables with a single connected component; therefore, every conjunct is only added to a single subspecification assigned to a connected component. While a conjunct can share variables with both some connected component and the set of input variables, the break statement in line (line 11) ensures that no conjunct is added to both a subspecification assigned to a connected component and the subspecification for input-only conjuncts. Hence, every conjunct is added to at most one subspecification, and therefore every conjunct occurs in at most one component specification. Thus, $\mathit{prop}(\psi_i) \cap \mathit{prop}(\psi_j) \subseteq I$ follows for all $i \neq j$ with $1 \leq i, j \leq n$. Since the component interface of component c_i is defined by $I_i = \mathit{prop}(\psi_i) \cap I$ and $O_i = \mathit{prop}(\psi_i) \cap O$, we obtain that $O_i \cap O_j = \emptyset$ holds for all $i \neq j$ with $1 \leq i, j \leq n$. □

Hence, since Algorithm 5.2 produces syntactically independent decompositions, soundness and completeness of modular synthesis when using Algorithm 5.2 as decomposition algorithm for LTL specifications follows with Corollary 5.2. However, while the LTL decomposition algorithm is simple and ensures soundness and completeness of modular synthesis, its effectiveness strongly depends on the structure of the LTL formula. Therefore, we consider optimizations of the notion syntactic LTL independence for common classes of LTL formulas, for which Algorithm 5.2 does not find fine-grained decompositions, in the subsequent sections.

5.4. Assumption Dropping for LTL Decomposition

In practice, LTL formulas in *assume-guarantee form* are very common. Assume-guarantee formulas require that some *guarantees* are satisfied if certain *assumptions* are satisfied. Thus, an assume-guarantee LTL formula is of the form $\varphi = \bigwedge_{i=1}^{\ell} \varphi_i \rightarrow \bigwedge_{j=1}^{m} \psi_j$, where the formulas φ_i are assumptions and the formulas ψ_j are guarantees. Such formulas allow, for instance, for restricting the possible environment behavior by posing assumptions on input variables. The system is then only required to satisfy the guarantees as long as the environment behaves according to the assumptions. Furthermore, formulas that fall into the well-known *Generalized Reactivity(1)*-fragment of LTL, for which efficient polynomial-time symbolic synthesis algorithms exists [PPS06, BJP+12], are in assume-guarantee form.

However, when rewriting an LTL formula $\varphi = \bigwedge_{i=1}^{\ell} \varphi_i \rightarrow \bigwedge_{j=1}^{m} \psi_j$ in assume-guarantee form into conjunctive form, we obtain $\bigwedge_{j=1}^{m} \left(\bigwedge_{i=1}^{\ell} \varphi_i \rightarrow \psi_j \right)$. While φ consists of m conjuncts after rewriting, all conjuncts contain all ℓ assumptions. Suppose that an output variable a occurs in some assumption φ_i of φ. Then, after rewriting φ into conjunctive form, all conjuncts of φ feature output variable a. Thus, all conjuncts share the output variable a, and therefore they depend on each other in the sense that the system cannot be decomposed into several components. In fact, the basic LTL decomposition algorithm from Algorithm 5.2 described in the previous section returns a single component, namely one with the rewritten form of φ as component specification and the full input and output sets as component interface. However, some guarantee ψ_j of the initial formula φ might be realizable even if φ_i is violated. This can, for instance, be the case if restricting the environment behavior is only necessary for a few parts of the system requirements. Then, intuitively, assumption φ_i can be removed for guarantee ψ_j without altering realizability of the conjunct. Dropping φ_i eliminated the dependency between the conjunct featuring ψ_j and the other conjuncts due to φ_i, possibly resulting in independence of the conjunct featuring ψ_j from the other ones. A decomposition algorithm accounting for this can yield a more fine-grained decomposition and thus smaller synthesis subtasks for modular synthesis than the basic LTL decomposition algorithm presented in the previous section.

In this section, we introduce a criterion for dropping assumptions while ensuring equirealizablity of the resulting formula and the original formula. This allows for extending the results from Sections 5.2 and 5.3 with assumption dropping. In particular, we show how to decompose LTL formulas in assume-guarantee form utilizing assumption dropping while ensuring that the three requirements (i) composability, (ii) realization and violation, respectively, and (iii) equirealizability, which ensure soundness and completeness of modular synthesis, are met. Afterward, we present a decomposition algorithm for assume-guarantee LTL formulas that extends the basic LTL decomposition algorithm from Section 5.3 with assumption dropping.

5.4.1. Criterion for Assumption Dropping

First, we study when an assumption can be removed from an LTL formula while maintaining equirealizability of the original formula and the one without the dropped assumption. Intuitively, given an LTL formula $\varphi = \bigwedge_{i=1}^{\ell} \varphi_i \rightarrow \psi$, we can drop an assumption φ_i if it does not share any variables with the guarantee ψ. However, if φ_i can be violated by the system, i.e., if $\neg\varphi_i$ is realizable, equirealizability cannot be guaranteed when dropping φ_i. As an example, consider the LTL formula $\varphi = \Diamond(i_1 \wedge o_1) \rightarrow \Box(i_2 \wedge o_2)$ with inputs $I = \{i_1, i_2\}$ and outputs $O = \{o_1, o_2\}$. Note that φ consists of a single assumption $\Diamond(i_1 \wedge o_1)$ and a single guarantee $\Box(i_2 \wedge o_2)$. Although assumption and guarantee do not share any variables, the assumption cannot be dropped: a strategy that never sets O_1 to *true* realizes φ trivially since it violates the assumption. Yet, the guarantee, is not realizable since i_2 is an input variable: for all input sequences that do not set i_2 to *true* at every point in time, $\Box(i_1 \wedge o_2)$ is violated no matter how the system behaves.

In addition, dependencies between input variables may yield unrealizability if an assumption is dropped since information about the remaining inputs might get lost. As an example, consider a system with inputs $I = \{i_1, i_2, i_3, i_4\}$ and outputs $O = \{o\}$, and the LTL formula

$$\varphi = ((\square i_1 \rightarrow i_2) \land (\neg \square i_1 \rightarrow i_3) \land (i_2 \leftrightarrow i_4) \land (i_3 \leftrightarrow \neg i_4)) \rightarrow (\square i_1 \leftrightarrow o)\,.$$

Clearly, the only guarantee $\square i_1 \leftrightarrow o$ is not realizable since i_1 is an input variable, and the system would need to predict whether or not i_1 will be set to *true* at every point in time in order to set output o to the correct valuation in the very first time step. The four assumptions of φ, however, provide information on i_1. If i_1 is set to *true* at every point in time, then the assumptions are only satisfied if i_2 is set to *true* in the very first step (first assumption), if, consequently, i_4 is also set to *true* in the very first step (third assumption), and thus if i_3 is set to *false* in the very first step (fourth assumption). If i_1 is not set to *true* at every point in time, then the assumptions are only satisfied if i_3 is set to *true* in the very first step (second assumption), if consequently i_4 is set to *false* in the very first step (fourth assumption), and thus if i_2 is also set to *false* in the very first step (third assumption). Hence, by observing the inputs i_2, i_3, and i_4 in the very first step, we can conclude whether or not sequences satisfying the assumptions set i_1 to *true* in every time step. Therefore, φ is realizable. However, none of the four assumptions can be dropped as otherwise information on i_1 gets lost. Clearly, the first two assumptions cannot be dropped as they share i_1 with the guarantee. If we drop the third assumption, then the connection of i_2 with both i_3 and i_4 is lost. If we drop the fourth assumption instead, we lose the connection of i_3 with both i_2 and i_4. Hence, the input behavior on the very first step is no unique identifier of whether or not i_1 will be set to *true* in every time step for all sequences that satisfy the assumptions anymore. In both cases, the resulting formula is thus unrealizable, although φ is realizable.

We utilize these two observations as well as our initial intuition for when assumptions can be removed for formulating a criterion for safely dropping assumptions in LTL formulas $\varphi = \bigwedge_{i=1}^{\ell} \varphi_i \rightarrow \bigwedge_{j=1}^{m} \psi_j$ in strict assume-guarantee form:

Definition 5.9 (Assumption-Dropping Criterion for Strict Assume-Guarantee Formulas)**.** Let I and O be finite sets of inputs and outputs with $I \cap O = \emptyset$. Let $\varphi = (\varphi_1 \land \varphi_2) \rightarrow \psi$ be an LTL formula over atomic propositions $I \cup O$. Then, φ_2 *qualifies for dropping* if, and only if

1. $\mathit{prop}(\varphi_1) \cap \mathit{prop}(\varphi_2) = \emptyset$,
2. $\mathit{prop}(\varphi_2) \cap \mathit{prop}(\psi) = \emptyset$, and
3. $\neg\varphi_2$ is unrealizable.

Intuitively, an assumption thus qualifies for dropping if it neither shares variables with the guarantees nor with the other assumptions and if it cannot be violated by the system. This criterion indeed allows for safely dropping assumptions, i.e., if an assumption qualifies for dropping according to the assumption-dropping criterion, then the original specification and the one resulting from dropping the assumption are equirealizable. Intuitively, a strategy realizing the formula that does not contain the droppable assumptions can easily be extended to a strategy realizing the initial formula with transducer extension. If the formula resulting from assumption dropping is unrealizable, however, the counterstrategy does not directly extend to a counterstrategy for the initial formula since dropping the assumption may have caused unrealizability. Since the negation of the droppable assumption is unrealizable, however, we can combine the counterstrategy for it with the counterstrategy for the obtained formula to a counterstrategy for the initial formula:

Lemma 5.10. *Let I and O be finite sets of input and output variables with $I \cap O = \emptyset$ and let $V = I \cup O$. Let $\varphi = (\varphi_1 \land \varphi_2) \rightarrow \psi$ be an LTL formula over atomic propositions V. If φ_2 qualifies for dropping for φ, then $\varphi_1 \rightarrow \psi$ is realizable if, and only if, φ is realizable.*

Proof. Let $\varphi' := \varphi_1 \rightarrow \psi$. Let $V_1 := \mathit{prop}(\varphi')$ and $V_2 := \mathit{prop}(\varphi_2)$. Let $I_1 := V_1 \cap I$, $I_2 := V_2 \cap I$, $O_1 := V_1 \cap O$, and $O_2 := V_2 \cap O$. First, suppose that φ' is realizable. Then, there exists a deterministic and complete finite-state $(2^{I_1}, 2^{O_1})$-transducer realizing φ'. Let $\mathcal{T}$ be the finite-state $(2^{I}, 2^{O})$-transducer obtained by extending $\mathcal{T}_1$ according to Definition 5.3 to the full sets I and O. In the following, we show that $\mathcal{L}(\varphi') \subseteq \{\sigma \cap V_1 \mid \sigma \in \mathcal{L}(\varphi)\}$ holds. By definition of V_1, only the variables in V_1 affect the satisfaction of φ'. Hence, $\sigma \cup \sigma' \in \left\{\sigma \in (2^V)^\omega \mid \sigma \cap V_1 \in \mathcal{L}(\varphi')\right\}$ holds for all $\sigma \in \mathcal{L}(\varphi')$ and all $\sigma' \in (2^{V_2})^\omega$. By the semantics of conjunction and implication, we have $\left\{\sigma \in (2^V)^\omega \mid \sigma \cap V_1 \in \mathcal{L}(\varphi')\right\} \subseteq \mathcal{L}(\varphi)$. Thus, $\mathcal{L}(\varphi') \subseteq \{\sigma \cap V_1 \mid \sigma \in \mathcal{L}(\varphi)\}$ holds as well and therefore it follows with Lemma 5.1 that $\mathcal{T}$ realizes φ.

Next, suppose that φ' is unrealizable. Then, there exists a deterministic and complete finite-state counterstrategy $(2^{O_1}, 2^{I_1})$-transducer $\mathcal{T}_1^c$ for φ'. Moreover, Since $\neg\varphi_2$ is unrealizable by assumption, there exists a counterstrategy $(2^{O_2}, 2^{I_2})$-transducer $\mathcal{T}_2^c$ for $\neg\varphi_2$. Thus, $\mathcal{T}_1^c$ and $\mathcal{T}_2^c$ realize $\neg\varphi'$ and φ_2, respectively. Since φ_2 qualifies for dropping for φ, we have both $\mathit{prop}(\varphi_1) \cap \mathit{prop}(\varphi_2) = \emptyset$ and $\mathit{prop}(\varphi_2) \cap \mathit{prop}(\psi) = \emptyset$ by definition of the assumption-dropping criterion. Thus, $V_1 \cap V_2 = \emptyset$ holds. Therefore, it follows immediately that $\mathcal{L}(\neg\varphi')$ and $\mathcal{L}(\varphi_2)$ are independent sublanguages of $L := \left\{\sigma \in (2^V)^\omega \mid \sigma \cap V_1 \in \mathcal{L}(\neg\varphi') \land \sigma \cap V_2 \in \mathcal{L}(\varphi_2)\right\}$. Hence, by Lemma 5.4, $\mathcal{T}_1^c \mid\mid \mathcal{T}_2^c$ realizes L. By the semantics of conjunction, $L = \mathcal{L}(\neg\varphi' \land \varphi_2)$ holds. Moreover, by definition of φ' as well as by the semantics of conjunction and implication, we have $\mathcal{L}(\neg\varphi' \land \varphi_2) = \mathcal{L}(\neg\varphi)$. Hence, $\mathcal{T}_1^c \mid\mid \mathcal{T}_2^c$ realizes $\neg\varphi$. Furthermore, since $\mathcal{T}_1^c$ and $\mathcal{T}_2^c$ are both counterstrategy transducers, they are of the same transducer type, i.e., they are either both Mealy or Moore transducers. The parallel composition does not alter the transducer type if both transducers are of the same type and hence, by definition of counterstrategy transducers, $\mathcal{T}_1^c \mid\mid \mathcal{T}_2^c$ is a counterstrategy transducer for φ. Thus, φ is unrealizable. □

Therefore, dropping assumptions if they qualify for dropping according to the assumption-dropping criterion for strict assume-guarantee formulas, ensures equirealizability of the original formula and the one resulting from dropping these assumptions. Hence, assumptions that qualify for dropping are indeed not necessary for realizability of the original formula in the sense that whether or not there exists a strategy for the system that realizes the specification does not depend on the existence of the assumption.

We can thus immediately utilize the assumption-dropping criterion for decomposing LTL specifications in strict assume-guarantee form, i.e., formulas of the form $\varphi = \bigwedge_{i=1}^{\ell} \varphi_i \rightarrow \bigwedge_{j=1}^{m} \psi_j$, in further cases. First, we rewrite the formula into the conjunctive form $\bigwedge_{j=1}^{m} \left(\bigwedge_{i=1}^{\ell} \varphi_i \rightarrow \psi_j\right)$. Then, for each of the individual guarantees ψ_j, we drop assumptions from the conjunction of assumptions $\bigwedge_{i=1}^{\ell} \varphi_i$ whenever possible according to the assumption-dropping criterion. Note that Lemma 5.10 guarantees *equirealizability* of the initial formula and the formula obtained from assumption dropping. However, it does not guarantee *language equivalence*. In particular, the conjunction of the conjuncts of φ after dropping assumptions is thus not necessarily equivalent to φ. Therefore, when building a decomposition by grouping conjuncts that share input variables similar to basic LTL decomposition after dropping assumptions, the languages of the component specifications are not guaranteed to be sublanguages of $\mathcal{L}(\varphi)$. Hence, we cannot directly reuse the results from the previous sections but show in the following that due to the equirealizability of the initial formula and the formula obtained from assumption dropping, the three requirements for soundness and completeness of modular synthesis are nevertheless satisfied. First, we show that if the resulting conjuncts only share input variables and are realizable, then the parallel composition of transducers realizing them realizes φ:

Lemma 5.11. *Let I and O be finite sets of input and output variables with $I \cap O = \emptyset$. Let $\varphi = (\varphi_1 \land \varphi_2 \land \varphi_3) \rightarrow (\psi_1 \land \psi_2)$ be an LTL formula over atomic propositions $I \cup O$. Suppose that φ_1 qualifies for dropping for $(\varphi_1 \land \varphi_2 \land \varphi_3) \rightarrow \psi_2$ and that φ_2 qualifies for dropping for $(\varphi_1 \land \varphi_2 \land \varphi_3) \rightarrow \psi_1$. Let $\varphi' = (\varphi_1 \land \varphi_3) \rightarrow \psi_1$ and $\varphi'' = (\varphi_2 \land \varphi_3) \rightarrow \psi_2$. Suppose that $prop(\varphi') \cap prop(\varphi'') \subseteq I$ holds. If both φ' and φ'' are realizable, then the parallel composition $\mathcal{T}_1 \mid\mid \mathcal{T}_2$ of transducers $\mathcal{T}_1$ and $\mathcal{T}_2$ realizing φ' and φ'', respectively, realizes φ.*

Proof. First, let both φ' and φ'' be realizable. Then, there exist deterministic and complete finite-state transducers $\mathcal{T}_1$ and $\mathcal{T}_2$ realizing φ' and φ'', respectively. By assumption, we have $prop(\varphi') \cap prop(\varphi'') \subseteq I$. It follows with Lemmas 5.7 and 5.8 and Definition 5.6 that $\mathcal{L}(\varphi')$ and $\mathcal{L}(\varphi'')$ are independent sublanguages of $\mathcal{L}(\varphi' \land \varphi'')$. Hence, by Lemma 5.4, the parallel composition of $\mathcal{T}_1$ and $\mathcal{T}_2$, i.e., $\mathcal{T}_1 \mid\mid \mathcal{T}_2$, is deterministic and complete and realizes $\mathcal{L}(\varphi' \land \varphi'')$. Therefore, $Traces(\mathcal{T}_1 \mid\mid \mathcal{T}_2) \subseteq \mathcal{L}(\varphi' \land \varphi'')$ holds. By the semantics of implication and conjunction as well as by the definitions of φ, φ', and φ'', it then follows that $Traces(\mathcal{T}_1 \mid\mid \mathcal{T}_2) \subseteq \mathcal{L}(\varphi)$ holds as well. Thus, $\mathcal{T}_1 \mid\mid \mathcal{T}_2$ realizes $\mathcal{L}(\varphi)$ and hence φ is realizable. □

If one of the formulas resulting from assumption dropping for the individual conjuncts of $\bigwedge_{j=1}^{m} (\bigwedge_{i=1}^{\ell} \varphi_i \rightarrow \psi_j)$ is unrealizable, in contrast, then there exists a counterstrategy transducer $\mathcal{T}_1^c$ for it. Similar to the equirealizability proof for assumption dropping in general, we can utilize the counterstrategy transducer $\mathcal{T}_2^c$ for the negation of the droppable assumptions to build a counterstrategy transducer for φ from $\mathcal{T}_1^c$. The counterstrategy extension of the parallel composition of $\mathcal{T}_1^c$ and $\mathcal{T}_2^c$ is then guaranteed to be a counterstrategy transducer for φ:

Lemma 5.12. *Let I and O be finite sets of input and output variables with $I \cap O = \emptyset$. Let $\varphi = (\varphi_1 \land \varphi_2 \land \varphi_3) \rightarrow (\psi_1 \land \psi_2)$ be an LTL formula over atomic propositions $I \cup O$. Suppose that φ_1 qualifies for dropping for $(\varphi_1 \land \varphi_2 \land \varphi_3) \rightarrow \psi_2$. Let $\mathcal{T}_2^c$ be a counterstrategy transducer for $\neg\varphi_2$. Let $\varphi' = (\varphi_1 \land \varphi_3) \rightarrow \psi_1$. If φ' is unrealizable, then the counterstrategy extension of $\mathcal{T}_1^c \mid\mid \mathcal{T}_2^c$, where $\mathcal{T}_1^c$ is a counterstrategy transducer for φ', is a counterstrategy transducer for φ.*

Proof. Let $V_1 = prop(\varphi')$, $I_1 = V_1 \cap I$, and $O_1 = V_1 \cap O$. Suppose that φ' is unrealizable. Then, there exists a deterministic and complete finite-state counterstrategy $(2^{O_1}, 2^{I_1})$-transducer $\mathcal{T}_1^c$ for φ'. Since φ_2 qualifies for dropping by assumption, $\neg\varphi_2$ is unrealizable and hence there is a deterministic and complete finite state counterstrategy $(2^{O\cap prop(\varphi_2)}, 2^{I\cap prop(\varphi_2)})$-transducer $\mathcal{T}_2^c$ for $\neg\varphi_2$. Thus, $\mathcal{T}_1^c$ realizes $\neg\varphi'$ and $\mathcal{T}_2^c$ realizes φ_2. Moreover, $prop(\varphi_2) \cap prop(\varphi') = \emptyset$ holds by the assumption-dropping criterion. It thus follows similarly to the second part of the proof of Lemma 5.10 that $\mathcal{T}_1^c \mid\mid \mathcal{T}_2^c$ is a counterstrategy transducer for $\neg(\neg\varphi' \wedge \varphi_2)$: since we have $prop(\varphi_2) \cap prop(\varphi') = \emptyset$ and by the semantics of conjunction, $\mathcal{L}(\neg\varphi')$ and $\mathcal{L}(\varphi_2)$ are independent sublanguages of $\mathcal{L}(\neg\varphi' \wedge \varphi_2)$. Therefore, by Lemma 5.4, the transducer $\mathcal{T}_1^c \mid\mid \mathcal{T}_2^c$ is deterministic and complete and realizes $\neg\varphi' \wedge \varphi_2$. Moreover, since $\mathcal{T}_1^c$ and $\mathcal{T}_2^c$ are both counterstrategy transducers, they are of the same transducer type, i.e., they are either both Mealy or Moore transducers. The parallel composition does not alter the transducer type if both transducers are of the same type and hence, by definition of counterstrategy transducers, $\mathcal{T}_1^c \mid\mid \mathcal{T}_2^c$ is a counterstrategy transducer for $\neg(\neg\varphi' \wedge \varphi_2)$ and thus for $\varphi' \vee \neg\varphi_2$. By construction of φ', we have $\varphi' \vee \neg\varphi_2 = (\varphi_1 \wedge \varphi_2 \wedge \varphi_3) \rightarrow \psi_2$. Hence, by the semantics of conjunction and implication, we clearly have $\{\sigma \cap prop(\varphi' \vee \neg\varphi_2) \mid \sigma \in \mathcal{L}(\varphi)\} \subseteq \mathcal{L}(\varphi' \vee \neg\varphi_2)$. Therefore, it follows with Corollary 5.1 that the counterstrategy extension of $\mathcal{T}_1^c \mid\mid \mathcal{T}_2^c$ is a deterministic and complete finite-state counterstrategy transducer for φ. □

From these two observations formalized in Lemmas 5.11 and 5.12, it now follows immediately that rewriting a strict assume-guarantee formula into conjunctive form and then dropping assumptions for the individual conjuncts according to the assumption-dropping criterion ensures that the initial formula φ is equirealizable to the constructed subformulas.

Corollary 5.3. *Let I and O be finite sets of input and output variables with $I \cap O = \emptyset$ and let $V = I \cup O$. Let $\varphi = (\varphi_1 \wedge \varphi_2 \wedge \varphi_3) \rightarrow (\psi_1 \wedge \psi_2)$ be an LTL formula over atomic propositions V. Suppose that φ_1 qualifies for dropping for $(\varphi_1 \wedge \varphi_2 \wedge \varphi_3) \rightarrow \psi_2$ and that φ_2 qualifies for dropping for $(\varphi_1 \wedge \varphi_2 \wedge \varphi_3) \rightarrow \psi_1$. Let $\varphi' = (\varphi_1 \wedge \varphi_3) \rightarrow \psi_1$ and $\varphi'' = (\varphi_2 \wedge \varphi_3) \rightarrow \psi_2$. If $prop(\varphi') \cap prop(\varphi'') \subseteq I$ holds, then φ is realizable if, and only if, both φ' and φ'' are realizable.*

Analyzing assumptions thus allows for decomposing LTL formulas into independent components in further cases. Nevertheless, it ensures the three requirements for soundness and completeness of modular synthesis. We incorporate assumption dropping into the search for independent conjuncts in the subsequent section.

5.4.2. LTL Decomposition with Assumption Dropping

In the following, we present an extension of the basic LTL decomposition algorithm presented in Section 5.3.2 that incorporates assumption dropping into the search for independent conjuncts. Note that the algorithm is only applicable to LTL formulas in *strict* assume-guarantee form, i.e., formulas of the form $\varphi = \bigwedge_{i=1}^{\ell} \varphi_j \rightarrow \bigwedge_{j=1}^{m} \psi_j$. In Section 5.5, we extend this algorithm to formulas consisting of several assume-guarantee conjuncts.

Recall that we can incorporate the dropping of assumptions into the basic LTL decomposition algorithm by first rewriting the strict assume-guarantee formula φ into conjunctive form. Note that the conjuncts of the rewritten formula are again in strict assume-guarantee form. Then, we drop as many assumptions as possible according to the assumption-dropping criterion for strict assume-guarantee formulas (see Definition 5.9) for the individual conjuncts if φ. Lastly, we decompose the resulting conjuncts according to the results on LTL independence from Section 5.3. By Lemmas 5.11 and 5.12 as well as Corollary 5.3, the three requirements for soundness and completeness of modular synthesis are then satisfied.

To incorporate the elimination of assumptions according to the assumption-dropping criterion for strict assume-guarantee LTL formulas into the decomposition algorithm, we first observe the following implications of the requirements of Lemmas 5.11 and 5.12 as well as Corollary 5.3. First, the two formulas φ' and φ'' resulting from dropping assumptions may only share input variables. Hence, it follows immediately that shared assumptions may only contain input variables. Consequently, assumptions that contain output variables may influence the decomposability of the specification since they cannot be shared between component specifications. We thus call output variables *decomposition-critical.*

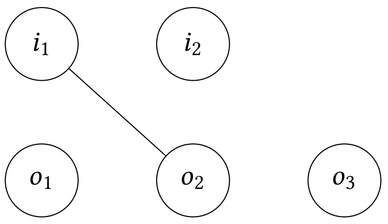
Second, recall that assumptions cannot be dropped if (i) they share *any* variables with a non-droppable assumption, (ii) they share *any* variables with the considered guarantees, or (iii) they can be violated by the system for all input sequences. Therefore, also input variables that occur in assumptions can prevent that assumptions can be dropped, and consequently, they can constitute a dependency between two output variables. We thus call input variables that are "connected" to output variables through assumptions *decomposition-critical* as well. The set of decomposition-critical variables of an LTL formula φ in strict assume-guarantee form is denoted with $V_\varphi^{crit} \subseteq I \cup O$. Note that both $V_\varphi^{crit} \subseteq I \cup O$ and $O \subseteq V_\varphi^{crit}$ hold by construction. We call assumptions that do not contain any decomposition-critical variables *free.*

Example 5.2. It is crucial to recognize that input variables may introduce dependencies between output variables in guarantee conjuncts via assumptions which may lead to non-separability of guarantee conjuncts. As an example, consider the LTL formula

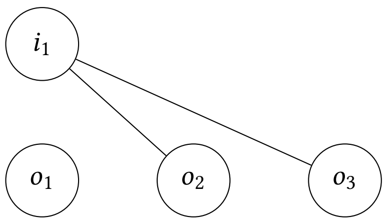
$$\varphi = (\square i_2 \land \square(o_2 \rightarrow \bigcirc \neg i_1)) \rightarrow (\square(i_2 \rightarrow o_1) \land \square(\neg o_2 \land i_2) \land \square(i_1 \rightarrow \neg o_3) \land \diamondsuit o_3)$$

in strict assume-guarantee form with inputs $I = \{i_1, i_2\}$ and outputs $O = \{o_1, o_2, o_3\}$. The formula is not realizable due to the last three guarantees: $\square(i_1 \rightarrow \neg o_3)$ and $\diamondsuit o_3$, are only realizable simultaneously if input variable i_1 is set to *false* at some point in time. The second assumption, i.e., $\square(o_2 \rightarrow \bigcirc \neg i_1)$, allows for satisfying the last two guarantees if o_2 is set to *true* eventually. Then, either i_1 is set to *false* in the next time step, satisfying the assumption and allowing for fulfilling the last two guarantees, or the assumption is violated and thus the formula is trivially satisfied. Yet, the third guarantee, i.e., $\square(\neg o_2 \land i_2)$, prevents this.

In particular, the last three guarantee conjuncts of φ thus cannot be separated as only their connection via the second assumption leads to unrealizability of the whole formula φ. This also matches the assumption-dropping criterion and, in particular, the prerequisites of Corollary 5.3. Since the last two guarantees of φ share output variables, they cannot be separated. Since the second assumption of φ cannot be dropped for either $\square(\neg o_2 \land o_2)$ or $\square(i_1 \rightarrow \neg o_3)$ as outlined above, the second guarantee of φ cannot be separated from the third and fourth one. Hence, the connection of these guarantees through the second assumption and, in particular, the connection of the third guarantee with this assumption via an input variable needs to be taken into account during decomposition of the initial specification φ Input i_1 is thus decomposition-critical and therefore assumption $\square(o_2 \rightarrow \bigcirc \neg i_1)$ is not free. △



(a) Assumption dependency graph $\mathcal{D}_\varphi^A$.



(b) Assume-guarantee dependency graph $\mathcal{D}_\varphi^{AG}$.

Figure 5.1.: Dependency graphs for the formula φ from Example 5.2.

Recall that we utilized a dependency graph in the basic LTL decomposition algorithm from Section 5.3.2 to determine which conjuncts share output variables. In a similar fashion, we determine decomposition-critical variables for LTL specifications in strict assume-guarantee form. We build the so-called *assumption dependency graph*, which is based on the assumptions of the formula – which are in conjunctive form – and not on the guarantees. Moreover, in contrast to the conjunct dependency graph for basic LTL decomposition algorithm (see Definition 5.8), *all* variables occurring in the formula serve as nodes of the graph, not only the output variables. This implements the change from shared output variables in the basic LTL decomposition algorithm to shared variables in general for decomposition-critical variables. An undirected edge between two variables in the assumption dependency graph denotes that variables occur in the same assumption. Formally:

Definition 5.10 (Assumption Dependency Graph)**.**
Let I and O be finite sets of input and output variables with $I \cap O = \emptyset$ and let $V = I \cup O$. Let $\varphi = \bigwedge_{i=1}^{\ell} \varphi_i \rightarrow \bigwedge_{j=1}^{m} \psi_j$ be an LTL formula over atomic propositions V. The *assumption dependency graph* $\mathcal{D}_\varphi^A = (\mathcal{V}, \mathcal{E})$ of φ is defined by $\mathcal{V} = V$ and $(a, b) \in \mathcal{E}$ if, and only if, $a \neq b$ and both $a \in \mathit{prop}(\varphi_j)$ and $b \in \mathit{prop}(\varphi_j)$ for some $1 \leq j \leq \ell$.

Hence, the assumption dependency graph captures which variables, both inputs and outputs, are shared in the assumptions of a strict assume-guarantee formula. If two variables are contained in the same connected component, then they either occur in the same assumption or they are connected via one or more assumptions. That is, intuitively, all assumptions that contain variables that lie in the same connected component of the assumption dependency graph cannot be separated in the sense that either none of them or all of them need to be dropped due to the first requirement of the assumption-dropping criterion, namely that an assumption can only be dropped if it does not share variables with any other assumption. Thus, all variables that lie in the same connected component as an output variable are decomposition-critical.

Example 5.3. Reconsider the LTL formula φ with $I = \{i_1, i_2\}$ and outputs $O = \{o_1, o_2, o_3\}$ from Example 5.2. The assumption dependency graph $\mathcal{D}_\varphi^A$ of φ is depicted in Figure 5.1a. It only considers the assumption of φ, i.e., $\square i_2$ and $\square(o_2 \rightarrow \neg i_1)$. Clearly, only the second assumption contributes a single edge in $\mathcal{D}_\varphi^A$. All output variables are decomposition-critical by definition. Moreover, since i_1 and o_2 lie in the same connected component of $\mathcal{D}_\varphi^A$, input variable i_1 is decomposition-critical as well. This meets our expectations that, as illustrated in Example 5.2, the connection of the second and third guarantee conjuncts of φ via i_1 and the second assumption is crucial for the unrealizability of φ. Input variable i_2, in contrast, does not share a connected component with an output variable and is thus not decomposition-critical, resulting in the set $V_\varphi^{crit} = \{i_1, o_1, o_2, o_3\}$ of decomposition-critical variables for the initial formula φ. △

The assumption dependency graph allows for determining non-separable assumptions and for identifying decomposition-critical variables. For computing valid components of the initial strict assume-guarantee LTL formula, we further need to take the guarantees into account. In particular, we need to identify non-droppable assumptions for concrete guarantees to be able to discover shared variables between conjuncts of $\bigwedge_{j=1}^{m} \left(\bigwedge_{i=1}^{\ell} \varphi_i \rightarrow \psi_j\right)$. The assumption dependency graph already establishes connections via shared variables between the assumptions. Hence, it remains to determine connections between guarantees and assumptions through shared variables. Again, we utilize a dependency graph, the so-called *assume-guarantee dependency graph.* It considers both assumptions and guarantees since it needs to establish connections between them to identify non-droppable assumptions. Moreover, it determines dependencies between guarantees together with their non-droppable assumptions to determine valid components, i.e., components with specifications that do not share output variables. Since assumptions are non-droppable if they share *any* variable with a guarantee, we cannot restrict the assume-guarantee dependency graph to output variables as is done for the dependency graph for basic LTL decomposition. Rather, it needs to take all decomposition-critical variables into account. Hence, the assume-guarantee dependency graph is built over all decomposition-critical variables, and there exists an edge between two variables if, and only if, they occur either in the same assumption or in the same guarantee. Formally:

Definition 5.11 (Assume-Guarantee Dependency Graph)**.**
Let I and O be finite sets of input and output variables with $I \cap O = \emptyset$ and let $V = I \cup O$. Let $\varphi = \bigwedge_{i=1}^{\ell} \varphi_i \rightarrow \bigwedge_{j=1}^{m} \psi_j$ be an LTL formula over atomic propositions V. The *assume-guarantee dependency graph* $\mathcal{D}_\varphi^{AG} = (\mathcal{V}, \mathcal{E})$ of φ is defined by $\mathcal{V} = V_\varphi^{crit}$ and $(a, b) \in \mathcal{E}$ if, and only if, $a \neq b$ and both $a \in prop(\xi)$ and $b \in prop(\xi')$ hold for some $\xi, \xi' \in \bigcup_{i=1}^{\ell} \varphi_i \cup \bigcup_{j=1}^{m} \psi_j$.

Note that the assume-guarantee dependency graph can be seen as an extension of the assumption dependency graph: we add edges induced by guarantees, and we delete input variables that are not decomposition-critical. Since the assume-guarantee dependency graph includes dependencies introduced by both assumptions and guarantees and utilizes both input and output variables, it allows for determining non-droppable assumptions for each guarantee as well as dependencies between guarantees together with their non-droppable assumptions. If an assumption and a guarantee contain variables that lie in the same connected component of the assume-guarantee dependency graph, then they are not separable in the sense that the assumption cannot be dropped for the guarantee according to the assumption-dropping criterion. If two guarantees contain variables that lie in the same connected component, then they share decomposition-critical variables. If they share output variables, their dependence is immediate. If they only share input variables, their dependence is more implicit. For both guarantees, assumptions that contain the shared input variable cannot be dropped. Since the input is decomposition-critical, it is connected to an output variable via assumptions. Since droppable assumptions may not share any variables with non-droppable assumptions, it thus follows that for both guarantees, assumptions containing the output variable cannot be dropped. Therefore, the component specifications for the guarantees share output variables.

Example 5.4. Reconsider the LTL formula

$$\varphi = (\Box i_2 \land \Box(o_2 \rightarrow \bigcirc \neg i_1)) \rightarrow (\Box(i_2 \rightarrow o_1) \land \Box(\neg o_2 \land i_2) \land \Box(i_1 \rightarrow \neg o_3) \land \Diamond o_3)$$

from Example 5.2. The assume-guarantee dependency graph $\mathcal{D}_\varphi^{AG}$ of φ is depicted in Figure 5.1b. It features the decomposition-critical variables V_φ^{crit} of φ as nodes. As described in Example 5.3, all output variables and input variable i_1 are decomposition-critical, while input i_2 is not. All assumption conjuncts as well as all guarantee conjuncts of φ may induce edges in $\mathcal{D}_\varphi^{AG}$. Since i_2 is not decomposition-critical, however, the first assumption conjunct as well as the first two guarantee conjuncts do not cause any edges. Moreover, since self-loops are not contained in the dependency graph by definition, the last guarantee conjunct does not add edges. Hence, $\mathcal{D}_\varphi^{AG}$ contains the two edges between i_1 and o_2 as well as between i_1 and o_3 induced by the remaining two conjuncts. This indicates that the outputs o_2 and o_3 depend on each other, and therefore conjuncts containing o_2 and o_3 cannot be separated. Thus, in fact, the last three guarantees of φ cannot be separated. This meets our expectations since otherwise the unrealizability of φ cannot be detected as illustrated in Example 5.2. $\triangle$

We can thus utilize the assume-guarantee dependency graph of a given LTL formula in strict assume-guarantee form to identify the decomposition of φ. Algorithm 5.3 describes how an LTL formula φ in strict assume-guarantee form with inputs I and outputs O is decomposed into independent components based on an analysis of the connected components of the assume-guarantee dependency graph. First, we separate assumptions and guarantees of φ (lines 1 and 2). Note that both the assumptions and guarantees are in conjunctive form. Thus we obtain a list of assumptions and guarantees, respectively, by simply removing the top-level conjunctions. We then compute the decomposition-critical propositions of φ (line 3). This is done by first building the assumption dependency graph $\mathcal{D}_\varphi^{A}$ of φ, then computing the connected components, and lastly identifying the propositions that lie in the same connected component as an output variable. Based on the decomposition-critical propositions, we build the assumption-guarantee dependency graph $\mathcal{D}_\varphi^{AG}$ of φ (line 4) and compute its connected components (line 5). We identify free assumptions, i.e., assumptions that do not contain decomposition-critical variables, (lines 9 to 11). Then, we refine the subspecifications with all other assumptions based on the connected components and the shared propositions similar to the refinement with conjuncts in Algorithm 5.2, i.e., the basic LTL decomposition algorithm (lines 13 to 16). Similarly, we add the guarantees to the corresponding subspecifications (lines 17 to 22). Afterward, we construct the components. For each subspecification, we add the necessary free assumptions (line 23). Note that all free assumptions could be safely added to all subspecifications. To obtain small subspecifications and thus also smaller synthesis subtasks, however, we only add those free assumptions to a subspecification that are needed, i.e., those free assumptions that

Algorithm 5.3: Decomposition Algorithm for Strict Assume-Guarantee Formulas

```
   Input: $\varphi$: LTL, I: List Variable, O: List Variable
   Output: components: List (LTL, List Variable, List Variable)
1  assumptions ← getAssumptions($\varphi$)
2  guarantees ← getGuarantees($\varphi$)
3  decCriticalProps ← getDecompositionCriticalPropositions($\varphi$)
4  agDependencyGraph ← buildAGDependencyGraph($\varphi$, decCriticalProps)
5  cc ← agDependencyGraph.connectedComponents()
6  subspecs ← [|cc|+1]: List LTL    // LTL list of length |cc|+1, initialized with true
7  freeAssumptions ← []
8  foreach $\varphi_i$ ∈ assumptions do
9      propositions ← getPropositions($\varphi_i$) ∩ decCriticalProps
10     if |propositions| = 0 then
11         freeAssumptions.append($\varphi_i$)
12     else
13         foreach (spec,vars) ∈ zip(subspecs, cc ++ [I]) do
14             if propositions ∩ vars ≠ ∅ then
15                 spec ← spec.addAssumption($\varphi_i$)
16                 break
17 foreach $\psi_i$ ∈ guarantees do
18     propositions ← getPropositions($\psi_i$) ∩ decCriticalProps
19     foreach (spec,vars) ∈ zip(subspecs, cc ++ [I]) do
20         if propositions ∩ vars ≠ ∅ then
21             spec ← spec.addGuarantee($\psi_i$)
22             break
23 subspecs.addNeededFreeAssumptions(freeAssumptions)
24 components ← buildComponents(subspecs)
25 return components
```

feature variables that occur in the subspecification. Then, we construct the components, i.e., the component specifications as well as the component interfaces (line 24) as in the basic LTL decomposition algorithm. Lastly, the components are returned (line 25).

Example 5.5. Reconsider the LTL formula

$$\varphi = (\Box i_2 \land \Box(o_2 \rightarrow \bigcirc \neg i_1)) \rightarrow (\Box(i_2 \rightarrow o_1) \land \Box(\neg o_2 \land i_2) \land \Box(i_1 \rightarrow \neg o_3) \land \Diamond o_3)$$

from Example 5.2 and the assume-guarantee dependency graph $\mathcal{D}_{\varphi}^{AG}$ depicted in Figure 5.1b and described in Example 5.4, respectively. Since $\mathcal{D}_{\varphi}^{AG}$ contains two connected components, namely $\{o_1\}$ and $\{i_1, o_2, o_3\}$, the Algorithm 5.3 yields two components c_1 and c_2, one for each of the connected components Since the assumption $\Box(o_2 \rightarrow \bigcirc \neg i_1)$ contains decomposition-critical variables, it is added to the components it shares variables with, i.e., in this case c_2. Similarly, the guarantees of φ are added to the components, resulting in $\Box(i_2 \rightarrow o_1)$ being assigned to c_1, while all three other guarantees are assigned to c_2. Since $\Box i_2$ is a free assumption, it is added to those components that share variables with the assumption. In this case, this is only c_1. All in all, we thus obtain the two components $c_1 = (\Box i_2 \rightarrow \Box(i_2 \rightarrow o_1), \{i_2\}, \{o_1\})$ and $c_2 = (\Box(o_2 \rightarrow \bigcirc\neg i_1) \rightarrow (\Box(\neg o_2 \wedge i_2) \wedge \Box(i_1 \rightarrow \neg o_3) \wedge \Diamond o_3), \{i_1\}, \{o_2, o_3\})$. △

Algorithm 5.3 then computes a decomposition of (I, O) that ensures that the resulting component specifications do not share output variables. Moreover, the algorithm ensures that assumptions that are not contained in a component specification neither share any variables with the guarantees nor with the assumptions of the component specification:

Lemma 5.13. *Let $\mathcal{A}$ be a monolithic architecture with input variables I and output variables O. Let $\varphi = \bigwedge_{i=1}^{\ell} \varphi_i \rightarrow \bigwedge_{j=1}^{m} \psi_j$ be an LTL formula over atomic propositions $I \cup O$. Algorithm 5.3 terminates with a decomposition $\mathbb{D}$ for input φ, I, O such that all $c_k \in \mathbb{D}$ are of the form $c_k = (\xi_k, I_k, O_k)$, where $\xi_k = \bigwedge_{\xi \in A_k} \xi \rightarrow \bigwedge_{\xi' \in G_k} \xi'$ with $A_k \subseteq \bigcup_{i=1}^{m} \varphi_i$ and $G_k \subseteq \bigcup_{j=1}^{n} \psi_j$. Moreover, we have*

1. *$prop(\xi_j) \cap prop(\xi_j) \subseteq I$ for all $c_j, c_k \in \mathbb{D}$ with $c_j \neq c_k$,*
2. *$prop(\varphi_i) \cap prop(\psi_j) = \emptyset$ for all $c_k \in \mathbb{D}$, all $\varphi_i \in \bigcup_{i=1}^{m} \varphi_i \setminus A_k$, and all $\psi_j \in G_k$, and*
3. *$prop(\varphi_i) \cap prop(\varphi_j) = \emptyset$ for all $c_k \in \mathbb{D}$, all $\varphi_i \in \bigcup_{i=1}^{m} \varphi_i \setminus A_k$, and all $\varphi_j \in A_k$.*

Proof. The form of the component specifications follows immediately from their construction. First, we show $prop(\xi_j) \cap prop(\xi_k) \subseteq I$ holds for all $c_j, c_k \in \mathbb{D}$. Let $o \in O$ be an output variable. By definition of the assume-guarantee dependency graph, o is contained in exactly one connected component. Furthermore, all conjuncts are assigned to the same subspecification in lines 13 to 16 and lines 17 to 22, respectively. Thus, o is part of the component specification of exactly one component of the decomposition $\mathbb{D}$. Hence, $prop(\xi_j) \cap prop(\xi_k) \subseteq I$ follows immediately for all components $c_j, c_k \in \mathbb{D}$ with $c_j \neq c_k$.

Next, we show that $prop(\varphi_i) \cap prop(\psi_j) = \emptyset$ holds for all $c_k \in \mathbb{D}$ as well as all $\varphi_i \in \bigcup_{i=1}^{m} \varphi_i \setminus A_k$ and all $\psi_j \in G_k$ holds. Let $v \in prop(\psi_j)$ be some variable that occurs in some guarantee ψ_j of c_k's component specification. If v is decomposition-critical, then it follows similar to the first case that v is contained in exactly one connected component of the assume-guarantee dependency graph and thus all assumptions featuring v are assigned to the same subspecification in lines 13 to 16 as ψ_j is assigned to in lines 17 to 22. Hence, $(prop(\varphi_i) \cap prop(\psi_j)) \cap V_{\varphi}^{crit} = \emptyset$ follows. If v is not decomposition-critical, however, then it is not represented by a node in the assume-guarantee dependency graph $\mathcal{D}_{\varphi}^{AG}$. By construction, v is an input variable. Moreover, v is not connected via shared variables in assumptions to a decomposition-critical variable. Hence, if $v \in prop(\varphi_i)$ holds, all variables in φ_i are not decomposition-critical and thus φ_i is free. Therefore, φ_i is added to the same subspecification as ψ_j is in line 23 and hence $\varphi_i \in A_k$ holds, yielding a contradiction. Thus, we have $v \notin prop(\varphi_i)$ and therefore, combining this result with the one for decomposition-critical variables, $(prop(\varphi_i) \cap prop(\psi_j)) = \emptyset$ follows.

Lastly, $prop(\varphi_i) \cap prop(\varphi_j) = \emptyset$ for all $c_k \in \mathbb{D}$ as well as all $\varphi_i \in \bigcup_{i=1}^{m} \varphi_i \setminus A_k$ and all $\varphi_j \in A_k$ follows analogously to the previous case since assumptions and guarantees are added to the subspecifications in the same fashion. Hence, considering an assumption $\varphi_j \in A_k$ instead of a guarantee $\psi_j \in G_k$ does not affect the proof. □

However, the decomposition computed by Algorithm 5.3 does not necessarily guarantee equirealizability with the initial specification since Algorithm 5.3 only preserves the first two requirements of the assumption-dropping criterion for strict assume-guarantee formulas (see Definition 5.9). It does not ensure that the dropped assumptions, i.e., assumptions that lie in $\bigcup_{j=1}^{m} \psi_j \setminus G_k$, cannot be violated by the system for all input sequences. As outlined in Section 5.4.1, however, unrealizability of the negation of the dropped assumption is a crucial requirement for assumption dropping. Therefore, we need to incorporate such a check into modular synthesis with the LTL decomposition algorithm with assumption dropping.

For this sake, observe that if the negation of *all* assumptions is unrealizable, then so are the negations of the individual assumptions. Then, there exists a counterstrategy transducer $\mathcal{T}^c$ that realizes the conjunction of all assumptions, i.e., it realizes $\bigwedge_{i=1}^{\ell} \varphi_i$. Using *transducer restriction*, we can construct counterstrategy transducers from $\mathcal{T}^c$ for the individual negated assumptions. Intuitively, the restriction of a transducer $\mathcal{T}$ to a subset $V_1 \subseteq V$ of the transducer's variables V is a copy of the transducer that ignores inputs outside of V_1 and restricts the outputs of a transition to the outputs of V_1. To obtain a deterministic transducer, it chooses one of the transitions of $\mathcal{T}$ that are not distinguishable when only considering V_1. Formally:

Definition 5.12 (Transducer Restriction)**.**
Let I, I_1 and O, O_1 be finite sets of input and output variables with $I_1 \subseteq I$ and $O_1 \subseteq O$. Let $L \subseteq (2^{I \cup O})^\omega$ be a realizable language and let $\mathcal{T}^c = (T, T_0, \tau, \ell)$ be a deterministic and complete finite-state $(2^I, 2^O)$-transducer realizing L. We construct a $(2^{I_1}, 2^{O_1})$-transducer $\mathcal{T}_1^c = (T_1, T_{1,0}, \tau_1, \ell_1)$ from $\mathcal{T}^c$ as follows:

- $T_1 = T$,
- $T_{1,0} = T_0$,
- $(t, \iota, t') \in \tau_1$ if, and only if, $(t, \iota \cup \iota', t') \in \tau$ holds, and
- $(t, \iota, o) \in \ell_1$ if, and only if, there exists some $o' \in 2^O$ with $o' \cap O_1 = i$ and $(t, \iota \cup \iota', o') \in \ell$.

where $\iota' = pick\left(2^{I \setminus I_1}\right)$ and where $pick(M)$ picks one element of the non-empty set M.

In contrast to transducer extension (see Lemma 5.1), the restriction of a transducer $\mathcal{T}$ realizing a language $L \in (2^V)^\omega$ to a subset V_1 of $\mathcal{T}$'s variables V then realizes a language $L_1 \subseteq (2^{V_1})^\omega$ if L_1 is stricter than L in the sense that every word satisfying L_1 also satisfies L:

Lemma 5.14. *Let I, I_1 and O, O_1 be finite sets of inputs and outputs with $I \cap O = \emptyset$, $I_1 \subseteq I$, and $O_1 \subseteq O$. Let $V = I \cup O$ and $V_1 = I_1 \cup O_1$. Let $L \subseteq (2^V)^\omega$ be a language. Let $L_1 \subseteq (2^{V_1})^\omega$ with $\{\sigma \cap V_1 \mid \sigma \in L\} \subseteq L_1$. Let L be realizable and let $\mathcal{T}$ be a transducer realizing L. The restricted transducer $\mathcal{T}_1$ constructed as in Definition 5.12 from $\mathcal{T}$ represents a strategy that realizes L_1.*

Proof. By construction, $\mathcal{T}_1$ is a $(2^{I_1}, 2^{O_1})$-transducer. Completeness of $\mathcal{T}_1$ and finiteness of $\mathcal{T}_1$'s set of states follows immediately from the construction of $\mathcal{T}_1$ and the fact that $\mathcal{T}$ is a complete finite-state transducer. Choosing only one valuation $\iota' \in 2^{I \setminus I_1}$ of input variables outside of I_1 ensures that, for all $t \in T_1$ and all $\iota \in 2^{I_1}$, we only choose one successor state and one labeling of those that are defined by $\mathcal{T}$ for inputs that are not distinguishable when only considering variables in V_1. Hence, determinism of $\mathcal{T}_1$ follows. Furthermore, $\mathcal{T}_1$ neither introduces additional transitions nor labelings with respect to $\mathcal{T}$ and thus the transducer type is preserved. Therefore, it remains to show that $\mathit{Traces}(\mathcal{T}_1) \subseteq L_1$ holds. Let $\sigma \in \mathit{Traces}(\mathcal{T}_1)$. By construction of $\mathcal{T}_1$ and by definition of traces, there exists some trace $\sigma' \in \mathit{Traces}(\mathcal{T})$ of $\mathcal{T}$ such that $\sigma' \cap V_1 = \sigma$ holds. By assumption, $\mathcal{T}$ realizes L. Hence, $\sigma' \in L$ holds. Since $\{\sigma \cap V_1 \mid \sigma \in L\} \subseteq L_1$ holds by assumption and since we have $\sigma' \cap V_1 = \sigma$, it thus follows that $\sigma \in L_1$ holds. Since we chose $\sigma \in \mathit{Traces}(\mathcal{T}_1)$ arbitrarily, $\mathit{Traces}(\mathcal{T}_1) \subseteq L_1$ follows and therefore $\mathcal{T}_1$ realizes L. □

Hence, if the negation of all assumptions is unrealizable, we can construct counterstrategy transducers for the negations of the individual assumptions from the counterstrategy transducer for the negation of all assumptions and consequently they are unrealizable as well. In particularly, all assumptions that are potentially dropped according to Lemma 5.13 are thus unrealizable. If the negation of all assumptions is realizable, in contrast, then the full formula φ is trivially realizable. The transducer realizing the negation of all assumptions can be extended with transducer extension to a transducer that, by the semantics of implication, satisfies the full formula by ignoring inputs outside the assumption propositions and choosing arbitrary valuations for outputs outside the assumption propositions. We call this *strategy extension*.

The above observations enable us to incorporate the check for unrealizability of the negation of dropped assumptions directly into the modular synthesis algorithm, resulting in a modified version of Algorithm 5.1 that is depicted in Algorithm 5.4. Before decomposition, we derive the negated assumptions from the initial specification φ (line 1) and the corresponding inputs and outputs (lines 2 and 3). We then perform synthesis for the negated assumptions (line 4). If the negated assumptions are realizable, then we extend the synthesized strategy to a strategy realizing the full specification φ with strategy extension (line 6) and return it (line 7). Otherwise, i.e., if the negated assumptions are unrealizable, then we proceed with the usual modular synthesis algorithm since then all individual negated assumptions are unrealizable as well. Note, however, that we employ a slightly modified counterstrategy extension in line 17 that, in addition to the counterstrategy for the unrealizable component specification, takes the counterstrategy for the negated assumptions into account: we derive the counterstrategy for the negated assumptions which have been dropped for the considered component specification using transducer restriction. Lemma 5.12 then allows for concluding that the counterstrategy extension of the parallel composition of $\mathcal{T}^c$ and the counterstrategy for the negated dropped assumptions is then a counterstrategy transducer for φ.

Utilizing the results of this section, in particular Lemmas 5.11 to 5.14 as well as Corollary 5.3 and strategy extension and restriction, it follows that the modular synthesis algorithm with assumption checking from Algorithm 5.4 is sound and complete when using the LTL decomposition algorithm with assumption dropping from Algorithm 5.3 as decomposition algorithm.

Theorem 5.3 (Soundness and Completeness). *Let $\mathcal{A}$ be a monolithic architecture with input variables I and output variables O. Let φ be an LTL formula over atomic propositions $I \cup O$. Suppose that Algorithm 5.4 utilizes Algorithm 5.3 as decomposition algorithm. If modular synthesis returns* $(\mathtt{true}, \mathcal{T})$ *on input φ, I, O, then φ is realizable and $\mathcal{T}$ realizes φ. If it returns* $(\mathtt{false}, \mathcal{T}^c)$*, then φ is unrealizable and $\mathcal{T}$ is a counterstrategy transducer of φ.*

Algorithm 5.4: Modular Synthesis with Assumption Check

```
    Input: $\varphi$: Specification, I: List Variable, O: List Variable
    Output: realizable: Bool, s: Strategy
 1  negatedAssumptions ← getNegatedAssumptions($\varphi$)
 2  aInp ← getPropositions(negatedAssumptions) ∩ I
 3  aOut ← getPropositions(negatedAssumptions) ∩ O
 4  (aRealizable, aStrategy) ← synthesize(negatedAssumptions, aInp, aOut)
 5  if aRealizable then
 6      strategy ← extendStrategy(aStrategy, I, O)
 7      return (true, strategy)
 8  components ← decompose($\varphi$, I, O)
 9  subResults ← []: List (Bool, Strategy)
10  subStrategies ← []: List Strategy
11  foreach (subspec, cInp, cOut) ∈ components do
12      (cRealizable, cStrategy) ← synthesize(subspec, cInp, cOut)
13      subResults.append((cRealizable, cStrategy))
14      subStrategies.append(cStrategy)
15  foreach (cRealizable, cStrategy) ∈ subResults do
16      if ! cRealizable then
17          counterstrategy ← extendCounterstrategy(cStrategy, aStrategy, I, O)
18          return (false, counterstrategy)
19  strategy ← compose(subStrategies)
20  return (true, strategy)
```

Proof. First, suppose that modular synthesis returns (true, $\mathcal{T}$) on input φ, I, O. Then, either the negated assumptions are realizable or none of the component specifications of the components in the decomposition $\mathbb{D}$ is unrealizable as otherwise neither the return statement in line 7 of Algorithm 5.4, nor the one in line 20 is reached. First, suppose that the former is the case. Let φ' capture the assumptions of φ and let I_a and O_a denote the input and output variables of the assumptions φ', respectively. Then, there exists deterministic and complete finite-state $(2^{I_a}, 2^{O_a})$-transducer that realizes $\neg\varphi'$. We extend $\mathcal{T}_a$ to a $(2^I, 2^O)$-transducer $\mathcal{T}$ with strategy extension. Since φ' captures the assumptions of φ, we have $\mathcal{L}(\neg\varphi') \subseteq \{\sigma \cap V_1 \mid \sigma \in \mathcal{L}(\varphi)\}$ by the semantics of implication. Thus, $\mathcal{T}$ realizes $\mathcal{L}(\varphi)$ by Lemma 5.1. Next, suppose that the negated assumptions are unrealizable and that all component specifications of the components in $\mathbb{D}$ are realizable. Then, there exists a counterstrategy transducer $\mathcal{T}_a^c$ for all negated assumptions. Hence, $\mathcal{T}_a^c$ realizes φ'. Since every word satisfying φ' also satisfies all individual conjuncts φ'_i of φ' by the semantics of conjunction, the transducer restriction of $\mathcal{T}_a^c$ is a counterstrategy transducer for φ'_i by Lemma 5.14 and hence $\neg\varphi'_i$ is unrealizable. Thus, by Lemma 5.13, (i) the component specifications do not share output variables and (ii) for each component specification ξ_i, the assumptions of φ that are not contained in ξ_i are droppable according to the assumption-dropping criterion. Therefore, it follows by recursively applying

Corollary 5.3 that φ is realizable as well and, by recursively applying Lemma 5.11, that the parallel composition of all substrategies realizes φ. Thus, $\mathcal{T}$ realizes φ by construction of $\mathcal{T}$.

Second, suppose that modular synthesis returns (`false`, $\mathcal{T}^c$) on input φ, I, O. Then, there exists a component $c_i \in \mathbb{D}$ with unrealizable component specification ξ_i. Moreover, the negated assumptions are unrealizable as otherwise the return statement in line 18 of Algorithm 5.4 is not reached. As outlined in the second part of the first case, we can construct a counterstrategy transducer for the negated assumptions of φ that are not contained in ξ_i from the counterstrategy transducer for the negation of all assumptions with counterstrategy restriction. Hence, the negation of the dropped assumptions for ξ_i is unrealizable. Thus, by Lemma 5.13, the component specifications do not share output variables and, for each component specification ξ_i, the assumptions of φ that are not contained in ξ_i are droppable. Therefore, it follows by recursively applying Corollary 5.3 that φ is unrealizable as well. Algorithm 5.4 extends the parallel composition of counterstrategy transducer for ξ_i and the counterstrategy transducer for the dropped assumptions of ξ_i, which it obtains from the counterstrategy for all negated assumptions with transducer restriction, according to Definition 5.3 in line 17 to a transducer $\mathcal{T}$ and returns it. Thus, $\mathcal{T}$ is a counterstrategy transducer for φ by Lemma 5.12. □

Analyzing whether assumptions can be safely removed during decomposition of LTL formulas thus allows for finding more fine-grained decompositions. Nevertheless, the resulting LTL decomposition algorithm presented in this section ensures soundness and completeness of modular synthesis. However, the assumption-dropping criterion requires that the specification has a top-level implication. Hence, the results cannot be applied to LTL formulas that are not in *strict* assume-guarantee form. In the next section, we thus extend LTL decomposition with assumption dropping to formulas consisting of several assume-guarantee-style conjuncts.

5.5. Non-Strict Assumption Dropping

Basic LTL decomposition as described in Section 5.3 is applicable to specifications consisting of arbitrarily many conjuncts. Lemma 5.7, which establishes that conjuncts form sublanguages, requires a top-level conjunction in the considered LTL formula. Hence, it can be applied recursively to specifications consisting of more than two conjuncts. Moreover, neither Lemma 5.8, which states that the languages of LTL formulas that do not share output variables are noncontradictory, nor the language-based independence criterion from Section 5.2 restricts the structure of the specification. Thus they can be applied recursively as well. In particular, basic LTL decomposition is thus applicable to specifications with several assume-guarantee conjuncts, i.e., LTL formulas of the form $\varphi = (\varphi_1 \rightarrow \psi_1) \land \ldots \land (\varphi_k \rightarrow \psi_m)$. LTL decomposition with assumption dropping, in contrast, is restricted to LTL specifications consisting of a single assume-guarantee pair. The assumption-dropping criterion formalized in Definition 5.9 and particularly the results in Lemmas 5.11 and 5.12 as well as Corollary 5.3, on which LTL decomposition with assumption dropping as described in Section 5.4.2, relies, assumes a top-level implication in the specification. Therefore, we cannot apply the results to specifications that are not in strict assume-guarantee form, and thus, in particular, we cannot apply modular synthesis with LTL decomposition with assumption dropping as realized by Algorithms 5.3 and 5.4 to LTL specifications that consist of several assume-guarantee conjuncts, i.e., several conjuncts that are possibly in assume-guarantee form. In this section, we thus study how to extend the basic LTL decomposition algorithm with assumption dropping also for assume-guarantee specifications in non-strict form, i.e., LTL formulas consisting of several assume-guarantee conjuncts.

A naïve approach for this extension is to first consider all assume-guarantee conjuncts of the specification separately and to eliminate unnecessary assumptions according to the assumption-dropping criterion stated in Definition 5.9 before then decomposing the conjunction of the resulting specifications using basic LTL decomposition, i.e., using Algorithm 5.2. In general, however, this is not sound as the other assume-guarantee conjuncts of the specification may introduce dependencies between assumptions and guarantees of the considered conjunct that prevent that the assumption can be dropped. When considering the conjuncts during the assumption-dropping phase separately, such dependencies are not detected. As an example, consider, the LTL formula

$$\varphi = \square\neg(o_1 \land o_2) \land \square\neg(i_1 \leftrightarrow o_1) \land ((\square i_1) \rightarrow \square o_2)$$

with input variables $I = \{i_1\}$ and output variables $O = \{o_1, o_2\}$. Note that the first two conjuncts can be seen as assume-guarantee conjuncts with no assumptions. Clearly, φ is realizable by a strategy that sets o_1 to the same truth value as $\neg i_1$ and o_2 to the same truth value as i_1 at every point in time. Since the first conjunct $\square\neg(o_1 \land o_2)$ of φ contains both output variables, φ is not decomposable according to the basic LTL decomposition algorithm since LTL component specifications are required to share input variables only. Thus, Algorithm 5.2 returns only a single component, namely $c = (\varphi, I, O)$. The naïve approach for incorporating assumption dropping into LTL decomposition also for non-strict assume-guarantee formulas described above, in contrast, considers the third conjunct $(\square i_1) \rightarrow \square o_2$ of φ separately and checks whether the single assumption $\square i_1$ can be dropped. Clearly, it can be dropped since it neither shares variables with $\square o_2$, nor is its negation realizable by the system. Thus, the assumption dropping phase results in the formula $\varphi' = \square\neg(o_1 \land o_2) \land \square\neg(i_1 \leftrightarrow o_1) \land \square o_2$. However, φ' is not realizable. If input variable i_1 is constantly set to *false*, then the second conjunct $\square\neg(i_1 \leftrightarrow o_1)$ enforces o_1 to be set to *true* at every point in time. The third conjunct $\square o_2$ enforces that o_2 is constantly set to *true*, irrespective of the input i_1. The first conjunct $\square\neg(o_1 \land o_2)$, however, requires one of the output variables to be *false* in every time step. Thus, although the assumption $\square i_1$ can be dropped when considering $(\square i_1) \rightarrow \square o_2$ in isolation, it cannot be dropped in the context of the other two conjuncts since the realizability of the full formula is not preserved. In particular, the first conjunct $\square\neg(o_1 \land o_2)$ introduces a dependency between o_1 and o_2 while the second conjunct $\square\neg(i_1 \leftrightarrow o_1)$ causes a dependency between i_1 and o_1. Therefore, there is a transitive dependency between i_1 and o_2 due to which the assumption $\square i_1$ cannot be dropped for the third conjunct $(\square i_1) \rightarrow \square o_2$. This dependency is not detected when considering the conjuncts separately during the assumption-dropping phase.

In the following, we thus introduce a more sophisticated extension of the LTL decomposition algorithm with assumption dropping to LTL formulas in non-strict assume-guarantee form, i.e., LTL formulas with several conjuncts which are possibly in assume-guarantee form, which is, in contrast to the naïve approach described above, sound. Similar to the naïve approach, the main idea is to first check for assumptions that can be dropped in the different conjuncts and to then perform basic LTL decomposition as described in Algorithm 5.2. However, the assumption-dropping phase is not performed entirely separately for the individual conjuncts but takes the other conjuncts into account. This ensures that possible transitive dependencies between the assumptions and guarantees are detected.

First, we lift the results from Section 5.4.1 to the case where further conjuncts are present. Afterward, we describe an LTL decomposition algorithm that incorporates assumption dropping also for formulas consisting of several assume-guarantee conjuncts.

5.5.1. Assumption Dropping in the Presence of Conjuncts

A crucial first step for developing an extension of the LTL decomposition algorithm with assumption dropping to LTL formulas in non-strict assume-guarantee form is to analyze when assumptions can be dropped in the presence of other conjuncts. Recall that the assumption-dropping criterion for strict assume-guarantee formulas requires droppable assumptions (i) not to share any variables with other guarantees, (ii) not to share any variables with the guarantees, and (iii) to have an unrealizable negation, ensuring that the system cannot violate the assumption on its own. Clearly, all these requirements must be satisfied in the presence of other conjuncts as well. Furthermore, to take transitive dependencies into account, we require an assumption not to share any variables with the other conjuncts in order to qualify for dropping. This results in the following criterion for dropping assumptions in non-strict formulas.

Definition 5.13 (Assumption-Dropping Criterion for Non-Strict Formulas).
Let I and O be finite sets of input and output variables, respectively, with $I \cap O = \emptyset$. Let $\varphi = \psi' \wedge ((\varphi_1 \wedge \varphi_2) \rightarrow \psi)$ be an LTL formula over atomic propositions $I \cup O$. Assumption φ_2 *qualifies for non-strict dropping* if, and only if,

1. $\mathit{prop}(\varphi_1) \cap \mathit{prop}(\varphi_2) = \emptyset$,
2. $\mathit{prop}(\varphi_2) \cap \mathit{prop}(\psi') = \emptyset$,
3. $\mathit{prop}(\varphi_2) \cap \mathit{prop}(\psi) = \emptyset$, and
4. $\neg\varphi_2$ is unrealizable.

Intuitively, this criterion allows for safely dropping assumptions since preventing the shared variables between the droppable assumption and the other conjunct intercepts any dependencies between the other conjunct and the assumption. Thus, in particular, no transitive dependencies can be introduced by the other conjunct. That is, if an assumption qualifies for non-strict dropping, then the original specification and the one obtained from dropping the assumption are equirealizable:

Lemma 5.15. *Let I and O be finite sets of inputs and outputs with $I \cap O = \emptyset$ and let $V = I \cup O$. Let $\varphi = \psi' \wedge ((\varphi_1 \wedge \varphi_2) \rightarrow \psi)$ be an LTL formula over atomic propositions V. If φ_2 qualifies for non-strict dropping for φ, then $\psi' \wedge (\varphi_1 \rightarrow \psi)$ is realizable if, and only if, φ is realizable.*

Proof. Let $V_1 := prop(\varphi_1) \cup prop(\psi') \cup prop(\psi)$ and $V_2 := prop(\varphi_2)$. Let $I_1 := V_1 \cap I$, $I_2 := V_2 \cap I$, $O_1 := V_1 \cap O$, and $O_2 := V_2 \cap O$. Let $\varphi' := \psi' \land (\varphi_1 \rightarrow \psi)$. First, suppose that φ' is realizable. Then, there exists a deterministic and complete finite-state $(2^{I_1}, 2^{O_1})$-transducer $\mathcal{T}_1$ realizing φ'. Let $\mathcal{T}$ be the finite-state $(2^I, 2^O)$-transducer obtained by extending $\mathcal{T}_1$ with transducer extension according to Definition 5.3. As in the corresponding proof of the non-strict case, i.e., as in the proof of Lemma 5.10, it follows from the construction of φ' and V_1 that $\mathcal{L}(\varphi') \subseteq \{\sigma \cap V_1 \mid \sigma \in \mathcal{L}(\varphi)\}$ holds. Hence, by Lemma 5.1, the transducer $\mathcal{T}$ realizes φ. Consequently, φ is realizable.

Next, suppose that φ' is unrealizable. Then, there exists a deterministic and complete finite-state counterstrategy $(2^{O_1}, 2^{I_1})$-transducer $\mathcal{T}_1^c$ for φ'. Since $\neg\varphi_2$ is unrealizable by assumption, there exists a counterstrategy $(2^{O_2}, 2^{I_2})$-transducer $\mathcal{T}_2^c$ for $\neg\varphi_2$. Since φ_2 qualifies for dropping for φ, we have $prop(\varphi_1) \cap prop(\varphi_2) = \emptyset$, $prop(\varphi_2) \cap prop(\psi') = \emptyset$, and $prop(\varphi_2) \cap prop(\psi) = \emptyset$ by definition of the non-strict assumption-dropping criterion. Thus, $V_1 \cap V_2 = \emptyset$ holds. Note that, in contrast to the strict case considered in the proof of Lemma 5.10, the requirement $prop(\varphi_2) \cap prop(\psi') = \emptyset$ is crucial to obtain $V_1 \cap V_2 = \emptyset$ since φ' contains ψ' in the non-strict case as well. Similar to the proof of Lemma 5.10, we can now conclude that $\mathcal{T}_1^c \mid\mid \mathcal{T}_2^c$ is a counterstrategy transducer for φ. Hence, φ is unrealizable. □

Similar to assumption dropping for LTL formulas in strict assume-guarantee form described in the previous section, we utilize Lemma 5.15 for decomposing LTL formulas with several assume-guarantee conjuncts while employing assumption dropping where possible. We rewrite an LTL formula $\varphi = \psi' \land \bigwedge_{i=1}^{\ell} \varphi_i \rightarrow \bigwedge_{j=1}^{m} \psi_j$ into the conjunctive form $\psi' \land \bigwedge_{j=1}^{m}(\bigwedge_{i=1}^{\ell} \varphi_i \rightarrow \psi_j)$ and then drop assumptions for the individual guarantees $\psi_1, \dots, \psi_m$ according to the assumption-dropping criterion for non-strict formulas described in Definition 5.13. If the resulting conjuncts only share input variables and are realizable, then the parallel composition of transducers realizing them realizes the full formula φ:

Lemma 5.16. *Let I and O be finite sets of input and output variables with $I \cap O = \emptyset$. Let $\varphi = \psi_1' \land \psi_2' \land (\varphi_1 \land \varphi_2 \land \varphi_3) \rightarrow (\psi_1 \land \psi_2)$ be an LTL formula over atomic propositions $I \cup O$. Let φ_1 qualify for non-strict dropping for $\psi_1' \land \psi_2' \land (\varphi_1 \land \varphi_2 \land \varphi_3) \rightarrow \psi_2$ and φ_2 for non-strict dropping for $\psi_1' \land \psi_2' \land (\varphi_1 \land \varphi_2 \land \varphi_3) \rightarrow \psi_1$. Let $\varphi' = \psi_1' \land ((\varphi_1 \land \varphi_3) \rightarrow \psi_1)$ and $\varphi'' = \psi_2' \land ((\varphi_2 \land \varphi_3) \rightarrow \psi_2)$. Suppose that $prop(\varphi') \cap prop(\varphi'') \subseteq I$ holds. If both φ' and φ'' are realizable, then the parallel composition $\mathcal{T}_1 \mid\mid \mathcal{T}_2$ of transducers $\mathcal{T}_1$ and $\mathcal{T}_2$ realizing φ' and φ'', respectively, realizes φ.*

Recall that the proof of the analogous lemma for strict formulas, i.e., the proof of Lemma 5.11, relies on two properties: (i) $prop(\varphi') \cap prop(\varphi'') = \emptyset$ needs to hold to conclude that $\mathcal{L}(\varphi')$ and $\mathcal{L}(\varphi'')$ are independent sublanguages of $\mathcal{L}(\varphi' \land \varphi'')$ which allows for the construction of a transducer $\mathcal{T}$ realizing $\varphi' \land \varphi''$ and (ii) $\mathcal{L}(\varphi' \land \varphi'') \subseteq \mathcal{L}(\varphi)$ needs to hold to conclude that $\mathcal{T}$ also realizes φ. Both these requirements are satisfied in the non-strict case as well by construction of φ' and φ'' as well as by the assumption on the shared variables of φ' and φ''. Thus, the proof of Lemma 5.16 is analogous to the proof of Lemma 5.11.

Vice versa, we can extend a counterstrategy transducer for one of the formulas resulting from non-strict assumption dropping for the individual conjuncts of $\psi' \bigwedge_{j=1}^{m} \left(\bigwedge_{i=1}^{\ell} \varphi_i \rightarrow \psi_j\right)$ to a counterstrategy transducer for the full formula $\psi' \land \bigwedge_{i=1}^{\ell} \varphi_i \rightarrow \bigwedge_{j=1}^{m} \psi_j$ analogously to the case where no other conjuncts are present, i.e., analogously to Lemma 5.12:

Lemma 5.17. *Let I and O be finite sets of input and output variables with $I \cap O = \emptyset$. Let $\varphi = \psi_1' \land \psi_2' \land (\varphi_1 \land \varphi_2 \land \varphi_3) \rightarrow (\psi_1 \land \psi_2)$ be an LTL formula over atomic propositions $I \cup O$. Suppose that φ_2 qualifies for non-strict dropping for $\psi_1' \land \psi_2' \land (\varphi_1 \land \varphi_2 \land \varphi_3) \rightarrow \psi_2$. Let $\mathcal{T}_2^c$ be a counterstrategy transducer for $\neg\varphi_2$. Let $\varphi' = \psi_1' \land ((\varphi_1 \land \varphi_3) \rightarrow \psi_1)$. If φ' is unrealizable, then the counterstrategy extension of $\mathcal{T}_1^c \mid\mid \mathcal{T}_2^c$, where $\mathcal{T}_1^c$ is a counterstrategy transducer for φ', is a counterstrategy transducer for φ.*

The proof of the analogous lemma for strict formulas and strict assumption dropping, i.e., the proof of Lemma 5.12, again relies on two properties. First, $prop(\varphi_2) \cap prop(\varphi') = \emptyset$ needs to hold to conclude that $\mathcal{L}(\varphi_2)$ and $\mathcal{L}(\neg\varphi')$ are independent sublanguages of $\mathcal{L}(\varphi_2 \land \neg\varphi')$, which then allows for concluding that the parallel composition of counterstrategy transducers for $\neg\varphi_2$ and φ' realizes $\varphi_2 \land \neg\varphi'$. Second, every sequence satisfying φ also needs to satisfy $\neg\varphi_2 \lor \varphi'$ in order to conclude that the counterstrategy extension of $\mathcal{T}_1^c \mid\mid \mathcal{T}_2^c$ is a counterstrategy transducer for φ. Both these requirements are satisfied in the non-strict case as well by construction of φ' as well as by the definition of the non-strict assumption-dropping criterion. Thus, the proof of Lemma 5.17 is analogous to the proof of Lemma 5.12.

From these two observations formalized in Lemmas 5.16 and 5.17, it now follows immediately that rewriting a non-strict assume-guarantee formula into conjunctive form and then dropping assumptions for the individual conjuncts according to the non-strict assumption-dropping criterion preserves equirealizability:

Corollary 5.4. *Let I and O be finite sets of input and output variables with $I \cap O = \emptyset$. Let $\varphi = \psi_1' \land \psi_2' \land (\varphi_1 \land \varphi_2 \land \varphi_3) \rightarrow (\psi_1 \land \psi_2)$ be an LTL formula over atomic propositions $I \cup O$. Let φ_1 qualify for non-strict dropping for $\psi_1' \land \psi_2' \land (\varphi_1 \land \varphi_2 \land \varphi_3) \rightarrow \psi_2$ and φ_2 for non-strict dropping for $\psi_1' \land \psi_2' \land (\varphi_1 \land \varphi_2 \land \varphi_3) \rightarrow \psi_1$. Let $\varphi' = \psi_1' \land ((\varphi_1 \land \varphi_3) \rightarrow \psi_1)$ and $\varphi'' = \psi_2' \land ((\varphi_2 \land \varphi_3) \rightarrow \psi_2)$. If $prop(\varphi') \cap prop(\varphi'') \subseteq I$ holds, then φ is realizable if, and only if, both φ' and φ'' are realizable.*

Taking all other conjuncts into account thus allows for dropping assumptions of assume-guarantee conjuncts also in LTL specifications that are in non-strict assume-guarantee form while still preserving equirealizability. Thus, it enables LTL decomposition in further cases for non-strict formulas than basic LTL decomposition. Nevertheless, it ensures soundness and completeness of modular synthesis. We incorporate non-strict assumption dropping into the search for independent conjuncts in the subsequent section.

5.5.2. Non-Strict LTL Decomposition Algorithm

In the following, we extend the LTL decomposition algorithm with strict assumption dropping from Section 5.4.2 to LTL specifications that do not follow a strict assume-guarantee form but consist of multiple assume-guarantee conjuncts. The centerpiece of the algorithm is the bottom-up construction of components similar to the LTL decomposition with assumption dropping for strict assume-guarantee formulas (see Algorithm 5.3). However, it utilizes the non-strict version of the assumption-dropping criterion instead of the string one. This construction is formalized in Algorithm 5.5. In contrast to both other LTL decomposition algorithms presented in this chapter, it expects, in addition to the initial LTL specification φ and the input and output

Algorithm 5.5: Conjunct-based LTL Decomposition for Non-Strict Formulas

```
Input: $\varphi$: LTL, I: List Variable, O: List Variable, agConjunct: LTL
Output: components: List (LTL, List Variable, List Variable)
assumptions $\leftarrow$ getAssumptions(agConjunct)
guarantees $\leftarrow$ getGuarantees(agConjunct)
decCriticalProps $\leftarrow$ getDecompositionCriticalPropositions(agConjunct)
agDependencyGraph $\leftarrow$ buildAGDependencyGraph($\varphi$, decCriticalProps)
cc $\leftarrow$ agDependencyGraph.connectedComponents()
subspecs $\leftarrow$ [|cc|+1]: List LTL    // LTL list of length |cc|+1, initialized with true
freeAssumptions $\leftarrow$ []
foreach $\varphi_i \in$ assumptions do
    propositions $\leftarrow$ getPropositions($\varphi_i$) $\cap$ decCriticalProps
    if |propositions| = 0 then
        freeAssumptions.append($\varphi_i$)
    else
        foreach (spec,vars) $\in$ zip(subspecs, cc ++ [I]) do
            if propositions $\cap$ vars $\neq \emptyset$ then
                spec $\leftarrow$ spec.addAssumption($\varphi_i$)
                break
foreach $\psi_i \in$ guarantees do
    propositions $\leftarrow$ getPropositions($\psi_i$) $\cap$ decCriticalProps
    foreach (spec,vars) $\in$ zip(subspecs, cc ++ [I]) do
        if propositions $\cap$ vars $\neq \emptyset$ then
            spec $\leftarrow$ spec.addGuarantee($\psi_i$)
            break
foreach $\psi \in$ getConjuncts($\varphi \setminus$ agConjunct) do
    propositions $\leftarrow$ getPropositions($\psi$) $\cap$ decCriticalProps
    foreach (spec,vars) $\in$ zip(subspecs, cc ++ [I]) do
        if propositions $\cap$ vars $\neq \emptyset$ then
            spec $\leftarrow$ spec.addConjunct($\psi$)
            break
subspecs.addNeededFreeAssumptions(freeAssumptions)
components $\leftarrow$ buildComponents(subspecs)
return components
```

variables of the system, an assume-guarantee conjunct as input. This assume-guarantee formula is expected to be a conjunct of φ with a top-level implication. Taking this additional input into account is necessary due to the structure of the assumption-dropping criterion for non-strict assume-guarantee formulas, as it is based on the choice of one particular assume-guarantee conjunct. Hence, Algorithm 5.5 implements the construction of components based on one particular assume-guarantee conjunct of the formula.

Similar to the LTL decomposition algorithm with strict assumption dropping depicted in Algorithm 5.3, we employ two dependency graphs, the assumption dependency graph and the assume-guarantee dependency graph, to determine dependencies. Recall that the assumption dependency graph is used to detect decomposition-critical variables, i.e., variables that are connected to an output variable via assumptions. Since we only consider a single assume-guarantee conjunct in Algorithm 5.5, we only consider a single set of assumptions. Therefore, we build the assumption dependency graph and thus compute the decomposition-critical variables based on the assume-guarantee conjunct and not based on the full LTL formula (line 3). The assume-guarantee dependency graph, which is then used to determine independent components, in contrast, includes dependencies induced by *all* conjuncts of the LTL formula φ, not only the ones induced by the considered assume-guarantee conjunct (line 4). This is necessary since the non-strict assumption-dropping criterion forbids shared variables also between the assume-guarantee conjunct and the other conjuncts of φ. Intuitively, the remaining conjuncts of φ are thus treated similarly to the guarantees of the assume-guarantee conjunct. This similar treatment also carries over to constructing the component specifications. Assumptions and guarantees of the assume-guarantee conjunct are added to the respective component specification similar to Algorithm 5.3, i.e., they are added to the specification of the component defined by a connected component of the assume-guarantee dependency graph they share variables with (lines 8 to 22). Afterward, the remaining conjuncts of φ are added as additional conjuncts to the respective component specifications in a similar fashion as the guarantees of the assume-guarantee conjunct (lines 23 to 28). Lastly, the free assumptions are added (line 29) and the components are built from the resulting subspecifications (line 30).

Algorithm 5.5 then ensures that the resulting component specifications do not share output variables. Furthermore, for each component, the assumptions of the assume-guarantee conjunct that served as an input that are not present in the considered component's specification do not share any variables with both the assumptions and guarantees that are present in the assume-guarantee part of the component specification. Additionally, the non-present assumptions further do not share any variables with other parts of the component specification:

Lemma 5.18. *Let $\mathscr{A}$ be a monolithic architecture with input variables I and output variables O. Let $\varphi = \bigwedge_{\ell=1}^{k} \psi'_\ell \wedge (\bigwedge_{i=1}^{\ell} \varphi_i \rightarrow \bigwedge_{j=1}^{m} \psi_j)$ be an LTL formula over atomic propositions $I \cup O$. Let $\varphi' := \bigwedge_{i=1}^{\ell} \varphi_i \rightarrow \bigwedge_{j=1}^{m} \psi_j$. Algorithm 5.5 terminates with a decomposition $\mathbb{D}$ for input φ, I, O, φ' such that all $c_k \in \mathbb{D}$ are of the form $c_k = (\xi_k, I_k, O_k)$, where $\xi_k = \xi'' \wedge (\bigwedge_{\xi \in A_k} \xi \rightarrow \bigwedge_{\xi' \in G_k} \xi')$ with $A_k \subseteq \bigcup_{i=1}^{\ell} \varphi_i$ and $G_k \subseteq \bigcup_{j=1}^{m} \psi_j$. Moreover, we have*

1. *$prop(\xi_j) \cap prop(\xi_k) \subseteq I$ for all $c_j, c_k \in \mathbb{D}$ with $c_j \neq c_k$,*
2. *$prop(\varphi_i) \cap prop(\psi_j) = \emptyset$ for all $c_k \in \mathbb{D}$, all $\varphi_i \in \bigcup_{i=1}^{\ell} \varphi_i \setminus A_k$, and all $\psi_j \in G_k$,*
3. *$prop(\varphi_i) \cap prop(\varphi_j) = \emptyset$ for all $c_k \in \mathbb{D}$, all $\varphi_i \in \bigcup_{i=1}^{\ell} \varphi_i \setminus A_k$, and all $\varphi_j \in A_k$, and*
4. *$prop(\varphi_i) \cap prop(\xi'') = \emptyset$ for all $c_k \in \mathbb{D}$ and all $\varphi_i \in \bigcup_{i=1}^{\ell} \varphi_i \setminus A_k$*

Proof. The form of the component specifications follows immediately from their construction in the algorithm. Furthermore, properties (1) to (3) follow from Lemma 5.13 and the fact that Algorithm 5.5 does not differ from Algorithm 5.3 for the assume-guarantee conjunct φ' which

Algorithm 5.6: Decomposition Algorithm for Non-Strict Formulas

```
Input: $\varphi$: LTL, I: List Variable, O: List Variable
Output: components: List (LTL, List Variable, List Variable)
components ← [$\varphi$]
if $\varphi$.hasAGConjunct() then
    agConjuncts ← $\varphi$.getAllAGConjuncts()
    foreach conj ∈ agConjuncts do
        conjComponents ← []: List (LTL, List Variable, List Variable)
        foreach comp ∈ components do
            compSpec ← comp.getSpecification()
            if compSpec.containsConjunct(conj) then
                decomp ← nonStrictLTLDecomposition(compSpec, I, O, conj)
                conjComponents ← conjComponents ++ decomp
        components ← conjComponents
return components
```

is provided as input to Algorithm 5.5 since it is in strict assume-guarantee form by definition. Hence, it only remains to show that $prop(\varphi_i) \cap prop(\xi'') = \emptyset$ holds for all $c_k \in \mathbb{D}$ as well as all $\varphi_i \in \bigcup_{i=1}^{\ell} \varphi_i \setminus A_k$. The proof is analogous to the proof of property (2) of Lemma 5.13 since the remaining conjuncts are added to the component specifications in lines 23 to 28 in Algorithm 5.5 exactly as the guarantees in lines 17 to 22 in Algorithm 5.3. □

The decomposition returned by Algorithm 5.5 and, in fact, the applicability of Lemma 5.16 as well as Corollary 5.4, heavily relies on the considered assume-guarantee conjunct. Consequently, to obtain small synthesis subtasks, we apply Algorithm 5.5 for *all* choices of assume-guarantee conjuncts of the formula as otherwise possible decompositions might be missed. Therefore, after decomposing an LTL specification with non-strict assumption dropping, we reapply Algorithm 5.5 to the resulting components with the remaining assume-guarantee conjuncts. The resulting algorithm for LTL decomposition with non-strict assumption dropping is depicted in Algorithm 5.6 Note that Algorithm 5.5 only needs to be reapplied to components that indeed contain the considered assume-guarantee conjunct.

Reapplying Algorithm 5.5 allows for finding all decompositions that are possible according to the non-strict assumptions dropping criterion and thus according to Corollary 5.4. However, decompositions that can even be detected by the basic LTL decomposition algorithm which does not take assumption dropping into account might still be missed. As an example, reconsider the LTL formula $\varphi = (\Box i_2 \wedge \Box(o_2 \rightarrow \bigcirc \neg i_1)) \rightarrow (\Box(i_2 \rightarrow o_1) \wedge \Box(\neg o_2 \wedge i_2) \wedge \Box(i_1 \rightarrow \neg o_3) \wedge \Diamond o_3)$ in strict assume-guarantee form with inputs $I_\varphi = \{i_1, i_2\}$ and outputs $O_\varphi = \{o_1, o_2, o_3\}$ from Example 5.2. Recall that φ can be decomposed into two components as described in Example 5.5. Based on φ, we construct an LTL formula $\psi = \varphi \wedge \Box(i_1 \rightarrow \Diamond o_4)$ in non-strict assume-guarantee form with input variables $I_\psi = \{i_1, i_2\}$ and output variables $O_\psi = \{o_1, o_2, o_3, o_4\}$. When applying Algorithm 5.5 to ψ with the only assume-guarantee conjunct φ, then ψ is decomposed into two components. Since i_1 is decomposition-critical in φ (see Example 5.2), there exists an edge between i_1 and o_4 in the assume-guarantee graph due to the conjunct $\square(i_1 \rightarrow \diamondsuit o_4)$ as well as edges between i_1 and both o_2 and o_3 due to φ. Hence, Algorithm 5.5 concludes that output o_1 can be separated from the other three outputs, resulting in the two components. However, the two conjuncts of ψ only share the input variable i_1 and are thus separated by the basic LTL decomposition algorithm into two components $c_1 = (\varphi, I_\varphi, O_\varphi)$ and $c_2 = (\square(i_1 \rightarrow o_1), \{i_1\}, \{o_4\})$ for which separate synthesis tasks can be performed. Then, we can further decompose c_1 with Algorithm 5.5, resulting in a total of three components. Hence, we miss an independent component when applying Algorithm 5.5 to ψ directly.

Missing decompositions is not a problem introduced by the algorithm but already occurs when simply applying the non-strict assumption-dropping criterion. To overcome this weakness, modular synthesis performs decomposition in two steps. First, it decomposes the LTL formula with the basic decomposition algorithm from Algorithm 5.2. Afterward, it applies LTL decomposition with non-strict assumption dropping as depicted in Algorithm 5.6.

The decomposition computed by Algorithm 5.6, however, does not necessarily guarantee equirealizability of the component specification with the initial specification. Similar to the LTL decomposition algorithm for strict assume-guarantee formulas, Algorithm 5.6 only preserves the first three requirements of the non-strict assumption-dropping criterion. It does not ensure that the dropped assumptions cannot be violated by the system for all input sequences, i.e., that the negation of the dropped assumptions is unrealizable for the system. Therefore, we again need to incorporate such a check into modular synthesis when using the LTL decomposition algorithm with non-strict assumption dropping.

Recall that Algorithm 5.4 is a slightly modified version of modular synthesis that first tries to synthesize a strategy for the negation of the assumptions of the given strict assume-guarantee formula. If synthesis succeeds, we extend the resulting strategy to the full system since we have found a strategy that violates the assumptions for all environment inputs and thus trivially realizes the initial specification. When considering LTL formulas in non-strict assume-guarantee form, however, extending a strategy that realizes the negation of the assumptions of one of the assume-guarantee conjuncts might not yield a strategy that realizes the initial specification. This strategy is guaranteed to trivially realize one assume-guarantee conjunct of the formula, yet, it might violate other conjuncts. Therefore, we employ a different assumption check in modular synthesis when considering LTL decomposition with non-strict assumption dropping: if the negation of the assumptions of some assume-guarantee conjunct of the initial specification is realizable, then the synthesized strategy is first extended as in Algorithm 5.4. Afterward, it is model checked against the remaining conjuncts. If the check succeeds, then the extended strategy is indeed a strategy for the initial specification and is returned. Otherwise, the strategy does not comply with some of the other conjuncts. Hence, it does not serve as a strategy for the full formula; therefore, we do not return it. Instead, we proceed with modular synthesis for the decompositions obtained with basic LTL decomposition only.

Utilizing the results of this section, in particular Lemma 5.18 and Corollary 5.4, it now follows that the modular synthesis algorithm with modified assumption checking as described above is sound and complete when using the LTL decomposition algorithm with non-strict assumption dropping from Algorithm 5.6 as decomposition algorithm.

Theorem 5.4 (Soundness and Completeness)**.** *Let $\mathcal{A}$ be a monolithic architecture with input variables I and output variables O. Let φ be an LTL formula over atomic propositions $I \cup O$. Suppose that modular synthesis with modified assumption check utilizes Algorithm 5.6 as decomposition algorithm. If modular synthesis returns* $(\texttt{true}, \mathcal{T})$ *on input φ, I, O, then φ is realizable and $\mathcal{T}$ realizes φ. If it returns* $(\texttt{false}, \mathcal{T}^c)$*, then $\mathcal{T}$ is a counterstrategy transducer of φ.*

Proof. First, suppose that modular synthesis returns $(\texttt{true}, \mathcal{T})$ on input φ, I, O. Then, either the negated assumptions of one of the conjuncts are realizable and the strategy realizing them can be successfully extended, or all of the component specifications are realizable. First, suppose that the former is the case. Since the extended strategy is verified against the initial specification and only returned if the verification is successful, it follows immediately that the returned strategy realizes φ. Next, suppose that all of the component specifications of the components in the decomposition $\mathbb{D}$ are realizable. If only basic LTL decomposition has been applied, then it follows exactly as in the proof of Lemma 5.9 that the decomposition is syntactically independent. Thus, $\mathcal{T}$ realizes φ by Corollary 5.2. If further non-strict assumption dropping has been applied, then it follows analogously to the proof of Theorem 5.3 but by employing the respective non-strict results Lemmas 5.16 and 5.18 as well as Corollary 5.4 that $\mathcal{T}$ realizes φ.

Second, suppose that modular synthesis returns $(\texttt{false}, \mathcal{T}^c)$ on input φ, I, O. Then, there exists a component $c_i \in \mathbb{D}$ with unrealizable component specification ξ_i. If only basic LTL decomposition has been applied, then it follows as in the proof of Lemma 5.9 that the decomposition is syntactically independent. Thus, by Corollary 5.2, the transducer $\mathcal{T}$ is a counterstrategy transducer for φ. If further non-strict assumption dropping has been applied, then it follows analogously to the proof of Theorem 5.3 but by employing the respective non-strict results Lemmas 5.17 and 5.18 that $\mathcal{T}$ is a counterstrategy transducer for φ. □

Utilizing Algorithms 5.5 and 5.6, we can thus incorporate assumption dropping also into the decomposition of LTL specifications that are not in strict assume-guarantee form. Modular synthesis in its slightly adapted form described above is nevertheless sound and complete. It thus constitutes a compositional synthesis approach for monolithic systems with LTL specifications based on a syntactic analysis of the specification.

5.6. Experimental Evaluation

We implemented the modular synthesis algorithm as well as the decomposition algorithm for LTL specifications in non-strict assume-guarantee form. The LTL decomposition relies on the tool SyFCo [JKS16] for formula transformations. We first decompose the LTL specification with the decomposition algorithm for LTL formulas in non-strict assume-guarantee form described in Section 5.5 and then run synthesis sequentially on the resulting subspecifications. Note that parallelization of the synthesis tasks may further reduce the running time. We evaluate modular synthesis with two state-of-the-art synthesis tools, BoSy [FFT17] and Strix [MSL18], both in their 2019 release. BoSy implements the bounded synthesis [FS13] approach a Strix implements a game-based synthesis approach, The experimental evaluation was performed on a 3.6GHz quad-core Intel Xeon processor and 32GB of RAM.

Table 5.1.: Distribution of the number of components for all 346 benchmarks for LTL decomposition with assumption dropping for formulas in non-strict assume-guarantee form.

# components	1	2	3	4	5	6	7	8	9	10	11	12
# benchmarks	307	19	8	2	3	2	0	2	0	1	1	1

We evaluate our approach with the well-established benchmarks of the annual reactive synthesis competition SyntComp [BEJ14, JBB+17b, JBB+15, JBB+16, JB16, JBB+17a, JBC+19, JPA+22]. In particular, we consider the 346 publicly available LTL benchmarks from SyntComp 2020 [SYN20]. Note that only 207 of these benchmarks have more than one output variable and are, therefore, realistic candidates for decomposition. The LTL decomposition algorithm for formulas in non-strict assume-guarantee format as described in Section 5.5.2 terminates on all benchmarks in less than 26 milliseconds. Hence, even for non-decomposable specifications, the overhead of trying to perform decompositions first is negligible. The decomposition algorithm decomposes 39 LTL formulas into several components. Most of the benchmarks yield two or three components, and only a handful of formulas are decomposed into more than six components. The full distribution of the number of resulting components for all benchmarks is shown in Table 5.1. When applying only basic LTL decomposition algorithm as described in Section 5.3.2, in contrast, only 24 of the benchmarks are decomposed into one or more components. By construction, all decompositions found with the basic LTL decomposition algorithm are also found by the LTL decomposition algorithm for non-strict assume-guarantee formulas. This shows that assumption dropping has a considerable impact on the performance of the decomposition algorithm.

For all decomposable SyntComp benchmarks, we compare the synthesis running times for non-compositional synthesis and modular synthesis with both BoSy and Strix. We used a time out of 60 minutes. The results are shown in Figure 5.2. The plot relates the accumulated running time of all benchmarks solved so far (y-axis) to the number of solved instances (x-axis). Note that due to the negligible running time of specification decomposition, the plot looks similar when considering all SyntComp benchmarks instead of only the decomposable ones.

For both BoSy and Strix, one can observe that modular synthesis generates a slight overhead in the beginning and thus for small specifications. In general, however, modular synthesis decreases the execution time significantly, often by order of magnitude or more, indicating that it performs particularly well for larger and more complex benchmarks. To further study the performance of modular synthesis, we depict the running times of both BoSy and Strix for modular and non-compositional synthesis on exemplary SyntComp benchmarks as well as the number of resulting components in Table 5.2. For modular synthesis, the accumulated running time of all synthesis tasks is depicted. Interestingly, the non-compositional version of BoSy outperforms all other approaches for the *shift_8* and *shift_10* benchmark, although eight and ten components, respectively, can be found. Both benchmarks have a very simple structure and can be solved efficiently with the original synthesis tools. Most likely, the overhead produced by inducing eight and ten synthesis tasks, respectively, exceeds the advantage of smaller synthesis tasks in this case. On all other benchmarks, both BoSy and Strix decrease their synthesis times with modular synthesis notably compared to the original non-compositional approaches.

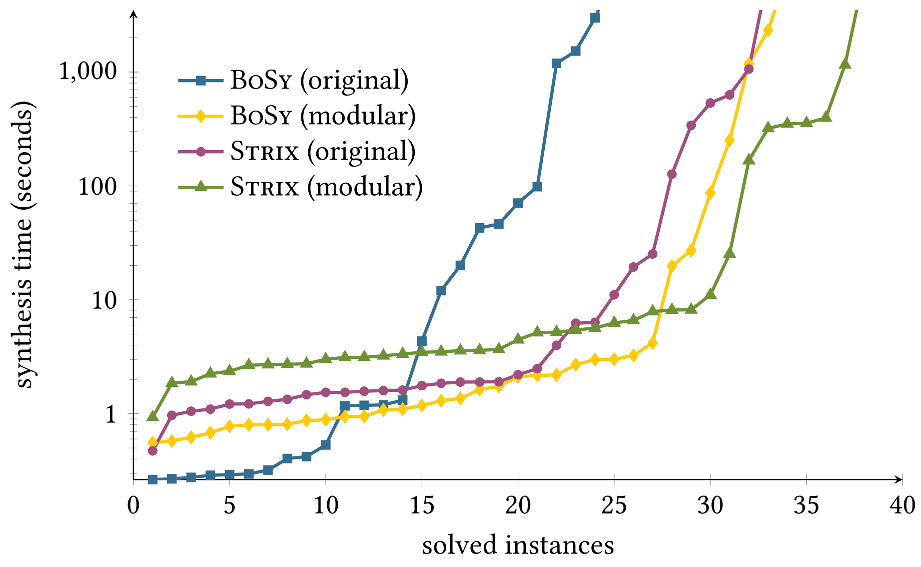


Figure 5.2.: Comparison of the performance of modular and non-compositional synthesis with BoSy and Strix on the decomposable SyntComp benchmarks for a timeout of 60 minutes. For the modular approach, the accumulated time for all synthesis tasks is depicted.

Particularly noteworthy is the benchmark *generalized_buffer_3*. In the 2021 edition of the reactive synthesis competition, SyntComp 2021, no competing tool was able to synthesize a solution for it within one hour. With modular synthesis, however, BoSy yields a result in less than 28 seconds. Note that the synthesis tool ltlsynt [MC18, RSDP22] solved the generalized buffer benchmark up to parameter 6 in the 2022 edition of the reactive synthesis competition [SYN22]. The developers of ltlsynt incorporated several optimizations into the release that competed in SyntComp 2022, including utilizing our LTL decomposition approach described in this chapter as a preprocessing technique. Our experimental results regarding the reduction of BoSy's runtime with modular synthesis for the benchmark *generalized_buffer_3* indicate that our decomposition algorithm might have a considerable impact on ltlsynt's significant improvement for the generalized buffer benchmark series.

5.7. Toward Compositional Smart Contract Synthesis

In this section, we illustrate the applicability of the results on system decomposition for sound and complete compositional monolithic synthesis of the previous sections of this chapter to the domain of smart contracts. *Smart contracts* are small programs that implement digital contracts constituting agreements between multiple parties. Typical smart contracts implement, for instance, digital coins, auctions, or asset transfers. The code of smart contracts is deployed on the blockchain. This eliminates the need for a trusted third party that establishes the correct execution of the contract. Furthermore, a smart contract cannot be altered after deployment.

Table 5.2.: Comparison of the synthesis time in seconds of BoSy and Strix for non-compositional and modular synthesis on exemplary SyntComp benchmarks. The timeout is 60 minutes. For modular synthesis, the accumulated running time of all synthesis tasks is depicted.

Benchmark	original		modular		
	BoSy	Strix	BoSy	Strix	# comp.
Cockpitboard	1526.32	11.06	**2.108**	8.168	8
Gamelogic	TO	1062.27	TO	**25.292**	4
LedMatrix	TO	TO	TO	**1156.68**	3
Radarboard	TO	126.808	**3.008**	11.04	11
Zoo10	1.316	1.54	**0.884**	2.744	2
generalized_buffer_2	70.71	534.732	**4.188**	7.892	2
generalized_buffer_3	TO	TO	**27.136**	319.988	3
shift_8	**0.404**	1.336	2.168	3.6	8
shift_10	**1.172**	1.896	2.692	4.464	10
shift_12	4.336	6.232	**3.244**	5.428	12

While this increases trust, it implies that bugs cannot be fixed retrospectively. Most smart contracts are written in the high-level programming language Solidity [Sol16] and are deployed on the Ethereum blockchain [Eth23, Woo14].

Smart contracts often involve financial transactions. Numerous bugs in smart contracts have thus led to huge monetary losses in recent history. As a consequence, there have been extensive efforts to improve the trustworthiness of smart contracts. Especially due to their usual conciseness and the criticality of their correctness, smart contracts are an inherent target for formal methods and, in particular, synthesis.

In the following, we first describe how the behavior of typical smart contracts can be specified with temporal logics. Afterward, we study how the decomposition techniques introduced in the previous sections of this chapter can be applied to smart contract specifications to enable compositional synthesis approaches.

5.7.1. Specifying Smart Contracts

Many smart contracts can be defined in terms of their *control flow* and the effect of *function calls*. The control flow describes the order in which transactions can occur and is thus of an inherent temporal nature. For instance, we can specify that, in an auction protocol, the function `bid`, which allows a user of the contract to submit a bid, may not be called after the auction has been closed via calling the function `close`. Furthermore, each function call results in specific actions to perform, such as, for instance, transferring money from one user of the contract to another. In the auction example, we can, for instance, specify that submitting a bid via function `bid` should result in storing the bid.

Many recent approaches for applying formal methods to the domain of smart contracts focus on modeling the control flow of a smart contract with a finite-state transducer (see, e.g., [ML18, MLSD19, WLC$^+$19]). Consequently, a contract's control flow is a natural target for reactive synthesis. Intuitively, the control flow transducer progresses only when a function is called. Furthermore, the effects of function calls are often instantaneous in the sense that they happen as soon as the respective function has been called and before any further function calls occur. Therefore, modeling a contract's control flow with a finite-state transducer allows for integrating function call effects.

In the following, we first elaborate on a suitable temporal logic for specifying the control flow of smart contract's as well as the effects of its function calls. Afterward, we describe the general structure of such smart contract specifications.

In recent work [FHKP23], we have identified that the control flow requirements of smart contracts can be formalized with *Temporal Stream Logic (TSL)* [FKPS19]. TSL is a temporal logic that separates temporal control and pure data. It extends LTL with the concept of *cells*, which, intuitively, allow for storing data of arbitrary type from possibly infinite domains, as well as uninterpreted *functions* and *predicates*. In TSL, data is represented as infinite streams of arbitrary type. The functions and predicates enable abstracting from concrete data points, thus allowing for focusing on the temporal control. The temporal structure of the data is expressed by temporal operators as in LTL.

In TSL, we can employ *updates* over arbitrary function terms to manipulate cells. For instance, an update $[\![\mathsf{x} \leftarrowtail f(\mathsf{y})]\!]$ denotes that the result of applying function f to cell y is assigned to cell x. Updates allow for specifying the evolution of cells, which can be seen as variables, over time. Predicates then enable performing checks on data, both on the input data and on cells. A TSL formula describes a system that receives an infinite input stream and produces an infinite stream of cell updates.

We do not describe TSL and particularly its syntax and semantics in detail here as we mainly focus on the structure of smart contract specifications, not on the particular characteristics of TSL. For a formal definition of TSL, we refer to [FKPS19]. A crucial observation regarding the semantics of TSL is, however, that functions and predicates are considered to be *uninterpreted*. Hence, a system satisfies a TSL formula if, and only if, the formula evaluates to *true* for *all* possible interpretations of the function and predicate symbols. While TSL has been extended with several first-order theories [FHP21, FHP22], TSL's original formulation with uninterpreted functions and predicates has been proven to be particularly beneficial in reactive synthesis [FKPS19, GHKF19], where the synthesis algorithm derives a control structure while the implementation of the functions and predicates is left to the developer. In reactive synthesis from TSL specifications, functions and predicates are consequently considered to be uncontrollable, while cell updates are controllable.

We introduced the past-time fragment of TSL, called pastTSL in recent work [FHKP23]. It is similar to the classical definition of TSL [FKPS19], yet, it employs LTL's *past-time* temporal operators instead of its future-time operators. So far, we only considered future-time operators such as *next* $\bigcirc$ or *globally* $\square$, which, in accordance with their name, reason about future time steps (see Section 2.4) in this thesis. Past-time operators, in contrast, only reason about the current and *previous time steps*. The operator *historically* $\boxminus$, for instance, describes that its operand needs to be satisfied in the current time step as well as in all previous time steps. Hence, it can be seen as the past-time counterpart of the operator *globally* $\square$. For the formal definition of pastTSL and all its temporal operators, we refer to [FHKP23].

We identified that the control flow requirements of many typical smart contracts can be expressed in the past-time fragment of TSL with an additional top-level $\square$-operator [FHKP23]. Intuitively, utilizing pastTSL allows for stating control flow requirements, for every function call, in terms of restrictions on the *history* of function calls and other properties such as access rights, i.e., that only a particular user, for instance, the contract's owner, may call a function, and timing restrictions, i.e., that a function can only be called before or after a certain amount of time has passed since the contract's initialization. In the auction example from above, for instance, we state that whenever function `bid` is called, function `close` must not have been called previously. Utilizing the past-time operator *historically* $\boxminus$, this results in the formula $\square\,(\mathit{bid}(\texttt{call}) \rightarrow \boxminus\neg \mathit{close}(\texttt{call}))$, where, $\mathit{bid}(\texttt{call})$ and $\mathit{close}(\texttt{call})$ are predicates over input `call`, modeling that the contract's functions `bid` and `close`, respectively, have been called. Note that using the top-level $\square$-operator is required to ensure that the control flow requirement for `bid` concerning previous calls to `close` is satisfied for calls to `bid` at every point in time and not only in the very first time step.

All other control flow requirements for function `bid` can be easily integrated into the control flow formula shown above by adding a further conjunct to the conclusion of the implication inside the scope of the $\square$-operator. If, for instance, we want to further specify that `bid` may only be called if function `cancel` has not been called previously, we obtain the overall formula $\square\,(\mathit{bid}(\texttt{call}) \rightarrow \boxminus\neg \mathit{close}(\texttt{call}) \wedge \boxminus\neg \mathit{cancel}(\texttt{call}))$, where $\mathit{cancel}(\texttt{call})$ is a predicate over input `call`, modeling that the contract's function `cancel` is called. In general, we thus obtain, for each function f_i of the smart contract, a single formula of the form

$$\varphi_{f_i} = \square\left(f_i(\texttt{call}) \rightarrow \varphi^{req}_{f_i,1} \wedge \ldots \wedge \varphi^{req}_{f_i,m}\right),$$

where the formulas $\varphi^{req}_{f_i,1}, \ldots, \varphi^{req}_{f_i,m}$ denote the m control flow requirements that are associated with the contract's function f_i. As control flow requirements can be stated with past-time operators, it follows that all formulas $\varphi^{req}_{f_i,i}$ fall into the past-time fragment with a single top-level $\square$-operator. This allows us to determine for every point in time $k \geq 0$ and every function f_i whether or not the control flow requirements permit calling f_i at point in time k solely by analyzing the current and the past time steps; the future time steps are irrelevant.

However, whether or not a function of the contract is called in a particular time step is not subject to the control of the smart contract. This is captured in our control flow specification by modeling function calls with predicates over the input `call`. Hence, a smart contract cannot enforce satisfaction of a control flow specification $\bigwedge_{1\leq i\leq n} \varphi_{f_i}$, which formalizes the allowed order of function calls. Instead, it has to *recognize* situations in which the intended control flow is violated. Utilizing Solidity's rollback functionality `revert`, a prohibited function call and all of its effects can then be undone. More precisely, when modeling function calls with predicates, the specifications modeling the control flow properties are, in many cases, not realizable since the formulas $\varphi^{req}_{f_i,1}, \ldots, \varphi^{req}_{f_i,m}$ usually describe previous function calls. Hence, we cannot utilize the above specifications for stating the control flow requirements of a smart contract.

Instead, we utilize a cell `error` to model the control flow of a smart contract. For instance, rather than stating that `close` may not have been called before `bid`, which would result in unrealizability, we state that calling function `bid` after function `close` results in updating cell `error` with a constant *raise*(), indicating that the control flow has been violated. This results in the formula $\square((\mathit{bid}(\texttt{call}) \land \neg(\boxminus\neg \mathit{close}(\texttt{call}) \land \boxminus\neg \mathit{cancel}(\texttt{call}))) \rightarrow [\![\texttt{error} \leftarrowtail \mathit{raise}()]\!])$. Furthermore, to accurately capture whether or not the control flow requirements have been violated and, in particular, to prevent that a system strategy simply always updates the cell `error` with the constant *raise*(), we state that whenever `bid` is called and `error` is updated with *raise*(), one of the control flow requirements for the function `bid` has been violated, resulting in the formula $\square((\mathit{bid}(\texttt{call}) \land [\![\texttt{error} \leftarrowtail \mathit{raise}()]\!]) \rightarrow \neg(\boxminus\neg \mathit{cancel}(\texttt{call}) \land \boxminus\neg \mathit{close}(\texttt{call}))$.

In general, we obtain the following formula $\varphi_{f_i}^{\mathit{ctrl}}$ that allows for recognizing whether or not the control flow requirements for the contract's function f_i have been violated:

$$\begin{aligned}\varphi_{f_i}^{\mathit{ctrl}} = &\square\left(\left(f_i(\texttt{call}) \land \neg\left(\varphi_{f_i,1}^{\mathit{req}} \land \ldots \land \varphi_{f_i,m}^{\mathit{req}}\right)\right) \rightarrow [\![\texttt{error} \leftarrowtail \mathit{raise}()]\!]\right)\\ &\land \square\left((f_i(\texttt{call}) \land [\![\texttt{error} \leftarrowtail \mathit{raise}()]\!]) \rightarrow \neg\left(\varphi_{f_i,1}^{\mathit{req}} \land \ldots \land \varphi_{f_i,m}^{\mathit{req}}\right)\right).\end{aligned}$$

The full specification of the control flow requirements of a smart contract with n functions $f_1, \ldots, f_n$ is then given by

$$\varphi_{\mathit{ctrlflow}} = \bigwedge_{1 \leq i \leq n} \varphi_{f_i}^{\mathit{ctrl}}.$$

In addition to the control flow requirements of a smart contract, we consider the effects of individual function calls, such as storing a submitted bid in an auction. As outlined above, these effects are often instantaneous in the sense that they happen in the same time step as the respective function call and thus without delay. Due to this observation, specifications of the effects of function calls have, apart from the usual $\square$-operator that ensures that the effect is not only required in the very first time step but in all, no temporal operators. For specifying the effects of a call of the contract's function `bid` in the auction protocol, for instance, one would consequently expect the formula $\square(\mathit{bid}(\texttt{call}) \rightarrow \varphi_{\mathit{store}})$, where φ_{store} is a formula encoding the particular effect of storing the submitted bid. However, this formula requires the storage of the submitted bid even if submitting the bid was prohibited by the control flow requirements. If, for instance, the auction was already closed and a user nevertheless submits a bid, then the user's call of `bid` violates the control flow specification. As outlined above, we cannot ensure that no bid is submitted after the auction is closed. Nevertheless, we need to ensure that function calls are only obliged to induce their effects if they adhere to the contract's control flow. Otherwise, realizability of the specification might not be guaranteed. Therefore, we include the control flow requirements in the specification of the function effects, resulting in a formula

$$\varphi_{f_i}^{\mathit{eff}} = \square\left(\left(f_i(\texttt{call}) \land \varphi_{f_i,1}^{\mathit{req}} \land \ldots \land \varphi_{f_i,m}^{\mathit{req}}\right) \rightarrow \varphi_{f_i,1}^{\mathit{eff}} \land \ldots \land \varphi_{f_i,\ell}^{\mathit{eff}}\right)$$

for function f_i, where the formulas $\varphi_{f_i,1}^{\mathit{eff}}, \ldots, \varphi_{f_i,\ell}^{\mathit{eff}}$ encode the ℓ effects of a call of f_i. All in all, smart contract specifications describing the control flow and the effects of function calls then usually follow a simple pattern:

$$\varphi_{sc} = \varphi_{\mathit{ctrlflow}} \land \bigwedge_{1 \leq i \leq n} \varphi_{f_i}^{\mathit{eff}}.$$

Consequently, such smart contract specifications naturally qualify as an interesting domain for specification decomposition, as introduced in the previous sections of this chapter, due to their conjunctive nature. Therefore, we study the results of our decomposition approaches for smart contract specifications in the following to determine whether smart contracts constitute a suitable domain for compositional synthesis via specification decomposition.

5.7.2. Decomposition Smart Contract Specifications

In this section, we study whether smart contract specifications can be decomposed into several components for compositional synthesis. Therefore, we first elaborate on TSL synthesis in general. Afterward, we utilize these observations to apply the independence criteria introduced in the previous sections to smart contract specifications.

In general, the realizability problem for TSL in its classical definition, i.e., without past-time operators, and consequently also the synthesis problem for TSL, is undecidable [FKPS19]. Even though past-time variants of temporal logics like LTL often lead to significantly easier algorithms as they restrict the expressible properties to safety properties, we establish in [FHKP23] that the realizability problem for pastTSL, and consequently also the synthesis problem for pastTSL, is undecidable as well. However, inspired by Finkbeiner et al.'s result that the realizability problem for TSL can be soundly approximated with the LTL realizability problem [FKPS19], we provide a sound approximation of the pastTSL realizability problem with the realizability problem for pastLTL specifications, i.e., the past-time fragment of LTL, in [FHKP23]. This allows for utilizing existing synthesis algorithms for specifications given in (past-time) LTL.

Following the future-time counterpart, the main idea of the approximation is to replace the predicate and update terms of the pastTSL formula with atomic propositions. Additional conjuncts ensure that the semantics of cells are respected. In particular, the conjuncts formalize that, for every cell, exactly one update is performed in every time step. For the formal definition of the pastLTL approximation, we refer to [FHKP23]. The approximation is sound in the sense that every strategy that realizes the approximated pastLTL formula can be translated into one for the original pastTSL formula. However, it is not complete as the basic property of function and predicates that they evaluate to the same value when applied to terms that evaluate to the same value is lost. Therefore, unrealizability of the pastLTL approximation does not necessarily result in unrealizability of the original pastTSL formula. This matches the respective LTL approximation of future-time TSL formulas [FKPS19]. In practice, however, we never encountered a realizable pastTSL specification with an unrealizable pastLTL specification.

Since pastTSL realizability and thus also pastTSL synthesis is approximated with pastLTL realizability and synthesis, respectively, we study the decomposability of smart contract specifications on the respective pastLTL approximation. Hence, we assume in the following that the formulas φ_{sc}, $\varphi_{ctrlflow}$, and $\varphi_{f_i}^{eff}$ denote the pastLTL approximations of the respective pastTSL formulas presented in the previous section. Furthermore, for each cell c occurring in the pastTSL specification, let φ_{c} denote the formula ensuring that the semantics of cell c are respected. Note that atomic propositions approximating updates serve as output variables in the synthesis problem since updates are intuitively controlled by the system. Atomic propositions approximating predicate terms, in contrast, constitute input variables since the evaluation of predicates is, due to the use of uninterpreted functions and predicates, uncontrollable.

The inherent conjunctive nature of smart contract specifications presented in the previous sections suggests decomposability into several components for synthesis. To substantiate this intuition, we employ the language-based independence criterion from Section 5.2 and utilize LTL-specific results from Section 5.3. Recall that, according to the language-based independence criterion from Section 5.2, a specification can be decomposed into two subspecifications if they form *independent sublanguages*. Two languages L_1 and L_2 are independent sublanguages of a language L if they are non-contradictory and if $L_1 \mid\mid L_2 = L$ holds (see Definition 5.6).

First observe that, by Lemma 5.7, we have $\mathcal{L}(\varphi_1) \mid\mid \mathcal{L}(\varphi_2) = \mathcal{L}(\varphi)$ for a conjunctive LTL formula $\varphi = \varphi_1 \land \varphi_2$. This result is not immediately applicable to our smart contract specifications since the control flow requirements feature past-time temporal operators, which are not present in the definition of LTL considered in Section 5.3. However, the proof of Lemma 5.7 only utilizes the semantics of conjunction. In particular, it does not rely on φ_1 and φ_2 being future-time LTL formulas. Therefore, the result carries over to our smart contract specifications. Consequently, we obtain that

$$\mathcal{L}(\varphi_{sc}) = \mathcal{L}(\varphi_{ctrlflow}) \mid\mid \mathcal{L}(\varphi^{eff}_{f_1}) \mid\mid \ldots \mid\mid \mathcal{L}(\varphi^{eff}_{f_n}) \mid\mid \mathcal{L}(\varphi_{\mathtt{error}}) \mid\mid \mathcal{L}(\varphi_{\mathtt{c}_1}) \mid\mid \ldots \mid\mid \mathcal{L}(\varphi_{\mathtt{c}_j})$$

holds, where $\mathtt{c}_1, \ldots, \mathtt{c}_c$ denote the c cells apart from the cell `error` occurring in the original pastTSL specification of the smart contract.

Second, we study whether the suggested sublanguages are non-contradictory according to Definition 5.5. Recall that for two LTL formulas φ_1 and φ_2 with $\mathit{prop}(\varphi_1) \cap \mathit{prop}(\varphi_2) \subseteq I$, where I is the set of input variables, the languages $\mathcal{L}(\varphi_1)$ and $\mathcal{L}(\varphi_2)$ are non-contradictory by Lemma 5.8. Observe that the proof of this property again does not utilize the fact that φ_1 and φ_2 pure future-time LTL formulas. Rather, it only relies on φ_1 and φ_2 not sharing output variables. Hence, although Lemma 5.8 is not immediately applicable to our smart contract specifications due to the existence of past-time operators, the result carries over. Consequently, we obtain that, for all of the contract's functions f_i, the languages $\mathcal{L}(\varphi_{ctrlflow})$ and $\mathcal{L}(\varphi^{eff}_{f_i})$ are non-contradictory if the specifications do not share output variables.

Recall that only updates in the original pastTSL formula constitute output variables in the pastLTL approximation. Furthermore, note that the pastLTL approximation introduces additional conjuncts $\varphi_{\mathtt{c}}$ that ensure that, for every cell $\mathtt{c}$, exactly one update is performed in every time step. Clearly, these additional conjuncts contain, for each cell, the atomic propositions of all possible updates occurring in the entire smart contract specification. Consequently, all parts of the smart contract specification that contain some update of cell $\mathtt{c}$ share output variables with the additional conjunct $\varphi_{\mathtt{c}}$. Hence, it follows that all conjuncts that contain an update of cell $\mathtt{c}$ transitively share output variables with each other via conjunct $\varphi_{\mathtt{c}}$.

The formula $\varphi_{ctrlflow}$, which specifies the control flow requirements of the smart contract, contains updates of the cell `error`. Therefore, it can only be separated from other parts of the smart contract specification if `error` is not updated in them. Fortunately, this is the case for the formulas specifying the effect of function calls since we introduced the cell `error` artificially in $\varphi_{ctrlflow}$ to be able to recognize violations of the control flow requirements.

Furthermore, during the course of specifying the control flow of ten common smart contracts in [FHKP23], we never experienced that the control flow specification required any other cell updates. While $\varphi_{ctrlflow}$ might contain other cells, they were always embedded in predicate or function terms. This is due to the intuition that cells represent variables of the contract. Cell updates consequently represent variable assignments. For specifying the allowed order of function calls, it might be necessary to argue about the *value* of cells. In the auction, for instance, one could utilize a cell `highestBid` to store the current highest bid, and submitting a new bid via function `bid` is only allowed if the new bid is higher than the current highest bid. However, such properties are formalized with *predicates* over cells rather than cell updates. In the analogy of variables in programs, one would rather require that a variable has a certain value than require that a variable has been assigned a certain value.

Hence, we experienced that, in smart contract specifications, the conjunct $\varphi_{ctrlflow}$ formalizing the contract's control flow requirements usually does not contain any cell updates other than the update of the artificial cell `error`. Furthermore, the formulas specifying the effects of function calls do not contain the cell `error` at all and thus, in particular, also no updates of cell `error`. Therefore, it follows that the languages $\mathcal{L}(\varphi_{cf})$ and $\mathcal{L}(\varphi_{ce})$, where

$$\begin{aligned}\varphi_{cf} &= \varphi_{ctrlflow} \land \varphi_{\texttt{error}}\\ \varphi_{ce} &= \bigwedge_{1 \leq i \leq n} \varphi^{eff}_{f_i} \land \bigwedge_{1 \leq j \leq c} \varphi_{\texttt{c}_j},\end{aligned}$$

are non-contradictory. Due to the conjunctive nature of the specification, we further have $\mathcal{L}(\varphi_{cf}) \mid\mid \mathcal{L}(\varphi_{ce}) = \mathcal{L}(\varphi_{sc})$ as outlined above. Hence, $\mathcal{L}(\varphi_{cf})$ and $\mathcal{L}(\varphi_{ce})$ form independent sublanguages of $\mathcal{L}(\varphi_{sc})$. By Theorem 5.1, the modular synthesis algorithm from Algorithm 5.1 is sound and complete for decompositions that form independent sublanguages of the original specification. In particular, it is guaranteed that the individual synthesis tasks for the subspecifications succeed whenever the original specification is realizable.

Consequently, it follows that, for a realizable smart contract specification φ_{sc}, there exist winning strategies for the subspecifications φ_{cf} and φ_{ce} describing the contract's control flow and the effects of its function calls, respectively. Hence, we can consider separate synthesis tasks for the control flow and the function call effects. Since we only argued about the *structure* of smart contract specifications outlined in the previous section and not about a concrete specification, this result generalizes to *all* smart contract specifications that follow the respective specification structure. Therefore, it is not necessary to analyze a specific smart contract specification to determine whether or not it can be decomposed into a part describing the contract's control flow and a part describing the effects of its function calls, but decomposability is guaranteed by the general specification structure.

This observation gave rise to the development of a synthesis algorithm for smart contract control flows specified in pastTSL in the above manner, which constructs a state machine realizing the control flow specification [FHKP23]. The approach focuses on the control flow only and does not take the effects of function calls into account. This is possible without sacrificing realizability of the specification as long as the specification follows the structure described in the previous section due to the general decomposability results introduced above.

The effects of function calls can then either, matching the conceptual context of compositional synthesis of this thesis, be synthesized separately, or they can be integrated manually.

Focusing on the latter possibility, we developed the tool SCSynt that fully automatically constructs Solidity code from a pastTSL control flow specification that adheres to the general specification structure [FHKP23]. First, SCSynt derives the pastLTL approximation of the given pastTSL specification. Then, making use of the past-time nature of smart contract specifications, it utilizes an efficient BDD-based synthesis algorithm for constructing a finite-state transducer that realizes the approximated specification. Afterward, it translates the transducer into Solidity code, which can then manually be equipped with further functionality, such as the effects of function calls, by the developer.

On a technical note, observe that we consider a slightly different specification structure in [FHKP23] than in this thesis. There, we do not utilize an artificial cell `error` to recognize situations in which the control flow requirements are violated. Instead, we include the initial control flow formulas, i.e., the formulas φ_{f_i}, as *assumptions* into the specification. In general, this does not allow recognizing violations of the control flow requirements in a synthesized strategy, as the strategy is not required to distinguish situations in which the control flow requirements are violated from those in which they are not. The employed synthesis algorithm in SCSynt, however, constructs an incomplete finite-state transducer that only produces infinite traces for input sequences that match situations in which all control flow requirements are met. In particular, in each state, it has an outgoing edge for some predicate representing a function call if, and only if, the function call adheres to the control flow specification in this particular time step and thus if, and only if, the cell `error` is not updated with $\mathit{raise}()$. Consequently, the synthesized state machine also allows for recognizing situations in which the control flow requirements are violated, yet, in a slightly different manner. The translation of the transducer to Solidity can be implemented for both versions of the specification structure and only requires minor changes to switch between the two variants. For more details on the slightly altered specification structure and the employed synthesis algorithm, we refer to [FHKP23].

Focusing on a smart contract's control flow enables efficient synthesis: we have synthesized Solidity code for ten smart contracts from their control flow specifications formalized in pastTSL with SCSynt, for all of which the synthesis procedure terminated in less than 13 seconds [FHKP23]. This underlines the particular applicability of compositional techniques and, especially, recognizing specification decomposability in certain domains such as smart contracts. Specifically, the observations of this section and the very encouraging results on efficient smart contract control flow synthesis of [FHKP23] suggest identifying more domains for which the approaches introduced in this chapter are particularly beneficial.

5.8. Summary

We have presented a modular synthesis algorithm that applies compositional techniques to reactive synthesis. It reduces the complexity of synthesis by first decomposition the specification into several components in a preprocessing step and then performing independent synthesis tasks for the components. We have introduced a language-based criterion for decomposition algorithms that, if satisfied, ensures soundness and completeness of modular synthesis. Focusing on specifications given as LTL formulas, we have lifted the language-based criterion to the temporal logic level utilizing a syntactic analysis of the formula. Based on the LTL independence criterion, we have presented an LTL decomposition algorithm that ensures soundness and completeness of modular synthesis. To optimize LTL independence for specifications in the common assume-guarantee form, we have introduced a sound and complete criterion for dropping assumptions from LTL formulas while ensuring equirealizability. This allows for more fine-grained decompositions in many cases. We have incorporated assumption dropping into the LTL decomposition algorithm. Furthermore, we have extended the assumption-dropping criterion to LTL formulas in a non-strict assume-guarantee form, i.e., to LTL formulas that consist of several conjuncts which are possibly in strict assume-guarantee form, and presented an accordingly adapted version of the LTL decomposition algorithm, which still ensures soundness and completeness of modular synthesis. We have implemented both modular synthesis and the LTL decomposition algorithm and evaluated it on the publicly available benchmarks of the reactive synthesis competition SyntComp. We have compared our approach for the state-of-the-art synthesis tools BoSy and Strix to their non-compositional forms. Our experiments clearly demonstrate a significant advantage of modular synthesis with LTL decomposition over traditional synthesis algorithms. While the overhead introduced by decomposing the specification is negligible, both BoSy and Strix are able to synthesize more solutions for more benchmarks with modular synthesis than in their non-compositional form. Moreover, on large and complex benchmarks, both tools improve their synthesis times notably. Lastly, we have demonstrated that specifications of smart contracts can be decomposed into independent parts describing the contract's control flow and the effects of function calls, thus allowing for compositional synthesis approaches. This indicates the particular applicability of our decomposition algorithms to specific domains.

Chapter 6

Dependency-based Incremental Synthesis of Dominant Strategies

In the previous chapter, we presented a modular synthesis algorithm that breaks down the synthesis of a monolithic system into separate synthesis tasks for individual system components. Furthermore, we introduced a suitable decomposition algorithm that, given an LTL specification, identifies system components for which winning strategies can be synthesized independently. Unfortunately, the decomposition approach only identifies more than a single independent component for 39 out of the 346 publicly benchmarks of the annual synthesis competition SyntComp [BEJ14]. For the vast majority of the benchmarks, our modular synthesis approach does not have any influence on the synthesis time.

In this chapter, we thus build upon the idea of applying compositional techniques to monolithic synthesis via decomposition. However, we sacrifice the conceptual simplicity of the decomposition technique to obtain more fine-grained decompositions. For this sake, we consider a weaker notion than winning as a strategy requirement for the individual synthesis tasks: *remorsefree dominance* [DF11] allows for violating the component specification as long as no other strategy would have satisfied it in the same situation. Synthesizing dominant strategies rather than winning ones thus allows for implicitly assuming that the other processes will not maliciously violate the shared goal. For safety specifications, remorsefree dominance is a compositional notion, just like winning, i.e., the composition of two dominant strategies is again dominant [DF14]. Furthermore, if a winning strategy exists, then all dominant strategies are winning. This directly leads to a compositional synthesis approach that synthesizes individual remorsefree dominant strategies [DF14], enabling modular synthesis for monolithic systems similar to the one introduced in the previous section also for dominant strategies.

Although synthesizing remorsefree dominant strategies rather than winning ones allows for finding solutions in more cases, the existence of a dominant strategy is not guaranteed. Often, a component c_i depends on the well-behavior of another component c_j in the sense that c_i needs to anticipate some future action by c_j to determine the correct behavior for itself. In such situations, there is no dominant strategy for c_i as the decision of which strategy is best for c_i depends on the specific strategy for c_j. Therefore, a suitable decomposition algorithm is needed for modular synthesis when considering remorsefree dominant strategies as well.

To further increase the number of system components, we address the synthesis problem with an *incremental* approach rather than a purely compositional one. As in modular synthesis introduced in the previous chapter, we split the system into components. However, we do not try to find remorsefree dominant strategies for each component individually. Rather, we proceed in an incremental fashion, thus considering components one after another. We call the order in which strategies for the components are constructed the *synthesis order*. Instead of searching for dominant strategies for all components, we then only require strategies to be dominant *under the assumption* that the components with a lower rank in the synthesis order do not deviate from their previously synthesized strategies. This enables the use of both *implicit* assumptions – by utilizing dominant rather than winning strategies – and *explicit* assumptions – by providing concrete strategies of components with a lower rank. Similar to modular synthesis with winning strategies, incremental synthesis reduces the complexity of synthesis by decomposing the system into components; additionally, it allows for more fine-grained decompositions by not performing the individual synthesis tasks completely compositionally but incrementally.

The key question in incremental synthesis is how to find a suitable synthesis order that ensures that all synthesis tasks in incremental synthesis succeed. We propose two methods for computing the synthesis order that offer different trade-offs between precision and computational cost. The first technique is based on a semantic dependency analysis of the output variables of the system. Intuitively, an output variable u depends on another output variable v if determining u's correct valuation at some point in time requires information about the valuation of v at a future point in time. We then build equivalence classes of output variables based on cyclic dependencies. These equivalence classes constitute the components of the system: a component controls all the system outputs that are contained in the respective equivalence class. The synthesis order is then defined following the dependencies between the derived components. Therefore, it resolves dependencies of a component c_i to a component c_j that prevent the existence of remorsefree dominant strategies for c_i by assigning c_j a lower rank in the synthesis order. This ensures that a strategy for c_j is synthesized prior to c_i's synthesis task, allowing for taking this strategy into account when synthesizing a remorsefree dominant strategy for c_i, therefore addressing c_i's need for information about c_j's behavior at a future point in time.

Similar to the LTL decomposition algorithm introduced in the previous chapter, the second technique is based on a syntactic analysis of the specification. Due to the different strategy requirements, however, the analysis differs inherently from the one for winning strategies. It thoroughly examines the structure of the LTL specification and takes the semantics of the different kinds of temporal operators into account. While the syntactic decomposition approach has less computational cost than the semantic one, it is less precise. In particular, it conservatively overapproximates the semantic dependencies, resulting in coarser decompositions and thus potentially less independent component. However, we introduce additional rules for simplifying the specification that needs to be considered for the individual components during their synthesis tasks, which allow for avoiding unnecessary syntactic dependencies in many cases. Realizability and unrealizability of the synthesis task is preserved by the simplifications.

We have implemented a prototype of our incremental synthesis algorithm based on the bounded synthesis tool BoSy [FFT17]. First, we integrated the synthesis of remorsefree dominant strategies instead of winning ones as described in Section 2.8.2 into the tool. Afterward, we put the incremental approach into practice. We compare our prototype to BoSy in its classical version, i.e., to the tool that non-compositionally synthesizes winning strategies for monolithic systems, on scalable benchmarks. The results clearly demonstrate the advantage of incremental synthesis over classical synthesis algorithms: incremental synthesis significantly outperforms BoSy for larger but decomposable systems.

Publications and Structure. This chapter is based on work published in the proceedings of the *18th International Symposium on Automated Technology for Verification and Analysis* [FP20a] and the extended version [FP20b] of this publication. The author of this thesis is the lead author of both publications.

This chapter is structured as follows. After introducing a running example, which we use throughout the chapter, we present the main concept of incremental synthesis and the synthesis order in Section 6.2 and prove the soundness and completeness of incremental synthesis for a particular class of synthesis orders. In Sections 6.3 and 6.4, we introduce the notions of semantic and the syntactic dependencies, respectively. Furthermore, for both kinds of dependencies, we present a decomposition algorithm based on an dependency analysis and show that soundness and completeness of incremental synthesis are guaranteed when computing the system's component as well as the synthesis order with either the semantic or the syntactic component selection algorithm. Afterward, we introduce rules for simplifying the specifications for individual synthesis tasks in incremental synthesis while preserving realizability and unrealizability in Section 6.5. Lastly, in Section 6.6, we present an experimental evaluation of the performance of incremental synthesis.

6.1. Running Example

In safety-critical systems such as autonomous cars, the correctness of the system's implementation with respect to a given specification is crucial. Hence, they are an obvious target for synthesis. An autonomous car, however, requires a vast amount of different functionalities, leading to an enormous state space when all functionalities are considered and synthesized as a single unit. While a compositional approach may reduce the complexity of synthesis, in most scenarios, there are neither winning nor remorsefree dominant strategies for the separate components due to the complex dependencies between the different functionalities. As an example, consider a specification for two functionalities of an autonomous car, gearing and acceleration. In the following, we call these functionalities the gearing unit and the acceleration unit, respectively. The acceleration unit is required to decelerate before curves and not to accelerate in curves. To prevent traffic jams, the autonomous car is additionally required to accelerate eventually if no curve is ahead. In order to save fuel, it should not accelerate and decelerate all the time. These requirements can be specified in LTL as follows:

$$\begin{aligned}\varphi_{acc} = {} & \square(ahead \rightarrow \bigcirc\, dec) \land \square(in \rightarrow \bigcirc \neg acc) \land \square\Diamond\, keep \\ & \land \square((\neg in \land \neg ahead) \rightarrow \bigcirc\Diamond\, acc) \land \square\neg(acc \land dec) \\ & \land \square\neg(acc \land keep) \land \square\neg(dec \land keep) \land \square(acc \lor dec \lor keep),\end{aligned}$$

where *ahead* and *in* are input variables denoting whether a curve is ahead or whether the car is in a curve, respectively. The output variables are *acc* and *dec*, denoting acceleration and deceleration, respectively, and *keep*, denoting that the current speed is kept. For simplicity, we assume that the car always moves forward irrespective of whether it decelerates, accelerates, or keeps the current speed, i.e., it never stops. Note that φ_{acc} is only realizable if we assume that a curve is not followed by another one with only one step in between infinitely often. Furthermore, we need assumptions that ensure that the road behaves *realistic* in the sense that, for instance, the car will not stay in a curve forever.

The gearing unit can choose between two gears. It is required to use the smaller gear when the car is accelerating and the higher gear if the car reaches a steady speed after accelerating. This can be specified in LTL as follows:

$$\begin{aligned}\varphi_{gear} = &\square((acc \land \bigcirc\, acc) \rightarrow \bigcirc\bigcirc g_1) \land \square((acc \land \bigcirc\, keep) \rightarrow \bigcirc\bigcirc g_2)\\ &\land \square\neg(g_1 \land g_2) \land \square(g_1 \lor g_2),\end{aligned}$$

where g_1 and g_2 are output variables denoting whether the first or the second gear, respectively, is used. A naïve compositional synthesis approach would try to synthesize strategies for the acceleration unit, i.e., a system controlling *acc*, *dec*, and *keep*, and the gearing unit, i.e., a system controlling g_1 and g_2, separately. However, there clearly do not exist winning strategies for the acceleration unit or the gearing unit when considering the full specification $\varphi_{car} = \varphi_{acc} \land \varphi_{gear}$ of the autonomous car since it contains requirements on both *acc*, *dec*, *keep* and g_1, g_2.

When applying the syntactic LTL decomposition algorithm introduced in Chapter 5 to the full specification $\varphi_{car} = \varphi_{acc} \land \varphi_{gear}$ of the autonomous car, we obtain dependencies between *acc*, *dec*, and *keep* as well as between g_1 and g_2 due to the respective mutual exclusion requirements. Furthermore, since the valuations of *acc* and *dec* affect the valuations of g_1 and g_2 due to the requirements stated in φ_{gear}, namely that the car has to react with the correct gear if it either is accelerating or reaches a steady speed after acceleration, the algorithm derives dependencies between *acc* and g_1 as well as between *acc* and g_2 and *keep* and g_2. Therefore, the dependency graph contains only a single connected component, which contains all output variables of the specification. Hence the syntactic LTL decomposition algorithm does not decompose the specification into multiple components.

In fact, if we require components to not consider output variables of other components as inputs – as we do in the approaches introduced in the previous chapter – we can never separate the gears g_1 and g_2 from *acc* and *keep* due to the first two conjuncts of φ_{gear}. Therefore, for a successful decomposition, we need to permit components that, similar to processes in certain distributed architectures, can observe the outputs of other components and react to them. However, the outputs of the other components are then considered to be adversarial. Thus, in particular, we cannot separate the gears g_1 and g_2 from *acc* and *keep*. When searching for a strategy controlling the gear, we need to take the situation into account in which *acc* occurs at some point in time, followed by *both acc* and *keep* in the next step. Then, the gearing unit needs to set both g_1 and g_2 to *true* in the next step due to the first two conjuncts of φ_{gear}, contradicting the mutual exclusion requirement for the gears.

Searching for a remorsefree dominant strategy [DF11] rather than a winning strategy for the gearing unit overcomes this problem. In the interplay of a strategy for the gearing unit and the acceleration unit, it will never be the case that both *acc* and *keep* are *true* at the same point in time due to their mutual exclusion requirement specified in φ_{acc}. Since no strategy at all for the gearing unit can handle situations where both *acc* and *keep* co-occur, a remorsefree dominant strategy is allowed to violate the specification in such situations. Due to the properties of remorsefree dominance, this strategy is even a valid choice when not splitting the specification into subspecifications but when taking the full specification φ_{car} into account. This enables us, similar to the dominance-based compositional synthesis algorithm for distributed systems [DF14], to perform compositional synthesis without the need for specification decomposition. However, while there exists a remorsefree dominant strategy for the gearing unit, φ_{car} is not admissible for the acceleration unit: as long as the car accelerates after a curve, the conjunct $\square((\neg in \land \neg ahead) \rightarrow \diamondsuit\, acc)$ is satisfied. If the gearing unit does not react correctly, the specification is violated. Yet, an alternative strategy for the acceleration unit that accelerates at a different point in time at which the gearing unit reacts correctly satisfies the specification. Thus, neither a compositional approach using winning strategies nor one using remorsefree dominant strategies is able to synthesize strategies for the acceleration unit and the gearing unit of the autonomous car.

However, the lack of a dominant strategy for the acceleration unit is only due to the uncertainty of whether the gearing unit will comply with the acceleration strategy. The only dominant strategy for the gearing unit is to react correctly to the change in speed. Hence, providing this knowledge to the acceleration unit by synthesizing the strategy for the gearing unit beforehand and making it available, yields a remorsefree dominant – and even winning – strategy for the acceleration unit. Thus, synthesizing the components *incrementally* instead of *compositionally* allows for separate strategies even if there is a dependency between the components.

6.2. Incremental Synthesis

In this section, we introduce a synthesis algorithm based on remorsefree dominant strategies, where, in contrast to compositional synthesis, the components are not necessarily synthesized independently but one after another. The strategies that are already synthesized provide further information to the one under consideration. Therefore, the specification is not required to be admissible for all components individually. Recall, for instance, that there is no dominant strategy for the acceleration unit for the autonomous car from Section 6.1. However, when provided with a dominant gearing strategy, there is even a winning strategy for the acceleration unit. Therefore, synthesizing strategies for the components incrementally rather than compositionally allows us to synthesize a strategy for the autonomous car.

Similar to the previous chapter, a component $c = (\varphi_c, I_c, O_c)$ of the system consists of a component specification φ_c, and a *component interface*, represented by sets I_c and O_c of input and output variables. In contrast to the system components considered in Chapter 5, however, we consider the full system specification φ as specification for all components. Therefore, we sometimes omit it from the component description. Component inputs and outputs are disjoint, i.e., we have $I_c \cap O_c$. Furthermore, component outputs are a subset of the outputs of the entire system, i.e., we have $O_c \subseteq O$. This coincides with the definition from the previous chapter.

However, contrasting the previous component definition, component inputs are not required to be a subset of the inputs of the entire system but also contain output variables of the system that are not assigned to the considered component, i.e., we have $I_c = (I \cup O) \setminus O_c$. Intuitively, this models that components can observe the behavior of all other components, resulting in a perfect information setting. As in the previous chapter (see Definition 5.1), a system decomposition $\mathbb{D}$ of (I, O) is then a vector of components. The components capture the entire system, i.e., each output variable is assigned to a component. Furthermore, we require that the components are disjoint, i.e., they do not share output variables.

Note that permitting components to observe other components' output variables and, in particular, react to them, requires us to consider strategies represented with Moore transducers only. Otherwise, the parallel composition of the resulting strategies is not guaranteed to be complete (see Section 2.6.1). Thus, contrasting the setting from Chapter 5 but similar to the setting for distributed systems in Part I, we always consider strategies to be representable by Moore transducers in the remainder of this chapter.

In the following, we first introduce the so-called synthesis order, which assigns the components a rank, defining the order in which the components are synthesized incrementally. Afterward, we introduce the incremental synthesis algorithm.

6.2.1. Synthesis Order

Given a decomposition $\mathbb{D} = \langle c_1, \ldots, c_n \rangle$, that partitions the system into n components, the *synthesis order* $<_{syn}$ defines in which order the components of a system are considered in the incremental synthesis algorithm. In particular, if $c_1 <_{syn} c_2$ holds, then a strategy for c_1 is synthesized in incremental synthesis before one for c_2. We model the synthesis order with a function $rank_{syn} : \mathbb{D} \rightarrow \mathbb{N}$ that assigns a *rank* to every component of the decomposition, i.e., to every $c \in \mathbb{D}$. Intuitively, we then aim at synthesizing strategies for components with lower ranks before those with higher ranks, providing the strategies for components with lower ranks to the following synthesis tasks. For the running example from Section 6.1, for instance, we would choose a ranking function that assigns the gearing unit a lower rank than the acceleration unit, ensuring that the synthesis task for the acceleration unit can rely on the results of the previous synthesis task.

The ranking function $rank_{syn} : \mathbb{D} \rightarrow \mathbb{N}$ can also assign the same rank to several components. Strategies for these components with the same rank are then synthesized compositionally, i.e., entirely separately. Hence, intuitively, they do not depend on each other in the sense that they do not need explicit information about their strategies. However, they might require the same information about the strategies of other components, namely those with smaller ranks.

As discussed in Chapter 3 for the distributed case, synthesizing strategies completely separately requires *compositionality*, i.e., that the parallel composition of the synthesized strategies again satisfies the strategy requirement such as winning or dominance. While compositionality is always guaranteed when seeking winning strategies, it is not for remorsefree dominant strategies (see Chapter 3). Therefore, the synthesis order needs to ensure that it only assigns the same rank to those components of the decompositions whose parallel composition is again remorsefree dominant. We formalize this requirement explicitly as follows.

Definition 6.1 (Compositionality-Preserving Synthesis Order)**.**
Let φ be an LTL formula over atomic propositions V. Let $\mathbb{D}$ be a decomposition of (I, O). Let $rank_{syn}$ be a ranking function defining the synthesis order. We call $rank_{syn}$ *compositionality-preserving* if, and only if, for all components $c_1, c_2 \in \mathbb{D}$ with $rank_{syn}(c_1) = rank_{syn}(c_2)$ it holds that if s_1 and s_2 are dominant strategies for φ and c_1 and c_2, respectively, then $s_1 \mid\mid s_2$ is dominant for φ and $c_1 \mid\mid c_2$.

In [DF14], Damm and Finkbeiner showed that compositionality is guaranteed in the distributed case for remorsefree dominance when considering *safety specifications* only (see also Theorem 3.1). In the monolithic case, which is considered in this part of the thesis, the result follows analogously when synthesizing strategies for the components of a decomposition of the single process separately. In the following, we extend this result to specifications where only a single component affects the liveness part. Intuitively, a violation of the liveness part can then always be led back to the single component affecting it, contradicting the assumption that its strategy is dominant. Formally:

Theorem 6.1. *Let φ be an LTL formula over atomic propositions V. Let $\mathbb{D}$ be a decomposition of (I, O). Let s_1 and s_2 be dominant strategies for φ and components $c_1 \in \mathbb{D}$ and $c_2 \in \mathbb{D}$, respectively. If φ is either (i) a safety property, or (ii) the liveness part of φ is only affected by output variables of c_1, then $s_1 \mid\mid s_2$ is dominant for φ and $c_1 \mid\mid c_2$.*

Proof. If (i) holds, i.e., if φ is a safety property, then compositionality follows analogously to the distributed case proven in [DF14]. Next, let (ii) hold, i.e., the liveness part of φ is only affected by output variables of c_1. For the sake of readability, let $I_1 \subseteq V$ and $I_2 \subseteq V$ denote the inputs of c_1 and c_2, respectively, while $O_1 \subseteq V$ and $O_2 \subseteq V$ denote the outputs of c_1 and c_2, respectively. Let $V_1 = I_1 \cup O_1$ and $V_2 = I_2 \cup O_2$. Let $I_{1,2} = (I_1 \cup I_2) \setminus (O_1 \cup O_2)$, let $O_{1,2} = O_1 \cup O_2$, and let $V_{1,2} = I_{1,2} \cup O_{1,2}$ be the inputs, outputs, and variables of $c_1 \mid\mid c_2$, respectively. Suppose that $s_1 \mid\mid s_2$ is not remorsefree dominant for φ and $c_1 \mid\mid c_2$. Then, there exists an alternative strategy t for $c_1 \mid\mid c_2$ and sequences $\gamma \in (2^{I_{1,2}})^\omega$ and $\gamma' \in (2^{V \setminus V_{1,2}})^\omega$ such that $comp(s_1 \mid\mid s_2, \gamma) \cup \gamma' \not\models \varphi$ holds, while we have $comp(t, \gamma) \models \varphi$. Let φ_{safe} and φ_{live} be the LTL formulas describing the safety and liveness properties φ can be disassembled into, i.e., such that $\varphi \equiv \varphi_{safe} \land \varphi_{live}$ holds.

First, suppose that $s_1 \mid\mid s_2$ violates φ_{safe} on input γ when considering γ', i.e., that we have $comp(s_1 \mid\mid s_2, \gamma) \cup \gamma' \not\models \varphi_{safe}$. In contrast, $comp(t, \gamma) \cup \gamma' \models \varphi_{safe}$ follows from the assumption and the semantics of conjunction. Hence, $s_1 \mid\mid s_2$ is not remorsefree dominant for φ_{safe} and $c_1 \mid\mid c_2$. Since φ_{safe} is a safety property, however, it follows immediately from the compositionality result for safety properties from [DF14] that s_1 and s_2 cannot both be dominant for φ and c_1 and c_2, respectively; yielding a contradiction.

Second, suppose that $s_1 \mid\mid s_2$ does not violate φ_{safe} on input γ when considering γ'. Then, it follows from the assumption that $comp(s_1 \mid\mid s_2, \gamma) \cup \gamma' \not\models \varphi$ holds and from the semantics of conjunction that we have $comp(s_1 \mid\mid s_2, \gamma) \cup \gamma' \not\models \varphi_{live}$. By assumption, the liveness part of φ and thus φ_{live} is only affected by the output variables of component c_1. Hence, the violation of φ_{live} is, intuitively, the fault of component c_1. Let $\gamma^{s_2} = comp(s_1 \mid\mid s_2, \gamma) \cap O_2$ and let $\gamma^{t_2} = comp(t, \gamma) \cap O_2$. Let t_1 be a strategy for component c_1 such that $comp(t_1, (\gamma \cup \gamma^{t_2}) \cap I_1) = comp(t, \gamma) \cap V_1$ holds.

Since $comp(t, \gamma) \cup \gamma' \models \varphi$ holds by assumption, $comp(t_1, (\gamma \cup \gamma^{t_2}) \cap I_1) \cup \gamma' \cup \gamma^{t_2} \cup (\gamma \cap (I_2 \setminus I_1)) \models \varphi$ thus follows. By assumption, strategy s_1 is remorsefree dominant for φ and c_1. Therefore, it follows that $comp(s_1, (\gamma \cup \gamma^{t_2}) \cap I_1) \cup \gamma' \cup \gamma^{t_2} \cup (\gamma \cap (I_2 \setminus I_1)) \models \varphi$ holds as well. By the semantics of conjunction, we thus, in particular, have $comp(s_1, (\gamma \cup \gamma^{t_2}) \cap I_1) \cup \gamma' \cup \gamma^{t_2} \cup (\gamma \cap (I_2 \setminus I_1)) \models \varphi_{live}$. The liveness part of φ and thus φ_{live} is only affected by the outputs of c_1 by assumption. Hence, $comp(s_1, (\gamma \cup \gamma^{t_2}) \cap I_1) \cap O_1 \models \varphi_{live}$ holds, i.e., the satisfaction of φ_{live} is guaranteed by $comp(s_1, (\gamma \cup \gamma^{t_2}) \cap I_1) \cap O_1$ irrespective of the valuations of variables outside of O_1. Since the inputs and outputs of a component are disjoint by definition, we have $I_1 \cap O_1 = \emptyset$ and thus, in particular, $comp(s_1, (\gamma \cup \gamma^{t_2}) \cap I_1) \cap O_1 = comp(s_1, (\gamma \cup \gamma^{s_2}) \cap I_1) \cap O_1$ follows with the definition of computations. Therefore, we have $comp(s_1, (\gamma \cup \gamma^{s_2}) \cap I_1) \cap O_1 \models \varphi_{live}$ and thus, in particular, $comp(s_1, (\gamma \cup \gamma^{s_2}) \cap I_1) \cup \gamma' \cup \gamma^{s_2} \cup (\gamma \cap (I_2 \setminus I_1)) \models \varphi_{live}$ holds. By definition of γ^{s_2}, we have $comp(s_1 || s_2, \gamma) \cap O_2 = \gamma^{s_2}$ and hence $comp(s_1, (\gamma \cup \gamma^{s_2}) \cap I_1) \cup \gamma^{s_2} \cup (\gamma \cap (I_2 \setminus I_1)) = comp(s_1 || s_2, \gamma)$ follows with the definition of computations. Thus, we have $comp(s_1 || s_2, \gamma) \cup \gamma' \models \varphi_{live}$, contradicting the assumption that $comp(s_1 || s_2, \gamma) \not\models \varphi$, while $comp(s_1 || s_2, \gamma) \models \varphi_{safe}$. □

Hence, if the considered specification is a safety property or if its liveness part is only affected by one of the components, we can synthesize dominant strategies for these components compositionally – and thus completely independently – while not losing remorsefree dominance of the parallel composition of the individual strategies. Therefore, the ranking function $rank_{syn}$ may assign the same value to several components if the specification ensures the above criterion for guaranteed compositionality.

In the following, we introduce the incremental synthesis algorithm that automatically derives strategies for all components in the order defined by the ranking function $rank_{syn}$ and thus defined by the synthesis order. We assume that a compositionality-preserving synthesis order is given and prove soundness of the algorithm under this assumption.

6.2.2. Incremental Synthesis Algorithm

In this section, we present the incremental synthesis algorithm. We assume that a decomposition of the system into components and a synthesis order defined by a ranking function $rank_{syn}$ are given. The incremental synthesis algorithm is described in Algorithm 6.1.

It expects an LTL specification φ as well as an array of components, representing the decomposition of the system, and a ranking function defining the synthesis order, i.e., mapping components to ranks, as input. The incremental synthesis algorithm proceeds *layer per layer*. A layer contains all components with the same rank in the synthesis order. The algorithm starts with the layer of components with the lowest rank. Then it considers one layer after the other in ascending order until reaching the layer with the highest rank in the synthesis order. To do so, we first order all components according to their synthesis rank and store them in the correct order in the array `orderedComponents` (line 1). The order of components of the same layer is irrelevant, and therefore, an arbitrary order can be chosen for them. We use variable `assumed` to store the parallel composition of all strategies that have been synthesized beforehand, i.e., the strategies for components of lower layers, while variable `layered` stores the parallel composition of all strategies on the same layer.

Algorithm 6.1: Incremental Synthesis

```
   Input: $\varphi$: LTL, C: Component[], rankSyn: (Component $\longrightarrow$ Int)
   Output: admissible: Bool, strategy: Strategy
1  orderedComponents ← orderComponents(C, rankSyn)
2  strategies ← []: List Strategy
3  assumed ← NULL: Strategy                          // strategy, initialized with null
4  for $i = 0$; $i <$ orderedComponents.$length$(); $i = i + 1$ do
5      layered ← NULL: Strategy                      // strategy, initialized with null
6      foreach $c \in$ orderedComponents[$i$] do
7          (adm, s) ← synthDominant($\varphi$, $c$.getInputs(), $c$.getOutputs(), assumed)
8          if adm then
9              strategies[$c$.getIndex()] ← s
10             if layered != NULL then
11                 layered ← layered || s
12             else
13                 layered ← s
14         else
15             return (false, NULL)
16     if assumed != NULL then
17         assumed ← assumed || layered
18     else
19         assumed ← layered
20 strategy ← compose(strategies)
21 return (true, strategy)
```

For each layer i of the synthesis order, we synthesize strategies $s_{i_1}, \dots, s_{i_k}$ for the k components $c_{i_k}, \dots, c_{i_k}$ of layer i such that, for each $j \in \{i_1, \dots, i_k\}$, the parallel composition of assumed and s_j is remorsefree dominant for the specification φ and the parallel composition of all components with lower ranks as well as the currently considered component c_j (line 7). If no previously synthesized strategies exist, assumed is NULL and therefore, we then synthesize a strategy s_j for component c_j such that only s_j is remorsefree dominant for φ and component c_j. We store the synthesized strategies at the corresponding positions of the array strategies (line 9) and store their parallel composition $s_{i_1} \mid\mid \dots \mid\mid s_{i_k}$ in layered (lines 10 to 13). Afterward, we store the parallel composition of assumed and layered in assumed to maintain the invariant that assumed stores the parallel composition of all strategies of components with lower ranks (lines 16 to 19). We continue until strategies for all components have been synthesized. If, for some component, the synthesis task synthDominant fails, i.e., if it is not admissible, the incremental synthesis algorithm aborts and indicates that the synthesis has failed with its return tuple (line 15). Otherwise, it composes all synthesized strategies according to the definition of the parallel composition of finite-state transducers (see Definition 2.12) and returns the resulting strategy (lines 20 and 21).

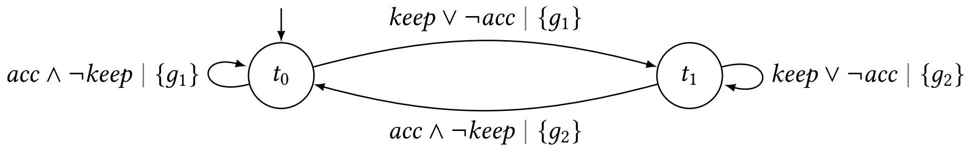

$acc \wedge \neg keep \mid \{g_1\}$
t_0
$keep \vee \neg acc \mid \{g_1\}$
$acc \wedge \neg keep \mid \{g_2\}$
t_1
$keep \vee \neg acc \mid \{g_2\}$

Figure 6.1.: Strategy s_{gear} for the gearing unit from the running example.

Intuitively, the incremental synthesis algorithm thus starts with synthesizing dominant strategies $s_{0_1}, \dots, s_{0_k}$ for the k components with the lowest rank. Since for these synthesis tasks `assumed` is still the empty strategy, `synthDominant` produces strategies that are on their own dominant, i.e., that do not rely on other strategies. Afterward, we synthesize dominant strategies $s_{1_1}, \dots, s_{1_{k'}}$ for the k' components with the next rank *under the assumption* of the parallel composition of $s_{0_1}, \dots, s_{0_k}$, i.e., under the assumption of the composed strategy $s_{0_1} || \dots || s_{0_k}$, which is stored in `assumed`. In particular, we seek for strategies $s_{1_1}, \dots, s_{1_{k'}}$ such that $s_{0_1} || \dots || s_{0_k} || s_{1_\ell}$ is dominant for φ and $c_{0_1} || \dots || c_{0_k} || c_{1_\ell}$, where ℓ is an index with $1 \leq \ell \leq k'$. As the synthesized strategies are always provided to the component with a higher rank via `assumed`, the algorithm continues accordingly until either a strategy for the last component of the layer with the highest rank is synthesized, or the synthesis task fails for some component.

Example 6.1. Reconsider the gearing unit and the acceleration unit of the autonomous car from the running example introduced in Section 6.1. Suppose that the acceleration unit has a higher rank than the gearing unit, i.e., we have $rank_{syn}(c_{gear}) < rank_{syn}(c_{acc})$. Assuming that the acceleration and gearing units are the only ones of the autonomous car we are interested in, Algorithm 6.1 starts with the gearing unit as the single process of the lowest rank in the synthesis order. Since c_{gear} is of the lowest rank, it does not assume any other strategies, and hence the incremental synthesis algorithm synthesizes a strategy s_{gear} for the gearing unit such that s_{gear} is dominant for the specification $\varphi_{gear} \wedge \varphi_{acc}$. Such a strategy adheres to the gearing restrictions, i.e., it uses the smaller gear when the car is accelerating and the higher one if it reaches a steady speed after accelerating. Figure 6.1 depicts a finite-state transducer representing such a dominant gearing strategy. Since c_{gear} is the only process of the lowest rank, it is the only process on this layer, and hence we only store s_{gear} in `assumed`.

Next, we consider the subsequent rank in the synthesis order, which is, since we are only interested in the gearing unit and the acceleration unit, the rank of process c_{acc}. Hence, Algorithm 6.1 synthesizes a strategy s_{acc} for c_{acc} such that $s_{gear} \,||\, s_{acc}$ is dominant for $\varphi_{gear} \wedge \varphi_{acc}$. A finite-state transducer representing such a strategy is depicted in Figure 6.2. Note that it is not winning for φ_{car}. If, for instance, the autonomous car always senses a curve in front of it, then the transducer will always stay in state t_1, thus always outputting *dec* and hence violating the requirement $\square\Diamond\, keep$. Furthermore, the requirement $\square((\neg in \wedge \neg ahead) \rightarrow \bigcirc\Diamond\, acc)$ can be violated if curves follow one another with only one step in between as well as when a curve is never left. This matches the observations about necessary assumptions for realizability of φ_{acc} from Section 6.1. Hence, the strategy s_{acc} is remorsefree dominant as it only violates the specification in situations in which no strategy at all can satisfy it. Since c_{acc} is the only component

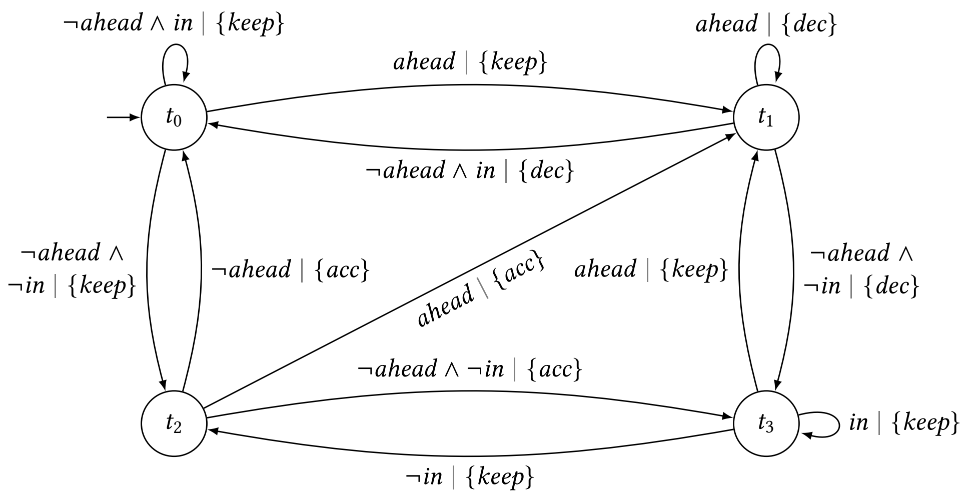


Figure 6.2.: Strategy s_{acc} for the acceleration unit from the running example.

of its rank, it is the only process on this layer, and therefore we only store the strategy s_{acc} in `assumed`. As no other processes are present in the considered model of the autonomous car, the algorithm returns the parallel composition $s_{acc} \mid\mid s_{gear}$ of the two strategies s_{gear} and s_{acc} for the gearing unit and the acceleration unit. △

In the following, we consider the soundness of the incremental synthesis algorithm. Clearly, due to the compositional synthesis of components with the same rank in the synthesis order, soundness highly depends on the provided ranking function $rank_{syn}$. As long as the synthesis order is compositionality-preserving as defined in the previous section, however, it follows from the construction of the algorithm as well as from Theorem 6.1 that the parallel composition of the strategies computed by Algorithm 6.1 is indeed dominant for φ and the overall system.

Theorem 6.2. *Let φ be an LTL formula over atomic propositions V. Let $\mathbb{D} = \langle c_1, \ldots, c_n \rangle$ be a decomposition of (I, O). Let $rank_{syn}$ be a compositionality-preserving function defining the synthesis order. Suppose that Algorithm 6.1 terminates with* $(\texttt{true}, s)$ for input φ, $\mathbb{D}$, and $rank_{syn}$. Then, strategy s is dominant for φ.

Proof. For the sake of readability, let $prev(i)$ denote the parallel composition of all components $c \in \mathbb{D}$ with a smaller rank than the components on the layer of the ordered components with index i, i.e., with $rank_{syn}(c) < rank_{syn}(c')$ for component c' on layer i.

First, observe that since Algorithm 6.1 terminates with $(\texttt{true}, s)$, line 15 is never reached and thus for each component of the decomposition, the synthesis task in line 7 succeeds. Hence, for each layer $\mathbb{L}$ of `orderedComponents`, the synthesized strategies of the components of $\mathbb{L}$ are stored in `layered` by successively building the parallel composition (see line 11). These strategies are then added to `assumed` by successively building the parallel composition (see line 17) and therefore it follows that, whenever the counter of the for-loop in line 4 is incremented, i.e., whenever a new layer is considered, `assumed` contains the parallel composition of the strategies of all components with a smaller rank.

Next, by definition of the synthesis task in line 7, we know that for every layer $\mathbb{L}$ of `orderedComponents` and every component $c_j \in \mathbb{L}$, it follows for the synthesized strategy s_j that `assumed` $|| s_j$ is dominant for φ and $prev(i) \mid\mid c_j$, where i is the index of layer $\mathbb{L}$. Let $\mathbb{L} = \{c_{j_1}, \ldots, c_{j_k}\}$. Since the ranking function $rank_{syn}$ is compositionality-preserving by assumption, it follows that (`assumed` $|| s_{j_1}) \mid\mid \ldots \mid\mid$ (`assumed` $|| s_{j_k})$ is remorsefree dominant for φ and $(prev(i) \mid\mid c_{j_1}) \mid\mid \ldots \mid\mid (prev(i) \mid\mid c_{j_k})$. Hence, since both `assumed` and $prev(i)$ coincide for all components c_{j_m} of layer $\mathbb{L}'$ by construction of Algorithm 6.1 as well as by definition of $prev(i)$, strategy `assumed` $|| s_{j_1} \mid\mid \ldots \mid\mid s_{j_k}$ is dominant for φ and $prev(i) \mid\mid (c_{j_1} \mid\mid \ldots \mid\mid c_{j_k})$.

Therefore, in particular for the layer $\mathbb{L} = \{c_{j_1}, \ldots, c_{j_k}\}$ with the highest rank, it holds that `assumed` $|| s_{j_1} \mid\mid \ldots \mid\mid s_{j_k}$ is remorsefree dominant for φ and $prev(i) \mid\mid (c_{j_1} \mid\mid \ldots \mid\mid c_{j_k})$. As argued above, `assumed` is the parallel composition of all strategies of components with a smaller rank than c_k, i.e., of all components $c' \in \mathbb{D}$ with $rank_{syn}(c') < rank_{syn}(c_k)$. Since $\mathbb{L}$ is the layer with the highest rank by assumption, `assumed` thus stores the parallel composition of the synthesized strategies of all components that do not lie in the layer with the highest rank. Hence, it follows immediately that the parallel composition of all synthesized strategies is dominant for φ and the parallel composition of all components. Since the synthesized strategies are stored in `strategies`, their parallel composition is stored in `strategy`. Since `strategy` is returned, we thus obtain that the returned strategy s, which is exactly $s_1 \mid\mid \ldots \mid\mid s_n$, is thus remorsefree dominant for φ and $c_1 \mid\mid \ldots \mid\mid c_n$. □

While soundness of incremental synthesis is thus guaranteed as long as the chosen synthesis order is compositionality-preserving, the success of incremental synthesis relies heavily on the choice of components, i.e., on the decomposition of the system. Incremental synthesis only terminates with a solution if all individual synthesis tasks succeed. Hence, the algorithm is only able to compute strategies for all components if the synthesis order guarantees the admissibility of every component when provided with the strategies of components with a lower rank. Thus, a clever decomposition of the system into components that takes the requirement of admissibility into account is crucial for the success and, in particular, the completeness of incremental synthesis. In the following sections, we thus introduce techniques for component selection that induce a synthesis order that ensure completeness of incremental synthesis.

6.3. Semantic Component Selection

In this section, we present an algorithm for selecting components as well as ordering them which is based on *semantic* dependencies between the output variables of the overall system. The algorithm directly induces a ranking function $rank_{syn}$ defining a compositionality-preserving synthesis order that ensures completeness of incremental synthesis.

We require specifications to be of the form $(\varphi^A_1 \land \cdots \land \varphi^A_\ell) \rightarrow (\varphi^G_1 \land \cdots \land \varphi^G_m)$, where the individual conjuncts φ^A_i and φ^G_j are conjunction-free and in negation normal form (NNF), i.e., negation is only applied to atomic propositions, in the following. Thus, a specification consists of *assumptions*, namely the formulas $\varphi^A_1, \ldots, \varphi^A_\ell$, and *guarantees*, namely the formulas $\varphi^G_1, \ldots, \varphi^G_m$. It is commonly considered to be a modeling flaw if assumptions can be violated by the system [KP10, BEJK14]. Then, a system strategy can satisfy the overall specification by violating the assumptions instead of satisfying the guarantees. Generally, however, assumptions are used to model restrictions of the system environment and thus it should not be possible for the system to violate them. In the following, we therefore suppose that an LTL specification $\varphi = (\varphi_1^A \land \dots \land \varphi_\ell^A) \rightarrow (\varphi_1^G \land \dots \land \varphi_m^G)$ is designed such that the system cannot satisfy it by violating the assumptions. In particular, we assume that if some strategy violates the assumptions on some input sequence, then all strategies do so. Observe that when considering dominant strategies instead of winning ones, assumptions can be treated as guarantees as long as the system cannot satisfy the specification by violating the assumptions:

Lemma 6.1. *Let $\varphi = (\varphi_1^A \land \ldots \land \varphi_\ell^A) \rightarrow (\varphi_1^G \land \ldots \land \varphi_m^G)$ be an LTL formula over atomic propositions V. Suppose that for all $\gamma \in (2^I)^\omega$, either $comp(s,\gamma) \models \varphi_1^A \land \ldots \land \varphi_\ell^A$ holds for all strategies s or $comp(s,\gamma) \not\models \varphi_1^A \land \ldots \land \varphi_\ell^A$ holds for all strategies s. Then, φ is admissible if, and only if, $\varphi_1^A \land \dots \land \varphi_\ell^A \land \varphi_1^G \land \dots \land \varphi_m^G$ is admissible.*

Proof. For the sake of readability, let $\varphi' = \varphi_1^A \land \dots \land \varphi_\ell^A \land \varphi_1^G \land \dots \land \varphi_m^G$. First, let φ be admissible. Then, there exists some dominant strategy s for φ. We claim that s is dominant for φ' as well. Suppose that it is not. Then, there exists some input sequence $\gamma \in (2^I)^\omega$ and some alternative strategy t such that $comp(s,\gamma) \not\models \varphi'$ holds, while we have $comp(t,\gamma) \models \varphi'$. By the semantics of conjunction and implication, it then follows that $comp(t,\gamma) \models \varphi$ holds as well. Since s is dominant for φ by assumption, $comp(s,\gamma) \models \varphi$ follows. If $comp(s,\gamma) \models \varphi_1^A \land \dots \land \varphi_\ell^A$ holds, then it follows from the semantics of implication as well as the fact that $comp(s,\gamma) \models \varphi$ holds that we have $comp(s,\gamma) \models \varphi_1^G \land \dots \land \varphi_m^G$ as well, contradicting the assumption that $comp(s,\gamma) \not\models \varphi'$ holds. Hence, we have $comp(s,\gamma) \not\models \varphi_1^A \land \dots \land \varphi_\ell^A$. By assumption, we then have $comp(t',\gamma) \not\models \varphi_1^A \land \dots \land \varphi_\ell^A$ for all alternative strategies t' as well. Thus, in particular, $comp(t,\gamma) \not\models \varphi_1^A \land \dots \land \varphi_\ell^A$ holds. Consequently, $comp(t,\gamma) \not\models \varphi'$ follows with the semantics of conjunction, contradicting the assumption that $comp(t,\gamma) \models \varphi'$ holds.

Second, let φ' be admissible. Then, there exists some dominant strategy s for φ'. By definition of remorsefree dominance, for every input sequence $\gamma \in (2^I)^\omega$, either $comp(s,\gamma) \models \varphi'$ holds or we have $comp(t,\gamma) \not\models \varphi'$ for all alternative strategies t. If the former is the case, it follows immediately from the semantics of conjunction and implication that $comp(s,\gamma) \models \varphi$ holds as well. Suppose that the latter holds. If there exists some strategy t with $comp(t,\gamma) \not\models \varphi_1^A \land \ldots \land \varphi_\ell^A$, then we have $comp(s,\gamma) \not\models \varphi_1^A \land \ldots \land \varphi_\ell^A$ for s by assumption as well. By the semantics of conjunction, it thus follows that $comp(s,\gamma) \models \varphi$ holds. Otherwise, if we have $comp(t,\gamma) \models \varphi_1^A \land \ldots \land \varphi_\ell^A$ for all strategies t, then, it follows from the assumption that $comp(t,\gamma) \not\models \varphi'$ holds for all alternative strategies t as well as the semantics of conjunction that $comp(t,\gamma) \not\models \varphi_1^G \land \ldots \land \varphi_m^G$ holds for all alternative strategies t. But then we have $comp(t,\gamma) \not\models \varphi$ for all alternative strategies t by the semantics of implication as well. Hence, s is also dominant for φ in this case. Consequently, it follows that s is remorsefree dominant for φ and hence φ is admissible. □

Thus, since admissibility is not influenced by replacing the implication with an conjunction as long as the system is not able to satisfy the specification by violating the assumptions, we assume in the following that specifications are of the form $\varphi_1^A \land \dots \land \varphi_n^A \land \varphi_1^G \land \dots \land \varphi_m^G$. That is, a specification consists of conjunction-free conjuncts, which are in negation normal form.

Based on this assumption, we introduce a decomposition algorithm that identifies equivalence classes of variables based on semantic dependencies between them. These equivalence classes then constitute the components. We first define two types of semantic dependencies between variables. Afterward, we introduce a decomposition algorithm based on these dependencies and study its soundness and completeness.

6.3.1. Semantic Dependencies

Intuitively, a variable $u \in V$ depends on the current or future valuation of a variable $v \in V$ if changing the valuation of u yields a violation of the specification φ that can be fixed by changing the valuation of v at the exact same point in time or a strictly later point in time, respectively. The change of the valuation of v needs to be necessary for the satisfaction of φ in the sense that not changing it would not yield the satisfaction of φ.

Formally, a semantic dependency is defined via so-called *minimal satisfying change sets.* For a specification φ, a sequence $\sigma \in (2^V)^\omega$ of variable valuations such that $\sigma \not\models \varphi$ holds, a variable $u \in V$, and a point in time $k \geq 0$, a satisfying change set is a pair (P, F) of subsets of output variables. Intuitively, P and F capture the variables that need to be changed with respect to σ in order to satisfy φ. Here, P captures the output variables that need to be changed in σ at point in time k, while F captures the output variables that need to be changed at a later point in time $j > k$. A minimal satisfying change set is then a satisfying change set (P, F) such that for not subsets P', F' of P and F the pair (P', F') is a satisfying change set. Formally, minimal satisfying change sets are defined as follows:

Definition 6.2 (Minimal Satisfying Change Set)**.**
Let φ be an LTL formula over atomic propositions V. Let $\sigma \in (2^V)^\omega$ be an infinite sequence such that $\sigma \not\models \varphi$ holds. Let $u \in V$ be a variable and let $k \geq 0$ be a point in time. For sets $P \subseteq V \setminus \{u\}$ and $F \subseteq V$, let $\Sigma^{P,F}$ be the set of sequences $\sigma' \in (2^V)^\omega$ such that

- $\sigma'_j = \sigma_j$ holds for all points in time j with $0 \leq j < k$,
- for all variables $v \in P$, we have $v \in \sigma'_k$ if, and only if, $v \notin \sigma_k$ holds,
- for all variables $v \in V \setminus P$, we have $v \in \sigma'_k$ if, and only if $v \in \sigma_k$,
- for all variables $v \in F$, there exists some point in time j with $j > k$ such that we have $v \in \sigma'_j$ if, and only if, $v \notin \sigma_j$ holds, and
- for all variables $v \in V \setminus F$ and all points in time j with $j > k$, we have $v \in \sigma'_j$ if, and only if, $v \in \sigma_j$ holds.

If there is a sequence $\sigma' \in \Sigma^{P,F}$ such that $\sigma' \models \varphi$ holds, then (P, F) is called *satisfying change set* for φ, σ, u, and k. If, additionally, for all $P' \subsetneq P$ and all $F' \subsetneq F$, we have $\sigma'' \not\models \varphi$ for all $\sigma'' \in \Sigma^{P',F'}$, then (P, F) is called *minimal satisfying change set* for φ, σ, u, and k.

A minimal satisfying change set is not necessarily unique. Furthermore, a pair (P, F) can be a minimal satisfying change set although there exist smaller sets P', F', i.e., with $|P'| < |P|$ and

$|F'| < |F|$, but $P' \nsubseteq P$ or $F' \nsubseteq F$ such that (P', F') is a satisfying change set. Hence, a minimal satisfying change set is not necessarily minimal in its size but only in terms of subsets.

Based on minimal satisfying change sets, we now define *semantic dependencies* between variables of the system. Intuitively, an output variable $u \in O$ depends on all of its minimal satisfying change sets (P, F) if, first of all, changing the valuation of u at a single point in time in a sequence satisfying the given LTL specification φ, yields a violation of φ. That is, if a sequence $\sigma \in (2^V)^\omega$ satisfies φ, while a sequence $\sigma' \in (2^V)^\omega$ obtained from solely changing the valuation of u at a single point in time $k \geq 0$ violates φ, and if changing variables according to (P, F) while maintaining the valuations of the variables in all other situations results in satisfaction of φ, then u depends on the pair (P, F). Since P defines the variables whose valuations need to be changed at the particular point in time k for which (P, F) is a minimal satisfying change set, u depends on the current valuation of all variables $v \in P$. Since F defines the variables whose valuations need to be changed at some later point in time, u depends on the future valuation of all variables $v \in F$. Formally, we define semantic dependencies as follows:

Definition 6.3 (Semantic Dependencies)**.**
Let φ be an LTL formula over atomic propositions V. Let $u \in O$ and let $k \geq 0$. Let $\sigma, \sigma' \in (2^V)^\omega$ be sequences such that $\sigma_k \cap \{u\} \neq \sigma'_k \cap \{u\}$ holds, while we have $\sigma_k \cap (V \setminus \{u\}) = \sigma'_k \cap (V \setminus \{u\})$ and while $\sigma_j = \sigma'_j$ holds for all $j \geq 0$ with $j \neq k$, and such that we have $\sigma \models \varphi$ and $\sigma' \not\models \varphi$. Let $\mathcal{M}$ be the set of minimal satisfying change sets (P, F) for φ, σ', u, and k. For all $(P, F) \in \mathcal{M}$ we say that *u depends semantically on* (P, F). If either $(P \cup F) \cap I = \emptyset$ or $(P' \cup F') \cap I \neq \emptyset$ holds for all $(P', F') \in \mathcal{M}$, then we say, for all variables $v \in P$, that *u depends semantically on the current valuation of v* and, for all variables $v \in V$, that *u depends semantically on the future valuation of v*. If $\mathcal{M} \neq \emptyset$ and $(P' \cup F') \cap I \neq \emptyset$ holds for all $(P', F') \in \mathcal{M}$, then we say that *u depends semantically on the input.*

If an output variable $u \in O$ depends semantically on the current valuation of another variable $v \in V$, we also call this a *present dependency* from u to v. If it depends semantically on the future valuation of a variable $v \in V$, we also call this a *future dependency* from u to v. Note that an output variable can also depend semantically on its own future valuation.

Example 6.2. Consider the autonomous car from the running example from Section 6.1 and its specification φ_{car}. It induces, among others, a present dependency from variable *acc* to variable *dec*: consider a scenario where the street does not contain any curve, i.e., neither *ahead* nor *in* is ever set to true. Since the car is neither in a curve nor directly before a curve at any point in time, the acceleration unit only needs to ensure mutual exclusion of *acc*, *dec*, and *keep* as well as that always either *acc*, *dec*, or *keep* is *true* and that both *acc* and *keep* are played infinitely often. Therefore, the sequence $\sigma = \{g_1, dec\}(\{g_2, keep\}\{g_2, acc\})^\omega$, for instance, satisfies φ_{car} since it ensures the above requirements for the acceleration unit as well as the requirements of the gearing unit defined by φ_{gear} and lies in the described scenario of a street that does contain curves. Consider the sequence $\sigma' = \{g_1, dec, acc\}(\{g_2, keep\}\{g_2, acc\})^\omega$. It differs from σ solely in the valuation of output variable *acc* in the very first time step. Furthermore, it violates mutual exclusion of *acc* and *dec*. Hence, it violates the specification φ_{acc} of the acceleration unit, and therefore it also violates the full specification φ_{car} of the autonomous car. Yet, mutual exclusion is the only requirement of the car that is violated by σ'. Thus, setting *dec* to *false* in the first time step yields a sequence $\sigma'' = \{g_1, \mathit{dec}, \mathit{acc}\}(\{g_2, \mathit{keep}\}\{g_2, \mathit{acc}\})^\omega$ that again satisfies φ_{car}. Therefore, for $P = \{\mathit{dec}\}$ and $F = \emptyset$, the pair (P, F) is a satisfying change set with respect to φ_{car}, σ', *acc*, and 0. Furthermore, it is minimal since the only pair (P', F') with $P' \subseteq P$ and $F' \subseteq F$, namely $(\emptyset, \emptyset)$, does not allow for a sequence that satisfies φ_{car} as it does not permit any changes in variable valuations. Thus, variable *acc* depends semantically on (P, F), and, in particular, *acc* depends semantically on the current valuation of variable *dec*.

Next, consider the sequence $\sigma = \{\mathit{dec}, g_1\}\{\mathit{in}, \mathit{acc}, g_1\}\{\mathit{in}, \mathit{keep}, g_2\}^\omega$. It is unrealistic as it models that we are in a curve in the second time step without ever sensing that a curve is ahead. However, as we did not exclude such unrealistic situations via assumptions in the specification of the car for simplicity, σ is a valid input sequence. Clearly, σ satisfies that exactly one of g_1 and g_2 as well as *acc*, *dec*, and *keep* is set to *true* at each time step. Furthermore, *keep* occurs infinitely often in σ. If the car is in a curve, it does not accelerate in the next time step. If it is neither in a curve nor senses one, then it accelerates eventually. Since the car never senses that a curve is ahead, the first conjunct of φ_{acc} is also satisfied. Moreover, we use the second gear when reaching a steady speed after accelerating and never accelerate for two consecutive steps. Hence, σ satisfies φ_{car}. Consider the sequence $\sigma' = \{\mathit{dec}, g_1\}\{\mathit{in}, \mathit{acc}, \mathit{keep}, g_1\}\{\mathit{in}, \mathit{keep}, g_2\}^\omega$, which differs from σ only in the valuation of *keep* at the second point in time and clearly violates mutual exclusion of *acc* and *keep*. Setting *acc* to *false* at the second point in time does not suffice to satisfy φ_{car} again as then the car does not accelerate eventually after neither being in a curve, nor sensing a curve in the very first time step. However, as the car is always in a curve from the second point in time on – which is, again, not a realistic situation but which we did not exclude and thus need to consider – the car is never allowed to accelerate after the second point in time. Hence, φ_{car} can only be satisfied by changing the valuation of input variable *in* as well as output variables *acc* and *keep*. For instance, sequence $\sigma'' = \{\mathit{dec}, g_1\}\{\mathit{keep}, g_1\}\{\mathit{in}, \mathit{acc}, g_2\}\{\mathit{in}, \mathit{keep}, g_2\}^\omega$ satisfies φ_{car} again but differs from σ' in the valuations of both *in* and *acc* at the second point in time as well as in the valuations of both *keep* and *acc* at the third point in time. As these changes are minimal, we thus obtain present dependencies from *keep* to both *in* and *acc* as well as future dependencies from *keep* to both *keep* and *acc*. Moreover, since a change in the valuation of an input variable is required, *keep* depends on the input. $\triangle$

If a variable $u \in V$ depends semantically on some pair (P, F) with $F \neq \emptyset$, then, intuitively, a strategy that defines the behavior of u most likely has to predict the future valuations of the variables in F to determine the correct valuation of u at a certain point in time. Irrespective of whether v is an input or output variable of the overall system, this prevents the existence of a dominant strategy for a component with output variable u that does not also control all variables in F. Furthermore, in our setting, strategies cannot react directly to an input since we model strategies with Moore transducers. Thus, the variables in P may prevent admissibility as well. Hence, if u depends on either the current or the future valuation of some variable $v \in O$ or on the input, then there might not exist a dominant strategy for the component controlling u if it does not also control the variables u depends on. If such dependencies do not exist, in contrast, admissibility of the specification for the component controlling u is guaranteed.

To show this formally, we construct a dominant strategy for the behavior of u by choosing arbitrary output sequences for input sequences for which φ is not satisfiable and output sequences that yield satisfaction of φ for input sequences for which φ is satisfiable. In general, these output sequences may not be computable by a strategy. However, this can only be the case if a strategy needs to predict the valuations of variables outside its control, and this need is precisely what is captured by semantic present and future dependencies:

Theorem 6.3. *Let φ be an LTL formula over atomic propositions V. Let c_i be some component of the system with output variables $O_i \subseteq O$. If, for all $u \in O_i$, variable u does not depend semantically on the current or future valuation of a variable $v \in V \setminus O_i$, then φ is admissible for c_i.*

Proof. Let $\gamma \in (2^{I_i})^\omega$ be some infinite input sequence for c_i. If $\gamma \cup v \not\models \varphi$ holds for all $v \in (2^{O_i})^\omega$, then, intuitively, no strategy for c_i at all can satisfy the specification φ on input γ. Thus, a dominant strategy for c_i may behave arbitrarily on input γ. Otherwise, there exists some $v^\gamma \in (2^{O_i})^\omega$ such that $\gamma \cup v^\gamma \models \varphi$ holds. We claim that there exists a strategy s_i for c_i such that we have $comp(s_i, \gamma) \cap O_i = v^\gamma$ for all input sequences $\gamma \in (2^{I_i})^\omega$ for which there exists some $v^\gamma \in (2^{O_i})^\omega$ such that $\gamma \cup v^\gamma \models \varphi$ holds. If there are multiple sequences $v^\gamma \in (2^{O_i})^\omega$ such that $\gamma \cup v^\gamma \models \varphi$ holds, then the strategy s_i chooses one of these sequences.

Suppose that such a strategy does not exist. Then, by definition of strategies, it is not possible to determine in each time step the correct output based on the history of in- and outputs. Hence, there exist input sequences $\gamma, \gamma' \in (2^{I_i})^\omega$ with $\gamma \neq \gamma'$ and a point in time $k \geq 0$ such that γ and γ' agree up to point in time $k-1$, i.e., we have $\gamma_{|k} = \gamma'_{|k}$, while for all $v, v' \in (2^{O_i})^\omega$ with $\gamma \cup v \models \varphi$ and $\gamma' \cup v' \models \varphi$, we have that v and v' disagree at point in time k, i.e., we have $v_k \neq v'_k$. Without loss of generality, we assume that k is the smallest such point in time. Then, there exist sequences $v, v' \in (2^{O_i})^\omega$ with $\gamma \cup v \models \varphi$ and $\gamma' \cup v' \models \varphi$ that agree up to point in time $k-1$ but disagree at point in time k. Hence, we have $v_{|k} = v'_{|k}$ and $v_k \neq v'_k$. Let $v \in (2^{O_i})^\omega$ and $v' \in (2^{O_i})^\omega$ be these sequences. Then, by construction, we have $\gamma \cup v \models \varphi$, while $\gamma \cup v' \not\models \varphi$ holds. Furthermore, since all sequences $v'' \in (2^{O_i})^\omega$ with $\gamma' \cup v'' \models \varphi$ disagree with v at point in time k, the prefix of v' of length k, i.e., up to point in time k cannot be extended such that it, together with γ, satisfies φ. That is, for every $v'' \in (2^{O_i})^\omega$ with $v''_{|k} = v'_{|k}$, we have $\gamma \cup v'' \not\models \varphi$. Thus, by construction of v', it holds that for every $v'' \in (2^{O_i})^\omega$ that agrees with v up to point in time $k-1$ but disagrees with v at point in time k, we have $\gamma \cup v'' \not\models \varphi$.

Let $u \in O_i$ be some output variable of component c_i such that the sequences v and v' differ on u at point in time k. By construction, we have $\gamma \cup v \models \varphi$. Let $v'' \in (2^{O_i})^\omega$ be the sequence that we obtain from v by only changing the valuation of u, i.e., let v'' be the sequence with $v_k \cap \{u\} \neq v''_k \cap \{u\}$ as well as $v_k \cap (V \setminus \{u\}) = v'' \cap (V \setminus \{u\})$ and $v_k \cap \{u\}\ v_j = v''_j$ for all $j \geq 0$ with $j \neq k$. Then, since every sequence of output variables that agrees with v up to point in time $k-1$ but disagrees with v at point in time k violates φ when combined with γ as shown above, it follows immediately that $\gamma \cup v'' \not\models \varphi$ holds.

Let $P \subseteq V \setminus \{u\}$ be the set of variables on which the input sequences $\gamma \cup v''$ and $\gamma' \cup v'$ differ at point in time k, i.e., let $P = \{v \in V \setminus \{u\} \mid (\gamma_k \cup v''_k) \cap \{v\} \neq (\gamma'_k \cup v'_k) \cap \{v\}\}$. Similarly, let $F \subseteq V$ be the set of variables on which $\gamma \cup v''$ and $\gamma' \cup v'$ differ at some point in time $k' \geq 0$ with $k' > k$, i.e., let $F = \{v \in V \mid \exists k' > k.\ (\gamma_{k'} \cup v''_{k'}) \cap \{v\} \neq (\gamma'_{k'} \cup v'_{k'}) \cap \{v\}\}$. By construction of γ and γ' as well as of v' and v'', we have $\gamma_{|k} = \gamma'_{|k}$ as well as $v'_{|k} = v''_{|k}$. Hence $\gamma \cup v''$ and $\gamma' \cup v'$ agree up to point in time $k-1$. Furthermore, by construction of v' and v'', these sequences agree in the valuation of u at point in time k, i.e., we have $u \in v'_k$ if, and only if, $u \in v''_k$ holds. Additionally, we have $\gamma' \cup v' \models \varphi$ by construction of v'. Therefore, if follows immediately from Definition 6.2 that (P, F) is a satisfying change set for φ, $\gamma \cup v''$, u, and k. Without loss of generality, we assume that γ' and v'' are chosen such that (P, F) is also minimal.

Furthermore, since $\gamma \neq \gamma'$ holds by assumption, we have $(P \cup F) \cap I_i \neq \emptyset$. Thus, there exists some variable $v \in I_i$ with either $v \in P$ or $v \in F$. By definition of semantic dependencies, u thus depends semantically on either the current or the future valuation of v. Since, by definition of components, the set of input and output variables of c_i are disjoint, $v \in V \setminus O_i$ holds, contradicting that no output of c_i semantically depends on a variable that is not output of c_i.

Hence, there exists a strategy s_i for c_i such that we have $comp(s_i, \gamma) \cap O_i = v^\gamma$ for all input sequences $\gamma \in (2^{I_i})^\omega$ for which there exists some $v^\gamma \in (2^{O_i})^\omega$ such that $\gamma \cup v^\gamma \models \varphi$ holds. In the following, we show that s_i is dominant for φ and c_i. Suppose that it is not. Then, there exists an input sequence $\gamma \in (2^{I_i})^\omega$ such that $comp(s_i, \gamma) \not\models \varphi$ holds, while there exists an alternative strategy t_i for c_i with $comp(t_i, \gamma) \models \varphi$. Clearly, φ can be satisfied by a sequence that agrees with γ on the valuations of the variables in I_i since $comp(t_i, \gamma) \models \varphi$ holds and we have $comp(t_i, \gamma) \cap I_i = \gamma$ by definition of computations. Thus, by construction of s_i, we have $comp(s_i, \gamma) \cap O_i = v^\gamma$, where $v^\gamma \in (2^{O_i})^\omega$ is a sequence such that $\gamma \cup v^\gamma \models \varphi$ holds. By definition of computations, however, we have $comp(s_i, \gamma) = \gamma \cup (comp(s_i, \gamma) \cap O_i$ and thus $comp(s_i, \gamma) = \gamma \cup v^\gamma$ holds; contradicting that we have $comp(s_i, \gamma) \not\models \varphi$. □

Utilizing the above result, we introduce an algorithm for identifying suitable components for incremental synthesis as well as a corresponding ranking function $rank_{syn}$, which defines a synthesis order, based on semantic dependencies in the subsequent section. The decomposition and the synthesis order ensure soundness of incremental synthesis.

6.3.2. Semantic Decomposition Algorithm

The semantic decomposition algorithm is based on determining semantic dependencies between output variables of the system. It then computes components by analyzing the dependencies and by maximizing the number of components while maintaining admissibility. To determine which output variables of the system need to be contained in the same component in order to ensure admissibility, we build the *semantic dependency graph* of the variables of the system based on their semantic dependencies:

Definition 6.4 (Semantic Dependency Graph).
Let φ be an LTL formula over atomic propositions V. The *semantic dependency graph* $\mathcal{D}^{sem}_\varphi$ of φ is defined by $\mathcal{D}^{sem}_\varphi = (\mathcal{V}^{sem}, \mathcal{E}^{sem})$ with $\mathcal{V}^{sem} = V$ and $\mathcal{E}^{sem} = \mathcal{E}^{sem}_P \cup \mathcal{E}^{sem}_F \cup \mathcal{E}^{sem}_I$, where $(u, v) \in \mathcal{E}^{sem}_P$ holds if, and only if, $u, v \in O$ and u depends semantically on the current valuation of v, where $(u, v) \in \mathcal{E}^{sem}_F$ holds if, and only if, $u, v \in O$ and u depends semantically on the future valuation of v, and where $(u, v) \in \mathcal{E}^{sem}_I$ holds, if, and only if, $u \in O$, $v \in I$ and u depends semantically on v.

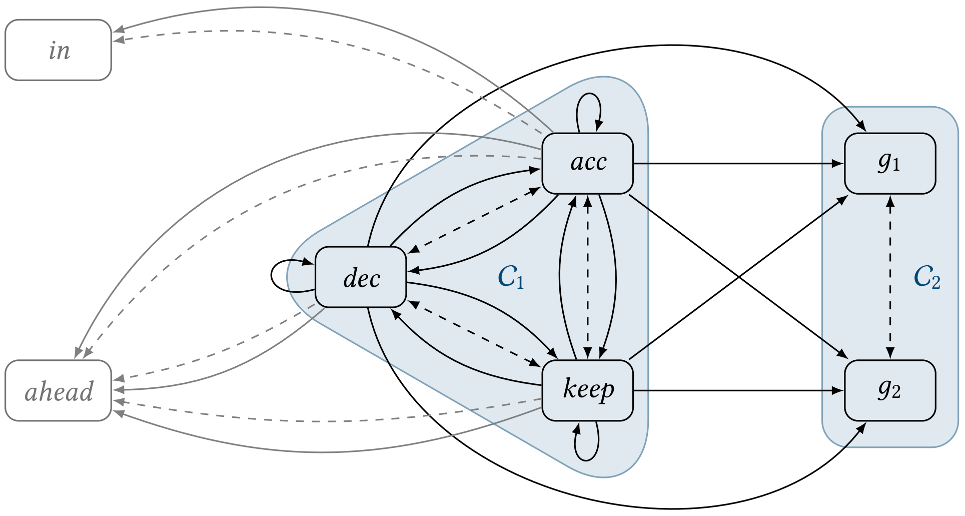


Figure 6.3.: Semantic dependency graph $\mathcal{D}^{sem}_{\varphi_{car}}$ induced by the specification φ_{car} of the running example. Nodes representing input variables and their edges are depicted in gray. For the sake of readability, we omit edges from *dec* and *keep* to *in* as well as edges from *dec* to *in*. Dashed edges represent present dependencies, solid ones represent future dependencies. The strongly connected components of the output variables are highlighted in blue.

To identify suitable components of the system for incremental synthesis, we now proceed in two steps: first, we eliminate vertices of $\mathcal{D}^{sem}_{\varphi}$ that represent input variables since only output variables of the system define components. Second, we compute the set C of strongly connected components of $\mathcal{D}^{sem}_{\varphi}$. The strongly connected components of $\mathcal{D}^{sem}_{\varphi}$ then define the components of the system: if $C = \{C_1, \dots, C_n\}$ holds, then we obtain n components $c_1, \dots, c_n$ such that, for every c_i, the set O_i of output variables is defined by $O_i = C_i$.

Example 6.3. Consider the autonomous car from Section 6.1. Its specification φ_{car} induces the semantic dependency graph $\mathcal{D}^{sem}_{\varphi_{car}}$ depicted in Figure 6.3. For the sake of readability, we omit edges from *dec* and *keep* to *in* as well as edges from *dec* to *in*. Note, however, that there also exist present and future dependencies from *dec* and *keep* to *in* as, for instance, due to the present dependency from *keep* to *in* which is explained in Example 6.2. When only considering the nodes representing output variables of the system, $\mathcal{D}^{sem}_{\varphi_{car}}$ contains two strongly connected components C_1 and C_2 – highlighted in blue – and thus induces two components c_1 and c_2 with $O_1 = \{acc, dec, keep\}$ and $O_2 = \{g_1, g_2\}$. Hence, we exactly identify the acceleration unit and the gearing unit of the autonomous car as components. △

In addition to the decomposition, the semantic dependency graph also induces the synthesis order by analyzing edges that connect strongly connected components and, thus, system components. Let C be the set of strongly connected components of $\mathcal{D}^{sem}_{\varphi}$. Let $C^0 \subseteq C$ be the set of strongly connected components such that, for all components $C_j \in C^0$ and all variables $u \in C_j$, we have $v \in C_j$ for all variables $v \in \mathcal{V}^{sem}$ with $(v, u) \in \mathcal{E}^{sem}$. That is, C^0 is, intuitively, the set of all strongly connected components of the semantic dependency graph $\mathcal{D}^{sem}_{\varphi}$ that do not have any incoming edges. For $i \in \mathbb{N}_0$ with $i > 0$, let $C^i \subseteq C$ be the set of strongly connected components such that (i) $C^i \cap C^\ell = \emptyset$ holds for all $\ell \in \mathbb{N}_0$ with $0 \leq \ell < i$, and, (ii) for all components $C_j \in C^i$ and all variables $u \in C_j$, we have, for all variables $v \in \mathcal{V}^{sem}$ with $(v, u) \in \mathcal{E}^{sem}$, either $v \in C_j$ or $v \in C_m$ for some component C_m with $C_m \in C^\ell$, where $\ell \in \mathbb{N}_0$ with $0 \leq \ell < j$. Hence, intuitively, C^i is the set of all strongly connected components whose incoming edges all originate from a node that lies in a strongly connected component with lower index $\ell < i$ and that do not already lie in a strongly connected component with a lower index. Note that every strongly connected component $C_j \in C$ then lies in exactly one set $C^i \subseteq C$.

We then obtain the synthesis order for the components as follows: we define a ranking function $rank_{syn}$ for the system components such that for all strongly connected components $C_i, C_j \in C$ representing system components c_i and c_j, respectively, with $C_i \in C^k$ and $C_j \in C^\ell$, we have (i) if $k < \ell$ holds, then we have $rank_{syn}(c_i) > rank_{syn}(c_j)$, and (ii) if $k > \ell$ holds, then we have $rank_{syn}(c_i) < rank_{syn}(c_j)$, and (iii) if $k = \ell$ holds and φ is a safety property or only of the components affects the liveness part of φ, then we have $rank_{syn}(c_i) = rank_{syn}(c_j)$, and (iv) otherwise we choose an arbitrary ordering of c_i and c_j, i.e., we choose either $rank_{syn}(c_i) < rank_{syn}(c_j)$ or $rank_{syn}(c_j) < rank_{syn}(c_i)$.

We can, for instance, compute a ranking function $rank_{syn}$ that satisfies these properties as follows. Let $m \geq 0$ be the highest index of a non-empty subset constructed as above, i.e., let $m = \max \left\{ i \geq 0 \mid C^i \neq \emptyset \right\}$. We first check whether φ is a safety property or whether its liveness part is only affected by a single component. If this is the case, then, for all strongly connected components $C_j \in C$ and thus for all components c_j, we assign $rank_{syn}(c_j) = m - i$ if $C_j \in C^i$ holds. Since, as argued above, every strongly connected component lies in exactly one subset of connected components, as computed above, the assignment is unique. Otherwise, i.e., if φ is a liveness property, where the liveness part is affected by more than one component, then we proceed as follows. For all non-empty subsets $C^i \subseteq C$, we assign the ranks $m - i + \sum_{0 \leq \ell < i}(|C^\ell| - 1)$ to $m - i + |C^i| - 1 + \sum_{0 \leq \ell < i}(|C^\ell - 1)|$ to the system components c_j representing the respective strongly connected components $C_j \in C^i$. Which of the components receives which rank is insignificant as, by construction of the components, they do not depend on each other.

Example 6.4. Reconsider the autonomous car from Section 6.1 with specification φ_{car} and its semantic dependency graph $\mathcal{D}^{sem}_{\varphi_{car}}$ depicted in Figure 6.3. It induces two components c_1 and c_2 representing the strongly connected components C_1 and C_2. We obtain the subsets $C^0 = \{C_1\}$ and $C^1 = \{C_2\}$. Hence, the ranking function $rank_{syn}$ ensures that $rank_{syn}(c_1) > rank_{syn}(c_2)$ holds and thus we obtain the synthesis order $c_2 <_{syn} c_1$. This matches the observation from Section 6.1 that a strategy for the gearing unit must be synthesized first. △

From the construction of the components as well as of the ranking function $rank_{syn}$, it follows that the resulting synthesis order is compositionality-preserving since the semantic decomposition algorithm only assigns the same rank to system components if either φ is a safety property, or its liveness part is only affected by a single component:

Lemma 6.2. *Let φ be an LTL formula over atomic propositions V. Let $\mathbb{D}$ and $rank_{syn}$ be the decomposition and the ranking function computed with semantic decomposition algorithm. Then, the synthesis order $<_{syn}$ induced by $\mathbb{D}$ and $rank_{syn}$ is compositionality-preserving.*

Together with Theorem 6.2, it thus follows immediately that incremental synthesis is sound when we compute the decomposition as well as the synthesis order with the semantic decomposition algorithm described in this section:

Corollary 6.1. *Let φ be an LTL formula over atomic propositions V. Let $\mathbb{D} = \langle c_1, \dots, c_n \rangle$ and $rank_{syn}$ be the decomposition and the ranking function computed with the semantic decomposition algorithm. Suppose that Algorithm 6.1 returns* $(\texttt{true}, s)$ for input φ, $\mathbb{D}$, and $rank_{syn}$. Then, strategy s is dominant for φ.

In the following, we study completeness of incremental synthesis when computing the system components and the synthesis order with the semantic decomposition algorithm. First, we focus on LTL specifications that do not induce semantic dependencies of output variables $u \in O$ to input variables $v \in I$ of the system. For such specifications, the success of the individual synthesis tasks in Algorithm 6.1 is guaranteed if the decomposition of the system and the synthesis order are computed with the semantic decomposition algorithm:

Lemma 6.3. *Let φ be an LTL formula over atomic propositions V. Let $\mathbb{D} = \langle c_1, \dots, c_n \rangle$ and $rank_{syn}$ be the decomposition and the ranking function computed with semantic decomposition algorithm. If, for all $u \in O$, output u does not depend semantically on an input variable $v \in I$, then Algorithm 6.1 returns* $(\texttt{true}, s)$ for input φ, $\mathbb{D}$, and $rank_{syn}$.

Proof. By construction, the components $c_1, \dots, c_n$ obtained with the semantic decomposition algorithm are built from the n strongly connected components $C_1, \dots, C_n$ of the semantic dependency graph $\mathcal{D}^{sem}_{\varphi}$ for φ. Thus, by definition of strongly connected components, there are no cyclic dependencies between system components. Furthermore, by construction of the ranking function $rank_{syn}$, we have $C_i \in C^m$ for all components c_i of the lowest rank, i.e., for all components $c_i \in \mathbb{L}_0$, where m is the highest index of a non-empty subset constructed as above, i.e., let $m = \max \left\{ i \geq 0 \mid C^i \neq \emptyset \right\}$. Thus, by definition of the subsets C^ℓ of strongly connected components, no output variable of a component c_i of the lowest rank is the source of an edge in $\mathcal{E}^{sem}$ to an output variable $v \in O \setminus O_i$ of the system that is no output of c_i. Hence, in particular, no output variable of c_i depends semantically on an output variable of the system. By construction, all output variables of components are output variables of the overall system. Furthermore, no output variable of the system depends semantically on some input variable of the system by assumption. Therefore, it follows that no output variable of a component c_i of the lowest rank depends semantically on any variable outside of O_i, i.e., outside of the outputs of c_i, at any point in time. Consequently, φ is admissible for φ and for all components c_i of the lowest rank by Theorem 6.3.

Next, let c_i be some component in $\mathbb{D}$ with non-lowest rank, i.e., a component of a layer $\mathbb{L}_j$ with $j > 0$. Let $prev(j)$ denote the parallel composition of all components c_ℓ with a smaller rank than the components on the layer $\mathbb{L}_j$ with index j, i.e., with $rank_{syn}(c_\ell) < rank_{syn}(c_i)$. Then, Algorithm 6.1 already computed strategies for all components c_ℓ with smaller rank. Let $prevStrat(i)$ denote their parallel composition. By construction of Algorithm 6.1, $prevStrat(i)$ is dominant for φ and $prev(j)$. By construction of the subsets $C^x \subset C$ of the strongly connected components, all components c_k for which there exists an edge $(u, v) \in \mathcal{E}^{sem}$ with $u \in c_i$ and

$v \in c_k$, i.e., an edge from c_i to c_k, lie in the subset $C^{i+1} \subseteq C$ of strongly connected components. Thus, by construction of the ranking function, we have $rank_{syn}(c_k) < rank_{syn}(c_i)$ for all such components c_k. Hence, since no output variable of the system depends on an input variable of the system by assumption, it thus follows that all output variables of component c_i may only depend semantically on output variables of components with a smaller rank. In Algorithm 6.1, we try to synthesize a strategy s_i for c_i such that $prevStrat(i) \,||\, s$ is dominant for φ and $prev(j) \,||\, c_i$. As shown above and by construction of the sets $C^x \subset C$, no output variable of the component $prev(j) \,||\, c_i$ depends semantically on a variable outside of the outputs of $prev(j) \,||\, c_i$. Thus, it follows with Theorem 6.3 that φ is admissible for $prev(j) \,||\, c_i$.

Since Algorithm 6.1 restricts the dominant strategy for φ and $prev(j) \,||\, c_i$, however, by fixing the strategies for the components in $prev(j)$, it does not follow immediately that the synthesis task for c_i in line 7 succeeds. In particular, it remains to show that there also exists a strategy s_i for c_i such that $prevStrat(i) \,||\, s_i$ is dominant for φ and $prev(j) \,||\, c_i$. Let t be a dominant strategy for φ and $prev(j) \,||\, c_i$. Since φ is admissible for $prev(j) \,||\, c_i$, such a strategy is guaranteed to exist. For the sake of readability, let $O_{prev} = \bigcup_{c_k \in prev(j)} O_k$. Let t_i be a strategy for c_i such that $comp(t_i, \gamma \cup \gamma') \cap O_i = comp(t, \gamma) \cap O_i$ holds for all sequences $\gamma \in (2^{V \setminus (O_i \cup O_{prev})})^\omega$ and $\gamma' \in (2^{O_{prev}})^\omega$. Hence, intuitively, t_i produces the same outputs for component c_i as t does, irrespective of the valuations of the output variables of the other processes. Similarly, let t_{prev} be a strategy for $prev(j)$ such that $comp(t_{prev}, \gamma \cup \gamma') \cap O_{prev} = comp(t, \gamma) \cap O_{prev}$ holds for all sequences $\gamma \in (2^{V \setminus (O_i \cup O_{prev})})^\omega$ and $\gamma' \in (2^{O_i})^\omega$.

We claim that $prevStrat(i) \,||\, t_i$ is dominant for φ and $prev(j) \,||\, c_i$. Suppose that it is not. Then, there exists some input sequence $\gamma \in (2^{V \setminus (O_i \cup O_{prev})})^\omega$ and some alternative strategy t' for $prev(j) \,||\, c_i$ such that $comp(prevStrat(i) \,||\, t_i, \gamma) \not\models \varphi$ holds, while we have $comp(t', \gamma) \models \varphi$. Since t is a dominant strategy for φ by assumption, we then have $comp(t, \gamma) \models \varphi$ as well. For the sake of readability, let

$$\begin{aligned} \gamma^i &= comp(prevStrat(i) \,||\, t_i, \gamma) \cap O_i \\ \gamma^{prev} &= comp(prevStrat(i) \,||\, t_i, \gamma) \cap O_{prev} \end{aligned}$$

Then, $comp(prevStrat(i) \,||\, t_i, \gamma) = comp(prevStrat(i), \gamma \cup \gamma^i)$ holds by construction of γ^i and by definition of computations of strategies. Therefore, $comp(prevStrat(i), \gamma \cup \gamma^i) \not\models \varphi$ follows. As shown above, strategy $prevStrat(i)$ is dominant for φ and $prev(j)$ and therefore, in particular, $comp(t_{prev}, \gamma \cup \gamma^i) \not\models \varphi$ holds as well. By construction of the strategy t_{prev}, we have

$$comp(t_{prev}, \gamma \cup \gamma^i) \cap O_{prev} = comp(t, \gamma) \cap O_{prev}.$$

Furthermore, by definition of computations, $comp(t_{prev}, \gamma \cup \gamma^i) \cap (V \setminus O_{prev}) = \gamma \cup \gamma^i$ holds and thus we obtain $comp(t_{prev}, \gamma \cup \gamma^i) \cap (V \setminus O_i) = comp(t, \gamma) \cap (V \setminus O_i)$. Moreover, we have $comp(t_{prev}, \gamma \cup \gamma^i) \cap O_i = \gamma^i$ and hence

$$comp(t_{prev}, \gamma \cup \gamma^i) \cap O_i = comp(prevStrat(i) \,||\, t_i, \gamma) \cap O_i$$

follows with the definition of the sequence γ^i. Note that $comp(prevStrat(i) \,||\, t_i, \gamma) \cap O_i$ is defined by strategy t_i. Therefore, $comp(prevStrat(i) \,||\, t_i, \gamma) \cap O_i = comp(t_i, \gamma \cup \gamma^{prev}) \cap O_i$ holds by construction of the sequence γ^{prev}. Since we have $comp(t_i, \gamma \cup \gamma^{prev}) \cap O_i = comp(t, \gamma) \cap O_i$ by definition of the strategy t_i, it thus follows that

$$comp(prevStrat(i) \mid\mid t_i, \gamma) \cap O_i = comp(t, \gamma) \cap O_i$$

holds. Combining the results, we thus obtain $comp(prevStrat(i) \mid\mid t_i, \gamma) = comp(t, \gamma)$ and therefore $comp(t, \gamma) \not\models \varphi$ follows; contradicting that $comp(t, \gamma) \models \varphi$ holds.

Hence, for all components c_i obtained with semantic component selection, the synthesis task in line 7 of Algorithm 6.1 succeeds. Therefore, it follows immediately that Algorithm 6.1 returns $(\texttt{true}, s)$, where s is the parallel composition of the n strategies $s_1, \dots, s_n$ synthesized for the components $c_1, \dots, c_n$ of the decomposition $\mathbb{D}$. □

Therefore, incremental synthesis always yields strategies for all components if the decomposition and the synthesis order are computed with the semantic decomposition algorithm and if the specification does not induce semantic dependencies from output variables to input variables of the system. This relies heavily on the fact that, in this setting, all existing semantic dependencies induced by the specification are dependencies to output variables and can thus be resolved by a clever selection of the components and a suitable definition of the ranking function defining the synthesis order.

General LTL specifications, however, can induce semantic dependencies from output variables to input variables of the system. Neither component selection nor the synthesis order can resolve semantic dependencies to input variables, and therefore admissibility is not guaranteed for such LTL specifications. Therefore, the individual synthesis tasks in line 7 of Algorithm 6.1 are not guaranteed to succeed, and hence incremental synthesis might not yield strategies for all components. If the synthesis task for a component c_i, which is the single component of the highest rank, fails, however, it follows that the LTL specification φ is unrealizable for the whole system. Note here that such a component can either be a component defined by the decomposition or the parallel composition of components defined by the decomposition.

Theorem 6.4. *Let φ be an LTL formula over atomic propositions V. Let $\mathbb{D}$ and $rank_{syn}$ be the decomposition and the ranking function computed with semantic decomposition algorithm. Let $c_i \in \mathbb{D}$ be some component such that $rank_{syn}(c_j) < rank_{syn}(c_i)$ holds for all components $c_j \in \mathbb{D}$. If the synthesis task in line 7 of Algorithm 6.1 fails for c_i while the synthesis tasks for all components with lower rank succeeded, then φ is unrealizable for the whole system.*

Proof. Suppose that the synthesis task in line 7 of Algorithm 6.1 fails for component c_i while the synthesis tasks for all components with lower rank succeeded. For the sake of readability, let $\overline{c_i}$ denote the parallel composition of all components of $\mathbb{D}$ except c_i. Let t_i be the parallel composition of the strategies of all components of $\mathbb{D}$ except c_i. Since c_i is the only component of highest rank by assumption, all these strategies have been synthesized previously and thus t_i exists. Furthermore, by construction of Algorithm 6.1 as well as the fact that components of the same rank are only synthesized separately if φ is a safety specification or its liveness part is only affected by one of the components, it follows that t_i is dominant for $\overline{c_i}$.

Suppose that φ is realizable for the full system. Then, φ is admissible for the full system as well [DF14]: since φ is realizable, there exists a strategy s for the entire system $\overline{c_i} \mid\mid c_i$

such that $s \models \varphi$ holds. Hence, it follows immediately from the definition of remorsefree dominance that s is dominant for φ and $\overline{c_i} \mid\mid c_i$ as well. Therefore, we can show similar as in the last part of the proof of Lemma 6.3 that the synthesis task for c_i succeeds. Let $\overline{s_i}$ be the strategy for $\overline{c_i}$ that always behaves as s restricted to the output variables of $\overline{c_i}$, i.e., with $\mathit{comp}(s, \gamma) \cap \overline{V_i} = \mathit{comp}(\overline{s_i}, \gamma \cup \gamma') \cap \overline{V_i}$ for all $\gamma \in (2^I)^\omega$ and all $\gamma' \in (2^{O_i})^\omega$, where $\overline{V_i}$ denotes the variables of $\overline{c_i}$. Similarly, let s_i be the strategy for c_i that always behaves as s restricted to the outputs of c_i, i.e., with $\mathit{comp}(s, \gamma) \cap V_i = \mathit{comp}(s_i, \gamma \cup \gamma') \cap V_i$ for all $\gamma \in (2^I)^\omega$ and all $\gamma' \in (2^{\overline{O_i}})^\omega$, where $\overline{O_i}$ denotes the outputs of $\overline{c_i}$. Since s realizes φ by assumption, we then have $\mathit{comp}(\overline{s_i}, \mathit{comp}(s, \gamma) \cap \overline{I_i}) \models \varphi$ for all $\gamma \in (2^I)^\omega$, where $\overline{I_i}$ denotes the inputs of $\overline{c_i}$. Thus, since t_i is dominant for $\overline{c_i}$ by assumption, $\mathit{comp}(t_i, \mathit{comp}(s, \gamma) \cap \overline{I_i}) \models \varphi$ holds for all $\gamma \in (2^I)^\omega$ as well. But then strategy s_i for c_i is a strategy for c_i such that $t_i \mid\mid s_i \models \varphi$, contradicting the assumption that the synthesis task for c_i fails. □

Thus, when encountering a component for which the synthesis task does not succeed in incremental synthesis, we can immediately deduce non-realizability of the LTL specification φ if there is no component with a higher or equal rank in the synthesis order than the one of the considered component. If synthesis in Algorithm 6.1 fails for other components, i.e., components of non-highest rank or if there are multiple components of highest rank, in contrast, non-realizability of the specification does not follow in general:

Example 6.5. Consider the LTL formula $\varphi = a \vee ((\bigcirc b) \leftrightarrow (\bigcirc\bigcirc i))$, where i is an input variable and both a and b are output variables, i.e., we have $I = \{i\}$ and $O = \{a, b\}$. Clearly, a depends semantically on the future valuation of b. Variable b, in contrast, does not depend semantically on the current or future valuation of a. Hence, the semantic dependency graph $\mathcal{D}^{sem}_{\varphi}$ of φ contains an edge from a to b but no edge from b to a. The semantic decomposition algorithm thus derives two components c_1 and c_2, where c_1 controls a and c_2 controls b, i.e., we have $O_1 = \{a\}$ and $O_2 = \{b\}$, and the ranking function rank_{syn} assigns c_2 a lower value than c_1, i.e., we have $\mathit{rank}_{syn}(c_2) < \mathit{rank}_{syn}(c_1)$. Therefore, Algorithm 6.1 tries to synthesize a strategy for c_2 first. Since c_2 is of lowest rank, incremental synthesis seeks for a strategy s_2 for c_2 that is dominant for φ and c_2. Yet, φ is not admissible for c_2 since a dominant strategy would need to predict the future valuation of i in order to determine the correct valuation for b. Hence, the synthesis task for c_2 fails and therefore Algorithm 6.1 returns (`false`, Null). Thus, incremental synthesis does not yield a solution. However, φ is realizable for the whole system since a strategy that sets output variable a to *true* in the very first time step satisfies φ irrespective of the valuations of b and, in particular, the input variable i. Thus, incremental synthesis in its current form is not complete for specifications that induce semantic dependencies of components of non-highest rank to input variables. △

Based on this observation, we *extend* the incremental synthesis algorithm as follows. Whenever we encounter a component $c_i \in \mathbb{D}$ of non-highest rank for which the synthesis task in line 7 of Algorithm 6.1 fails and which contains an output variable that semantically depends on an input variable of the system, then we *combine* component c_i with a component $c_j \in \mathbb{D}$ that is a direct successor of c_i in the synthesis order, i.e., we have $\mathit{rank}_{syn}(c_i) < \mathit{rank}_{syn}(c_j)$ and there does not exist a component $c_\ell \in \mathbb{D}$ with $\mathit{rank}_{syn}(c_i) < \mathit{rank}_{syn}(c_\ell) < \mathit{rank}_{syn}(c_j)$. Hence, we obtain a component $c_{i,j}$ with outputs $O_{i,j} = O_i \cup O_j$ and replace both components c_i and c_j with $c_{i,j}$. Furthermore, we assign $c_{i,j}$ the same rank as c_j, i.e., we define $rank_{syn}(c_{i,j}) = rank_{syn}(c_i)$. We then consider $c_{i,j}$ exactly as all other components of the system. In particular, we try to synthesize a strategy $s_{i,j}$ for $c_{i,j}$ such that $prevStrat(i) \mid\mid s_{i,j}$ is dominant for φ and $prev(\ell)$, where $prevStrat(i)$ denotes the parallel composition of the previously synthesizes strategies for components with lower ranks, where ℓ is the index of the layer $\mathbb{L}$ in which $c_{i,j}$ lies, and where $prev(\ell)$ denotes the parallel composition of all components with lower rank. If the synthesis task for $c_{i,j}$ succeeds, we proceed with the other components as in the basic incremental synthesis algorithm. Otherwise, we proceed with combining $c_{i,j}$ with a direct successor in the synthesis order until either a component is obtained for which the synthesis task succeeds or only a single component is left. For this extension of the incremental synthesis algorithms, completeness then follows immediately from Lemma 6.3 and Theorem 6.4:

Theorem 6.5. *Let φ be an LTL formula over atomic propositions V. Let $\mathbb{D}$ and $rank_{syn}$ be the decomposition and the ranking function computed with semantic decomposition algorithm. If the extended incremental synthesis algorithm returns* (`false`, Null) for input φ, $\mathbb{D}$, and $rank_{syn}$, then φ is unrealizable.

Thus, analyzing semantic dependencies of the output variables of the system and computing components as well as a ranking function based on these dependencies, ensures both soundness and completeness of (extended) incremental synthesis. Therefore, semantic component selection is a suitable decomposition algorithm for incremental synthesis with dominant strategies. In the next section, we discuss a further optimization of the semantic decomposition algorithm that allows for resolving present dependencies unidirectionally, thus possibly increasing the number of components in the decomposition.

6.3.3. Resolving Present Dependencies

As outlined before, present dependencies may also yield non-admissibility of a component since, in our setting, strategies are represented by Moore transducers, which are not able to react to an input immediately. This modeling choice relies, among others, on the assumption that the system components will be executed perfectly parallel. In many realistic systems, however, perfect parallelism is often not achieved. Instead, the components are executed slightly time-delayed in every time step, enabling components *can* to react to inputs immediately if the inputs are produced by a component that is executed beforehand in this time step.

We call the order in which the components are executed within a single time step the *implementation order* and denote it with $<_{impl}$. Hence, if $c_i <_{impl} c_j$ holds, then c_i is executed before c_j. Intuitively, the implementation order defines the communication interface between the components. It assigns a rank $rank_{impl}(c_i)$ to every component c_i of the system. Let c_i and c_j be components with output variables O_i and O_j, respectively. If $rank_{impl}(c_i) < rank_{impl}(c_j)$ holds, then component c_j can, intuitively, observe the valuations of the variables in O_i one step in advance, i.e., it is able to directly react to them, modeling knowledge about these variables in the whole system. The implementation order is not necessarily total. Slightly overloading notation, we also define the rank $rank_{impl}(u)$ of an output variable $u \in O$ of the system in the implementation order. To ensure consistency, we define the rank of a component to be the maximum rank of all of its output variables, i.e., we have $rank_{impl}(c_i) = \max \{rank_{impl}(u) \mid u \in O_i\}$.

The full implementation order or some parts of it might be fixed by the system design and, in particular, technical necessities prior to synthesis. If the implementation order is not fully defined, we can refine it during component selection, thereby resolving present dependencies unidirectionally, as follows. Let $u \in O$ be an output variable of the system that depends semantically on the current valuation of a variable $v \in V$, i.e., let $(u,v) \in \mathcal{E}_P^{sem}$ hold, where $\mathcal{D}_\varphi^{sem} = (\mathcal{V}^{sem}, \mathcal{E}_P^{sem})$ is the semantic dependency graph induced by the LTL specification φ. If $rank_{impl}(u) < rank_{impl}(v)$ holds, then v is already able to react to u immediately, and therefore we can remove the edge (u,v) from $\mathcal{E}_P^{sem}$. If $rank_{impl}(u) > rank_{impl}(v)$ holds, then v will not be able to react to u immediately, and therefore we keep the edge (u,v) in $\mathcal{E}_P^{sem}$ as the dependency can affect admissibility. If neither $rank_{impl}(u) < rank_{impl}(v)$ nor $rank_{impl}(u) > rank_{impl}(v)$ is already predefined in the implementation order, then we refine it: we add $rank_{impl}(u) < rank_{impl}(v)$ and remove the edge (u,v) from $\mathcal{E}_P^{sem}$. Due to preserving consistency of the implementation order while refining it, at most one of the present dependencies between two variables u and v with $(u,v), (v,u) \in \mathcal{E}_P^{sem}$ can be resolved in this way.

We can incorporate resolving present dependencies into the semantic decomposition algorithm as follows. Similar to the original version, we first compute the semantic dependency graph $\mathcal{D}_\varphi^{sem} = (\mathcal{V}^{sem}, \mathcal{E}_P^{sem})$ of the specification φ and remove nodes that correspond to input variables. Afterward, we remove present dependency edges if there exist analogous future dependency edges. That is, if both $(u,v) \in \mathcal{E}_P^{sem}$ and $(u,v) \in \mathcal{E}_F^{sem}$ hold, then we remove (u,v) from $\mathcal{E}_P^{sem}$ since, intuitively, future edges subsume present edges in the sense that they cannot be resolved. Hence, resolving the present dependency from u to v – if possible – would only unnecessarily restrict the implementation order, thereby possibly preventing that other present dependencies can be resolved, while not increasing the number of components in the decomposition. Once we have removed subsumed present edges, we resolve present dependencies unidirectionally, as described above, by refining the implementation order. Based on the resulting dependency graph, we then compute the components of the system as in the original decomposition algorithm, i.e., by computing the strongly connected components and defining the synthesis order according to the dependencies between the components.

Example 6.6. Consider the autonomous car from Section 6.1 and its dependency graph depicted in Figure 6.3. With the new decomposition algorithm, we eliminate the present dependency edges between *acc*, *dec*, and *keep* as they are subsumed by the respective future edges. Furthermore, assuming that there are no contradictory restrictions by the system design, we can resolve the present dependency between g_1 and g_2 in one direction by refining the implementation order. For instance, we can define $rank_{impl}(g_2) < rank_{impl}(g_1)$ and eliminate the present dependency edge from g_1 to g_2. The resulting dependency graph induces three components c_1, c_2, and c_3 with $O_1 = \{acc, dec, keep\}$, $O_2 = \{g_1\}$ and $O_3 = \{g_3\}$. Furthermore, we obtain $rank_{impl}(c_3) < rank_{impl}(c_2)$ and thus $c_3 <_{impl} c_2$. Lastly, we obtain the following synthesis order $c_2 <_{syn} c_3 <_{syn} c_1$, which still matches our intuition that a strategy for the gearing unit, and thus for the outputs g_1 and g_2 representing the gearing unit, must be synthesized first. △

As we do not alter the computation of the synthesis order, it is still guaranteed that the synthesis order induced by this slight variation of the semantic decomposition algorithm is compositionality-preserving (see Lemma 6.2). Hence, soundness of incremental synthesis follows immediately (see Corollary 6.1). Next, we study completeness.

By Theorem 6.3, we know that a specification φ is admissible for a component c_i if none of the output variables of c_i depends semantically on any variable outside of the control of c_i. We now extend it to the case where output variables of c_i may depend on the current valuation of variables outside of the control of c_i as long as these dependencies can be resolved with the implementation order:

Theorem 6.6. *Let φ be an LTL formula over atomic propositions V. Let c_i be some component of the system with output variables $O_i \subseteq O$. If, for all $u \in O_i$, variable u neither depends semantically on the current valuation of a variable $v \in V \setminus O_i$ if $rank_{impl}(c_i) \leq rank_{impl}(c_j)$ holds, where c_j is the component with $v \in O_j$, nor on the future valuation of a variable $v \in V \setminus O_i$, then φ is admissible for the component c_i.*

Proof. Let $\gamma \in (2^{I_i})^\omega$ be some infinite input sequence for c_i. If $\gamma \cup v \not\models \varphi$ holds for all $v \in (2^{O_i})^\omega$, then, intuitively, no strategy for c_i at all can satisfy the specification φ on input γ. Thus, a dominant strategy for c_i may behave arbitrarily on input γ. Otherwise, there exists some $v^\gamma \in (2^{O_i})^\omega$ such that $\gamma \cup v^\gamma \models \varphi$ holds. As in the proof of Theorem 6.3, we claim that there exists a strategy s_i for c_i such that we have $comp(s_i, \gamma) \cap O_i = v^\gamma$ for all input sequences $\gamma \in (2^{I_i})^\omega$ for which there exists some $v^\gamma \in (2^{O_i})^\omega$ such that $\gamma \cup v^\gamma \models \varphi$ holds. If there are multiple sequences $v^\gamma \in (2^{O_i})^\omega$ such that $\gamma \cup v^\gamma \models \varphi$ holds, then the strategy s_i chooses one of these sequences.

Suppose that such a strategy does not exist. As we use the very same strategy as in the proof of Theorem 6.3, we can show analogously that there exists a variable $u \in O_i$ that either (i) depends semantically on the current valuation of some variable $v \in I_i$, or (ii) depends semantically on the future valuation of some variable $v \in I_i$. In the latter case, we immediately obtain a contradiction to the assumption that no output of c_i depends syntactically on the future valuations of any variable outside of the control of c_i since the set of input and output variable of c_i are disjoint and thus $v \in V \setminus O_i$ holds. In the former case, if $rank_{impl}(c_i) \leq rank_{impl}(c_j)$ holds, then we obtain a contradiction to the assumption that no output variable of c_i depends on the output valuation of a component c_j with higher rank in the implementation order. Otherwise, i.e., if $rank_{impl}(c_i) > rank_{impl}(c_j)$ holds, then c_i is able to react to the outputs of c_j immediately and thus, in particular, it is able to react to the valuation of v immediately. Therefore, a strategy for c_i can choose the correct valuation of u for the considered input. If other, non-resolvable dependencies exist, we obtain a contradiction, as shown above. Otherwise, the desired strategy exists, contradicting the assumption that it does not exist.

Hence, all in all, the desired strategy s_i for c_i exists; therefore, we can show as in the proof of Theorem 6.3 that s_i is dominant for φ, concluding the proof. □

Consequently, admissibility of the given specification for the components is guaranteed when computing the decomposition as well as the implementation order and the synthesis order with the slightly modified semantic decomposition algorithm, which allows for unidirectionally resolving present dependencies between variables, introduced above. Therefore, completeness of incremental synthesis for this semantic decomposition algorithm follows analogously to the proof of Lemma 6.3 when considering specifications that do not induce any semantic dependencies to input variables of the entire system.

Similar to the original version of the semantic decomposition algorithm, we can extend the completeness result to general specifications (see Theorem 6.5) also for the modified semantic decomposition algorithm by merging components with their direct successors in the synthesis order whenever we encounter a component for which the synthesis task fails and that contains a semantic dependency to an input variable. Therefore, we can add the possibility of resolving present dependencies by refining the implementation order to the semantic component selection algorithm while maintaining soundness and completeness of incremental synthesis.

Semantic component selection, in both its variants, is thus a suitable decomposition algorithm for incremental synthesis with dominant strategies. However, computing semantic dependencies is hard: the semantic definition of dependencies, i.e., Definition 6.3, is a *hyperproperty* [CS10], i.e., a property that relates multiple execution traces. In particular hyperproperties with quantifier alternation, such as the definition of semantic variable dependencies, have been proven to constitute a challenging class of properties. Computing semantic dependencies and, thus, decomposing a system with the semantic decomposition algorithm is often not practical. In the following chapter, we thus introduce a syntactical criterion for determining present and future dependencies between variables more efficiently.

6.4. Syntactic Component Selection

In this section, we present an algorithm for selecting components as well as ordering them which is based on *syntactic* dependencies between the output variables of the overall system. Similar to the semantic decomposition algorithm, the syntactic one also directly induces a ranking function $rank_{syn}$ defining a compositionality-preserving synthesis order that ensures completeness of incremental synthesis.

We first introduce a syntactic criterion for dependencies between variables of the system. In the long term, syntactic dependencies are an overapproximation of semantic dependencies in the sense that every semantic dependency also induces a syntactic dependency. Similar to semantic component selection, we then introduce a syntactic decomposition algorithm that identifies equivalence classes of output variables of the system based on syntactic dependencies, which constitute the components of the system.

6.4.1. Syntactic Dependencies

Syntactic dependencies between output variables are determined by analyzing the *structure* of the LTL specification φ. In contrast to semantic dependencies, which are based on minimal satisfying change sets, it does not analyze which variables, or, more precisely, their valuations, need to be changed in order to achieve satisfaction of φ when fixing the valuation of a certain variable at some point in time, but deduces this information from the occurrences of the variables at different positions in the LTL formula.

Intuitively, we derive syntactic dependencies by analyzing the syntax tree of the LTL formula bottom-up. Thereby, we collect information about the number of $\bigcirc$-operators under which variables occur and whether they occur under any unbounded temporal operator, i.e., under $\mathcal{U}$, $\mathcal{W}$, $\square$, or $\diamondsuit$. Furthermore, we capture which binary operators connect the variables and group the variables accordingly.

Definition 6.5 (Syntactic Dependencies)**.**
Let φ be an LTL formula in negation normal form over atomic propositions V. Let $\mathbb{T}(\varphi)$ be the syntax tree of φ. Let q be a node of $\mathbb{T}(\varphi)$. If q is a unary operator, let $c(q)$ be its single child. If q is a binary operator, let $lc(q)$ be its left child and let $rc(q)$ be its right child. We assign a *dependency set* $\mathcal{D}_q \subseteq 2^{V\times\mathbb{N}_0\times\mathbb{B}}$ to q as follows:

- if q is a leaf, then $\mathcal{D}_q = \{\{(q, 0, \mathit{false})\}\}$,
- if $q = \neg$, then $\mathcal{D}_q = \mathcal{D}_{c(q)}$
- if $q = \wedge$, then $\mathcal{D}_q = \mathcal{D}_{lc(q)} \cup \mathcal{D}_{rc(q)}$,
- if $q = \vee$, then $\mathcal{D}_q = \bigcup_{M\in\mathcal{D}_{lc(q)}} \bigcup_{M'\in\mathcal{D}_{rc(q)}} \{M \cup M'\}$,
- if $q = \bigcirc$, then $\mathcal{D}_q = \bigcup_{M\in\mathcal{D}_{c(q)}} \{\{(u, x+1, y) \mid (u,x,y) \in M\}\}$,
- if $q = \square$, then $\mathcal{D}_q = \mathcal{D}_{c(q)} \cup \bigcup_{M\in\mathcal{D}_{c(q)}} \{\{(u, x, \mathit{true})\} \mid (u,x,y) \in M\}$,
- if $q = \diamondsuit$, then $\mathcal{D}_q = \left\{\bigcup_{M\in\mathcal{D}_{c(q)}} \{(u, x, \mathit{true}), (u, x, \mathit{false}) \mid (u,x,y) \in M\}\right\}$,
- if $q = \mathcal{U}$ or $q = \mathcal{W}$, then

$$\begin{aligned}\mathcal{D}_q = &\bigcup_{M\in\mathcal{D}_{lc(q)}} \bigcup_{M'\in\mathcal{D}_{rc(q)}} \{M \cup M'\}\\ \cup &\bigcup_{M\in\mathcal{D}_{lc(q)}} \bigcup_{M'\in\mathcal{D}_{rc(q)}} \bigcup_{(u,x,y)\in M} \{\{(u, x, \mathit{true})\} \cup M'\}\\ \cup &\left\{ \bigcup_{M'\in\mathcal{D}_{rc(q)}} \{(u, x, \mathit{true}), (u, x, \mathit{false}) \mid (u,x,y) \in M'\} \right\}.\end{aligned}$$

Let q be the root node of $\mathbb{T}(\varphi)$ and let $(u,x,y), (v,x',y') \in M$ for some $M \in \mathcal{D}_q$, $u, v \in V$, $x, x' \in \mathbb{N}_0$, and $y, y' \in \mathbb{B}$ with $(u,x,y) \neq (v,x',y')$. Then, we say that *u depends syntactically on the current valuation of v*, if, and only if, $u \neq v$ and either $y = y' = \mathit{false}$ and $x = x'$, or $y = \mathit{true}$ and $y' = \mathit{false}$ and $x \leq x'$, or $y = \mathit{false}$ and $y' = \mathit{true}$ and $x \geq x'$, or $y = y' = \mathit{true}$. Furthermore, we say that *u depends syntactically on the future valuation of v*, if, and only if, either $y' = \mathit{true}$, or $y' = \mathit{false}$ and $x < x'$. The *offset* of the future dependency is ∞ in the former case and $x' - x$ in the latter case.

Dependency sets are sets of sets of triples (u,x,y), where $u \in V$ is a variable, $x \in \mathbb{N}_0$ is a natural number (including zero), and $y \in \mathbb{B}$ is a Boolean flag. Intuitively, the second component x of a triple (u,x,y) denotes the number of $\bigcirc$-operators under which variable u, which is defined by the first component, occurs. The third component y is a Boolean flag that captures whether variable u occurs under an *unbounded temporal operator.* A bounded temporal operator defines a particular point in time at which the property needs to be satisfied. All other temporal operators are unbounded. That is, $\bigcirc$ is the only temporal operator that is bounded. All other temporal operators, i.e., $\square$, $\diamondsuit$, $\mathcal{U}$, and $\mathcal{W}$, are unbounded.

Every set $M \in \mathcal{D}_q$ that is contained in the dependency set of node q of the syntax tree $\mathbb{T}(\varphi)$ of φ defines dependencies between the variables in M. However, different sets $M, M' \in \mathcal{D}_q$ are independent. That is, while there are dependencies between the variables in M and dependencies between the variables in M', there are no dependencies from variables in M to variables in M' or vice versa – unless these dependencies are defined within M or M' itself or within another set $M'' \in \mathcal{D}_q$. The dependency sets are built recursively from the dependency sets of their children. The dependency set of the root node q of the syntax tree $\mathbb{T}(\varphi)$ of the given LTL formula φ then defines the syntactic present and future dependencies.

First, we consider syntactic present dependencies between variables $u \in V$ and $v \in V$. They are induced by two triples $(u, x, y), (v, x', y') \in M$ for some $M \in \mathcal{D}_q$ with $u \neq v$. Clearly, u depends on the current valuation of v if both u and v do not occur under an unbounded temporal operator, i.e., if $y = y' = \mathit{false}$ holds, and if they occur under the same number of $\bigcirc$-operators, i.e., if $x = x'$ holds, since then φ poses restrictions on u and v at the very same concrete time step. If u occurs under an unbounded temporal operator while v does not, i.e., if $y = \mathit{true}$ and $y' = \mathit{false}$ holds, and if u occurs under at most as many $\bigcirc$-operators as v does, i.e., if $x \leq x'$ holds, then u depends syntactically on the current valuation of v as well: φ poses restrictions on v at some concrete time step $k \geq 0$, while it poses unbounded restrictions on u at a point in time $j \geq 0$ with $j \leq k$. The unboundedness of the restrictions on u yields that φ might also pose restrictions on u at point in time k, thus possibly influencing the valuation of v at point in time k. If v occurs under an unbounded temporal operator while u does not, i.e., if $y = \mathit{false}$ and $y' = \mathit{true}$ hold, and if u occurs under at least as many $\bigcirc$-operators as v does, i.e., if $x \geq x'$ holds, then u depends syntactically on the current valuation of v as well: φ poses restrictions on u at some concrete time step $k \geq 0$, while it poses unbounded restrictions on v at a point in time $j \geq 0$ with $j \leq k$. The unboundedness of the restrictions on v then yields, similar to the case above, that φ might also pose restrictions on v at point in time k. Thus, u can possibly influence the valuation of v at point in time k. Lastly, u depends syntactically on the current valuation of v if both u and v occur under an unbounded temporal operator. Due to the unboundedness, φ can thus pose restrictions on u and v at an arbitrary late point in time, and therefore, in particular, the valuation of u can influence the valuation of v in the same time step.

Next, we consider syntactic future dependencies induced by triples $(u, x, y), (v, x', y') \in M$ for some $M \in \mathcal{D}_q$. Note that for future dependencies u and v do not necessarily differ. If v occurs under an unbounded temporal operator, i.e., if $y' = \mathit{true}$ holds, then φ can pose restrictions on v at arbitrary late points in time. Hence, no matter under which operators u occurs, the valuation of u at some point in time can influence the valuation of v at a later point in time. Therefore, u depends syntactically on the future valuation of v. Since the valuation of v can be influenced at any infinitely late point in time, the offset of this dependency is ∞. If, in contrast, v does not occur under an unbounded temporal operator, i.e., if $y' = \mathit{false}$ holds, and if u occurs under less $\bigcirc$-operators than v does, i.e., if $x < x'$ holds, then φ only poses restrictions on v in a concrete time step. Irrespective of whether or not u occurs under an unbounded temporal operator, φ can pose restrictions on φ at an earlier point in time than on v and therefore u depends syntactically on the future valuation of v. Since the valuation of v can only be influenced at a concrete point in time, the offset of this dependency is defined by $x' - x$.

In the following, we explain all cases of the above definition of the dependency set of a node q of $\mathbb{T}(\varphi)$ in detail. If q is a leaf node, then, by definition of the syntax of LTL, q is a atomic proposition, and thus we have $q \in V$. Hence, in particular, when considering node q to be the root node, then variable q neither occurs under any $\bigcirc$-operator nor under any temporal operator. Hence, in this case, q constitutes the single triple $(q, 0, \mathit{false})$. A disjunction $\psi \vee \psi'$ intuitively introduces dependencies between the disjuncts ψ and ψ' since the satisfaction of ψ affects the need of satisfaction of ψ' and vice versa. If one of the disjuncts is not satisfied, then the other one needs to be satisfied in order to satisfy the disjunction. Vice versa, if one of the disjuncts is satisfied, then the other one does not need to be satisfied. Hence, since the valuations of the variables occurring in ψ influence whether or not ψ is satisfied, they indirectly also influence whether or not ψ' needs to be satisfied in order to satisfy $\psi \vee \psi'$ and therefore they may influence the valuations of the variables occurring in ψ'. Therefore, we combine the dependency sets of the children of q into a single dependency set if $q = \vee$ holds. In a conjunction, in contrast, a conjunct needs to be satisfied irrespective of the other conjuncts. Therefore a conjunction does not introduce dependencies between the conjuncts. In order to satisfy a conjunction $\psi \wedge \psi'$, both conjuncts need to be satisfied. Thus, the valuation of the variables occurring in ψ does not influence the valuation of the variables occurring in ψ' since ψ' needs to be satisfied no matter whether or not ψ is satisfied. Thus, if $q = \wedge$ holds, we keep the dependency sets of the conjuncts separate.

If q is the $\neg$-operator, then we utilize the assumption that φ is in negation normal form. Due to this requirement, a negation can only occur in front of atomic propositions. Thus, a negation does not influence any dependencies; therefore, a negation does not alter the dependency set either. Similar to conjunction, the $\square$-operator does not introduce dependencies between the variables occurring in the property. The formula $\square\,\psi$ is fulfilled if ψ is satisfied at every point in time. Hence, irrespective of whether or not ψ is satisfied at some other point in time, it needs to be satisfied at a considered point in time. Thus, the valuation of the variables occurring in ψ does not influence the valuation of the variables occurring in ψ at some other point in time. Hence, we do not merge the sets $M \in \mathcal{D}_c(q)$ of the dependency set of ψ if $q = \square$ but keep them. Additionally, we include individual sets for all triples occurring in $\mathcal{D}_c(q)$, where we set the third component to *true* since $\square$ is an unbounded temporal operator. The $\diamondsuit$-operator, in contrast, is similar to a disjunction and thus introduces dependencies between the variables occurring in the property. The formula $\diamondsuit\,\psi$ is satisfied if ψ is fulfilled at some point in time. Hence, if ψ is not satisfied at some point in time, then it needs to be satisfied at some other point in time. If ψ is satisfied at some point in time, in contrast, then it does not need to be satisfied at another point in time. Thus, similar to the dependencies from one disjunct to another in a disjunction, the $\diamondsuit$-operator induces dependencies from the variables in ψ to themselves. Therefore, we combine all dependency sets of q if $q = \diamondsuit$ holds. Note here that it is necessary to include both triples with $y = \mathit{true}$ and triples with $y = \mathit{false}$ to also obtain future dependencies from a variable to itself if ψ only contains a single variable.

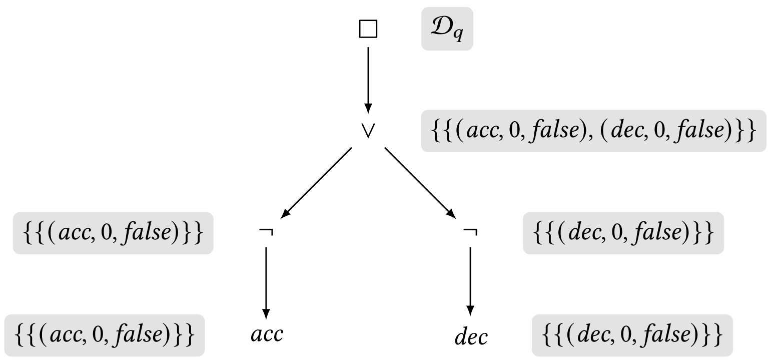

$\square$
$\mathcal{D}_q$
$\vee$
$\{\{(acc, 0, false), (dec, 0, false)\}\}$
$\{\{(acc, 0, false)\}\}$
$\neg$
$\neg$
$\{\{(dec, 0, false)\}\}$
$\{\{(acc, 0, false)\}\}$
acc
dec
$\{\{(dec, 0, false)\}\}$

Figure 6.4.: Syntax tree of φ_1 from Example 6.7 with dependency set annotations.

Both binary temporal operators $\mathcal{U}$ and $\mathcal{W}$, introduce dependencies between the left and the right component. If, for a property $\psi \,\mathcal{U}\, \psi'$ or $\psi \,\mathcal{W}\, \psi'$, the component ψ is not satisfied at some point in time, then component ψ' needs to be satisfied either in the same point in time or at an earlier point in time. If ψ' is not satisfied at some point in time, then it either needs to be satisfied at an earlier point in time or ψ needs to be satisfied at the same point in time. Thus, similar to a disjunction, the variables in ψ influence the valuation of the variables in ψ' and vice versa, and therefore we combine the dependency sets of the two components. Furthermore, the operators introduce dependencies from the variables in the right component ψ' to themselves. Suppose that the left component ψ is satisfied up to a point in time $k \geq 0$. If, at some point in time $k' \geq 0$ with $k' < k$, the right component ψ' is not satisfied, then it needs to be satisfied at a later point in time $k'' \geq 0$ with $k' < k'' \leq k + 1$ to still ensure that $\psi \,\mathcal{U}\, \psi'$ or $\psi \,\mathcal{W}\, \psi'$ is satisfied. As for the $\diamondsuit$-operator, we thus combine all dependency sets of the right child of q and consider both triples with $y = \mathit{true}$ and with $y = \mathit{false}$ to obtain future dependencies from a variable to itself if ψ' only contains a single variable. Lastly, there are future dependencies from ψ' to ψ since whether or not ψ is satisfied in the future affects the need of satisfaction of ψ' in the current step. If ψ will not be satisfied at some point in time $k \geq 0$ in the future, then it will need to be satisfied before or at point in time k. Hence, the valuations of the variables in ψ' clearly influence the valuations of the variables in ψ. Therefore, we combine the dependency sets of the nodes representing ψ and ψ', yet, in contrast to the very first part of the definition of the dependency sets for $\mathcal{U}$ and $\mathcal{W}$, we set the Boolean flag, which indicates whether or not the variable occurs under an unbounded temporal operator, to *true* for the variables in ψ. This captures the future dependency from ψ' to ψ. Note that we do not need to introduce further future dependencies from ψ to ψ' since whether or not ψ is satisfied at some point in time only affects the need for satisfaction of ψ' at the very same point in time, which is already captured by the present dependencies between the operands introduced above.

Example 6.7. Reconsider the autonomous car from the running example from Section 6.1 and its specification $\varphi_{car} = \varphi_{acc} \wedge \varphi_{gear}$. Let $\varphi_1 = \square\neg(acc \wedge dec)$ be the conjunct of φ_{acc} that establishes mutual exclusion between *acc* and *dec*. When translating it into negation normal form, we obtain $\square(\neg acc \vee \neg dec)$. In the following, we assume that φ_1 denotes the translation

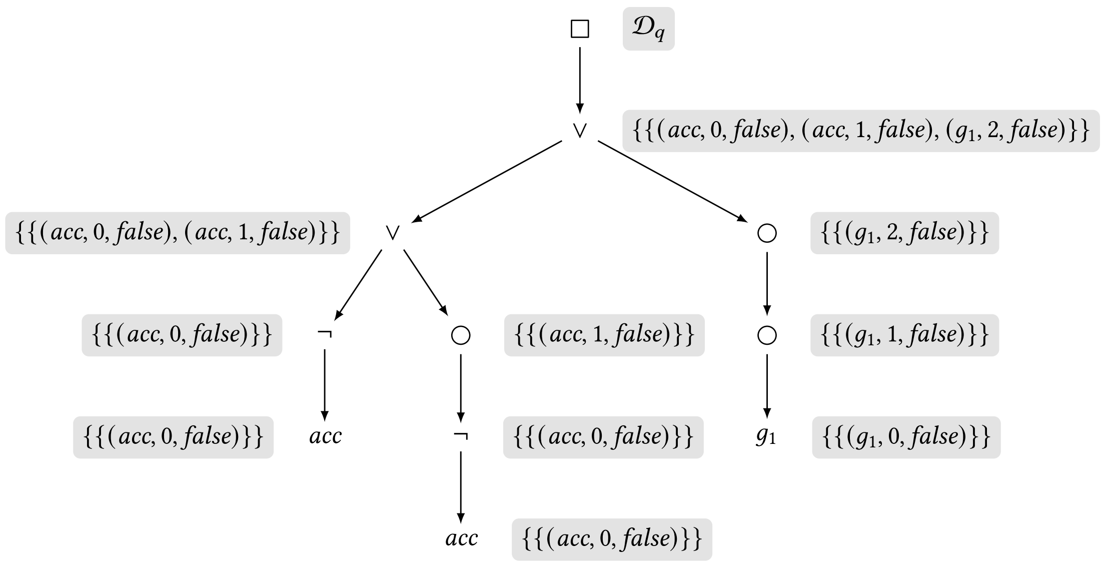

$\square$
$\mathcal{D}_q$
$\vee$
$\{\{(acc, 0, false), (acc, 1, false), (g_1, 2, false)\}\}$
$\{\{(acc, 0, false), (acc, 1, false)\}\}$
$\vee$
$\bigcirc$
$\{\{(g_1, 2, false)\}\}$
$\{\{(acc, 0, false)\}\}$
$\neg$
$\bigcirc$
$\{\{(acc, 1, false)\}\}$
$\bigcirc$
$\{\{(g_1, 1, false)\}\}$
$\{\{(acc, 0, false)\}\}$
acc
$\neg$
$\{\{(acc, 0, false)\}\}$
g_1
$\{\{(g_1, 0, false)\}\}$
acc
$\{\{(acc, 0, false)\}\}$

Figure 6.5.: Syntax tree of φ_2 from Example 6.7 with dependency set annotations.

into negation normal form, i.e., that $\varphi_1 = \square(\neg acc \vee \neg dec)$ holds. The syntax tree $\mathbb{T}(\varphi_1)$ of φ_1 with its dependency set annotations is depicted in Figure 6.4. Let q be the root node of the syntax tree $\mathbb{T}(\varphi_1)$ of φ_1. We then obtain the dependency set

$$\mathcal{D}_q = \{\{(acc, 0, false), (dec, 0, false)\}, \{(acc, 0, true)\}, \{(dec, 0, true)\}\}.$$

Hence, φ_1 induces a syntactic present dependency from acc to dec and vice versa.

Next, let $\varphi_2 = \square((acc \wedge \bigcirc acc) \rightarrow \bigcirc\bigcirc g_1)$ be the conjunct of φ_{gear} that establishes that the lower gear should be used while accelerating. When translating it into negation normal form, we obtain $\square((\neg acc \vee \bigcirc\neg acc) \vee \bigcirc\bigcirc g_1)$. In the following, we assume that φ_2 denotes the translation into negation normal form, i.e., that $\varphi_2 = \square((\neg acc \vee \bigcirc\neg acc) \vee \bigcirc\bigcirc g_1)$ holds. The syntax tree $\mathbb{T}(\varphi_1)$ of φ_1 with its dependency set annotations is depicted in Figure 6.5. Let q be the root node of the syntax tree $\mathbb{T}(\varphi_2)$ of φ_2. We then obtain the dependency set

$$\begin{aligned}\mathcal{D}_q = \{&\{(acc, 0, false), (acc, 1, false), (g_1, 2, false)\},\\ &\{(acc, 0, true)\}, \{(acc, 1, true)\}, \{(g_1, 2, true)\}\}.\end{aligned}$$

Hence, φ_2 induces a syntactic future dependency from acc to itself with offset 1 and two future dependencies from acc to g_1 with offsets 1 and 2, respectively. $\triangle$

As long as semantic dependencies do not range over several conjuncts of the LTL formula φ, every semantic dependency is captured by a syntactic dependency as well. If there is a semantic dependency from $u \in V$ to $v \in V$ and if φ does not contain conjunctions, then u and v occur in the same set $M \in D_q$, where q is the root node of $\mathbb{T}(\varphi)$, by construction. With structural induction on φ, it thus follows that every semantic dependency has a syntactic counterpart:

Lemma 6.4. *Let φ be a conjunction-free LTL formula in negation normal form over atomic propositions V. Let $u, v \in V$ be variables. If u depends semantically on the current valuation of v, then u depends syntactically on the current valuation of v as well. If u depends semantically on the future valuation of v, then u depends syntactically on the future valuation of v as well.*

Proof. Let $\mathbb{T}(\varphi)$ be the syntax tree of φ and let q denote its root node. Let u depend semantically on v, either on the current valuation or on the future valuation. We first show that there is a set $M \in \mathcal{D}_q$ in q's dependency set such that $(u, x, y), (v, x', y') \in M$ holds for some $x, x' \in \mathbb{N}_0$ and some $y, y' \in \mathbb{B}$. By the syntax of LTL, u and v need to be connected, directly or indirectly, in φ with a binary operator, i.e., with $\wedge$, $\vee$, $\mathcal{U}$, or $\mathcal{W}$. Since φ does not contain any conjunction by assumption, the only possible binary operators are $\vee$, $\mathcal{U}$, and $\mathcal{W}$. All three operators combine the dependency sets of their operands. Thus, if u and v occur in different operands, they are contained in the same set M after applying the binary operator Moreover, it is not possible to split a set M of a dependency set again after establishing it. Thus, there is a set $M \in \mathcal{D}_q$ such that $(u, x, y), (v, x', y') \in M$ holds for some $x, x' \in \mathbb{N}_0$ and some $y, y' \in \mathbb{B}$.

In the following, $k \geq 0$ denotes the point in time at which the valuation of u was changed in order to obtain a violation of φ, i.e., k denotes the point in time at which σ and σ' differ. Furthermore, $k' \geq 0$ denotes a point in time at which v needs to be changed to obtain satisfaction of φ again, i.e., k' denotes a point in time such that changing the valuation of v at point in time k' results in v being part of a minimal satisfying change set. Note that $k \leq k'$ holds by construction. Based on the above observation, we now show by structural induction on φ that u depends syntactically on v as well:

1. u and v both do not occur under any unbounded temporal operator. Then, a change in the valuation of u at a single point in time k may only cause a violation of φ if u occurs under k $\bigcirc$-operators. Analogously, a change in the valuation of v at a point in time k' with $k' \geq k$ may only cause the satisfaction of φ again if v occurs under k' $\bigcirc$-operators. Hence, there exists a set $M \in \mathcal{D}_q$ with $(u, x, \mathit{false}), (v, x', \mathit{false}) \in M$ and $k = x \leq x' = k'$. If u depends semantically on the current valuation of v, then $k = k'$ needs to hold and therefore we then have $x = x'$ as well. Thus, there also exists a syntactic present dependency from u to v. If u depends semantically on the future valuation of v, then $k < k'$ needs to holds and hence we then have $x < x'$ as well. Thus, there exists a syntactic future dependency from u to v as well.

2. u occurs under an unbounded temporal operator while v does not. Then, there is a set $M \in \mathcal{D}_\varphi$ with $(u, x, \mathit{true}), (v, x', \mathit{false}) \in M$ for some $x, x' \in \mathbb{N}_0$. Since v does not occur under an unbounded temporal operator, it follows similar to the previous case that changing the valuation of v at point in time k' can only cause the satisfaction of φ again if v occurs under k' $\bigcirc$-operators. Variable u, in contrast, needs to occur under at most k $\bigcirc$-operators as otherwise changing the valuation of u at point in time k cannot result in the violation of φ. Hence, we have $x \leq k \leq x' = k'$. If u depends semantically on the current valuation of v, then $k = k'$ holds and thus we have $x \leq x'$. Therefore, there then exists a syntactic present dependency from u to v as well. If u depends semantically on the future valuation of v, then $k < k'$ holds and hence we obtain $x < x'$. Thus, there exists a syntactic future dependency from u to v as well.

3. v occurs under an unbounded temporal operator while u does not. Then, there is a set $M \in \mathcal{D}_\varphi$ with $(u, x, \mathit{false}), (v, x', \mathit{true}) \in M$ for some $x, x' \in \mathbb{N}_0$. Analogous to the previous case, we obtain $k' \leq x' \leq x = k$. Hence, there is a syntactic present dependency from u to v. If u also depends semantically on the current valuation of v, then $k = k'$ and thus $x \leq x'$ holds. Thus, there exists a syntactic present dependency from u to v as well.

4. u and v occur under different unbounded temporal operators, or u and v occur under the same $\diamondsuit$-operator, or u and v both occur on the right side of the same $\mathcal{U}$- or $\mathcal{W}$-operator. Then, there is a set $M \in \mathcal{D}_\varphi$ with $(u, x, \mathit{true}), (v, x', \mathit{true}) \in M$ for some $x, x' \in \mathbb{N}_0$. Hence, u depends syntactically both on the current and the future valuation of v.

5. u and v occur on different sides of the same $\mathcal{U}$- or $\mathcal{W}$-operator. Let ψ be the left operand and let ψ' be the right operand of the temporal operator, where either u occurs in ψ and v occurs in ψ' or vice versa. In the former case, there exists a set $M \in \mathcal{D}_\varphi$ with $(u, x, \mathit{true}), (v, x', y') \in M$ for some $x, x' \in \mathbb{N}_0$ and some $y' \in \mathbb{B}$. If there is some triple $(v, x', \mathit{true}) \in M$, then there exists a syntactic present dependency as well as a syntactic future dependency from u to v. If $y' = \mathit{false}$ holds for all $(v, x', y') \in M$, then, by construction of the dependency sets, ψ' does not contain any unbounded temporal operator. Hence, the existence of a syntactic present dependency or a syntactic future dependency from u to v, if there exists a matching semantic one, follows as in the second case. If v occurs in ψ and u occurs in ψ', in contrast, there is a set $M \in \mathcal{D}_\varphi$ with $(u, x, y), (v, x', \mathit{true}) \in M$ for some $x, x' \in \mathbb{N}_0$ and some $y \in \mathbb{B}$. Hence, there exists a syntactic future dependency from u to v. Furthermore, if there is some triple $(u, x, \mathit{true}) \in M$, then there exists a syntactic present dependency from u to v as well. Otherwise, i.e., if $y = \mathit{false}$ holds for all $(u, x, y) \in M$, then by construction of the dependency sets, ψ' does not contain any unbounded temporal operators and therefore the existence of a syntactic present dependency, if there exists a matching semantic one, follows as in the third case.

6. u and v occur under the same $\square$-operator. Let ψ be the operand of the respective operator. If u or v occurs in any other subformula of φ, then the existence of a syntactic present dependency and a syntactic future dependency from u to v follows from the fourth case. Otherwise, changing the valuation of u at point in time k may only cause a violation of φ if it cause a violation of ψ at a point in time k'' with $k'' \leq k$. Similarly, changing the valuation of v at point in time k' may only yield satisfaction of φ if it causes satisfaction of ψ at point in time k. Hence, there is only a semantic present or future dependency from u to v in φ if there is one in ψ. Thus, it follows from the induction hypothesis that there are respective syntactic present or future dependencies from u to v.

7. u and v both occur on the left side of the same $\mathcal{U}$- or $\mathcal{W}$-operator. Let ψ be the left operand of the respective temporal operator. As in the previous case, the existence of both a syntactic present dependency and a syntactic future dependency from u to v follows immediately from the fourth case if u or v occurs in a different subformula of φ. If u or v occurs also in the right operand of the $\mathcal{U}$- or $\mathcal{W}$- operator, then existence of both a syntactic present dependency and a syntactic future dependency from u to v, if there exists a matching semantic one, follows from the fifth case. Otherwise, both u and v only

occur on the left side of the binary temporal operator. But then changing the valuation of u at point in time k may only cause a violation of φ if it causes a violation of ψ at a point in time k' with $k'' \leq k$. Analogously, changing the valuation of v at point in time k' may only yield satisfaction of φ again if it causes satisfaction of ψ at point in time k. Hence, there is only a semantic present or future dependency from u to v in φ if there is one in ψ. Thus, it follows from the induction hypothesis that there are respective syntactic present or future dependencies from u to v.

Thus, in all cases, every semantic present dependency is captured by a syntactic present dependency as well. Furthermore, every semantic future dependency is captured by a syntactic future dependency; proving the claim. □

Hence, syntactic dependencies overapproximate semantic dependencies as long as the specification is conjunction-free. Consequently, it follows immediately from Theorem 6.3 and Lemma 6.4 that a conjunction-free LTL specification is admissible for a component c_i if no output variable of c_i depends syntactically on a variable outside of the control of component c_i. Similar to semantic dependencies, this result enables a decomposition algorithm utilizing syntactic dependencies for conjunction-free specifications. In general, however, the definition of syntactic dependencies as stated in Definition 6.5 does not necessarily capture all semantic dependencies. In particular, semantic dependencies that range over several conjuncts cannot be detected. We say that a semantic dependency ranges over multiple conjuncts if the dependency is not induced by the individual conjuncts but only when considering their conjunction.

Example 6.8. Consider a system with three output variables $u, v, w \in O$ and the LTL specification $\varphi = (u \lor v) \land (\neg v \lor w)$. Then, φ induces the dependency set

$$\mathcal{D}_\varphi = \{\{(u, 0, \mathit{false}), (v, 0, \mathit{false})\}, \{(v, 0, \mathit{false}), (w, 0, \mathit{false})\}\}.$$

Hence, there are syntactic present dependencies between u and v and between v and w. Yet, there are further semantic dependencies. The sequence $\sigma = \{u\}\emptyset^\omega$ satisfies φ. Changing the valuation of u at point in time 0 yields the sequence $\sigma' = \emptyset^\omega$ which violates φ. The sequence $\sigma'' = \{v, w\}\emptyset^\omega$ resulting from changing the valuation of both v and w at point in time 0 satisfies φ. Only changing the valuation of one of the variables is not sufficient and thus (P, F) with $P = \{v, w\}$ and $F = \emptyset$ is a minimal satisfying change set for φ, σ', u, and 0; inducing a semantic present dependency from u to w, which is not captured by the syntactic dependencies. △

In the next section, we thus introduce an algorithm for identifying suitable components for incremental synthesis as well as a corresponding ranking function based on syntactic dependencies that addresses this issue.

6.4.2. Syntactic Decomposition Algorithm

Similar to the algorithm for semantic component selection presented in Section 6.3.2, the syntactic decomposition algorithm is based on determining syntactic dependencies between output variables of the system and by analyzing them to compute suitable components. It tries to maximize the number of components while maintaining admissibility. To determine which output variables of the system need to be contained in the same component in order to ensure admissibility, we build, similar to the semantic decomposition algorithm, the *syntactic dependency graph* of the variables of the system based on the syntactic dependencies:

Definition 6.6 (Syntactic Dependency Graph)**.**
Let φ be an LTL formula over atomic propositions V. The *syntactic dependency graph* $\mathcal{D}_\varphi^{syn}$ of φ is defined by $\mathcal{D}_\varphi^{syn} = (\mathcal{V}^{syn}, \mathcal{E}^{syn})$ with $\mathcal{V}^{syn} = V$ and $\mathcal{E}^{syn} = \mathcal{E}_P^{syn} \cup \mathcal{E}_F^{syn}$, where $(u,v) \in \mathcal{E}_P^{syn}$ holds if, and only if, u depends syntactically on the current valuation of v, and $(u,v) \in \mathcal{E}_F^{syn}$ holds if, and only if, u depends syntactically on the future valuation of v. Edges in $\mathcal{E}_F^{syn}$ are annotated with the offset of the corresponding syntactic dependency. Edges in $\mathcal{E}_P^{syn}$ are annotated with 0. We refer to the annotation of edge $(u,v) \in \mathcal{E}_F^{syn}$ with $a(u,v)$.

The syntactic dependency graph thus captures the syntactic dependencies induced by the specification. Note here that, in contrast to the semantic dependency graph, we do not distinguish edges to output variables and edges to input variables since the definition of syntactic dependencies, as opposed to the one for semantic dependencies, does not differ for them.

As outlined in the previous section, not all semantic dependencies necessarily have a syntactic counterpart. Hence, there might be semantic dependencies that are not represented by any edges in the syntactic dependency graph $\mathcal{D}_\varphi^{syn}$ of an LTL formula φ that contains conjunctions. If we directly proceed as in the semantic case with removing inputs and computing strongly connected components, which then define the components, admissibility of the computed components is thus not necessarily guaranteed for LTL specifications containing conjunctions. Therefore, we first build the *transitive output closure* over output variables of the syntactic dependency graph $\mathcal{D}_\varphi^{syn}$ of the LTL specification φ:

Definition 6.7 (Transitive Output Closure)**.**
Let φ be an LTL formula over atomic propositions V. Let $\mathcal{D}_\varphi^{syn} = (\mathcal{V}^{syn}, \mathcal{E}^{syn})$ be the syntactic dependency graph of φ with $\mathcal{E}^{syn} = \mathcal{E}_P^{syn} \cup \mathcal{E}_F^{syn}$. The *transitive output closure* $\mathbb{C}(\mathcal{D}_\varphi^{syn})$ of $\mathcal{D}_\varphi^{syn}$ is the graph $\mathbb{C}(\mathcal{D}_\varphi^{syn}) = (\mathcal{V}^{syn}, \mathbb{C}(\mathcal{E}^{syn}))$ with $\mathbb{C}(\mathcal{E}^{syn}) = \mathbb{C}(\mathcal{E}_P^{syn}) \cup \mathbb{C}(\mathcal{E}_F^{syn})$, where

- $(u,v) \in \mathbb{C}(\mathcal{E}_P^{syn})$ if, and only if, $u, v \in O$ and either $(u,v) \in \mathcal{E}_P^{syn}$ or, for some $m \in \mathbb{N}_0$ with $m > 0$, there exist outputs $u_1, \dots, u_m \in O$ with $(u, u_1) \in \mathcal{E}_P^{syn}$, $(u_m, v) \in \mathcal{E}_P^{syn}$, and $(u_i, u_{i+1}) \in \mathcal{E}^{syn}$ for all $i \in \mathbb{N}_0$ with $1 \leq i \leq m$. The edge (u,v) is annotated with 0.
- $(u,v) \in \mathbb{C}(\mathcal{E}_F^{syn})$ if, and only if, $u, v \in O$ and either $(u,v) \in \mathcal{E}_F^{syn}$ or, for some $m \in \mathbb{N}_0$ with $m > 0$, there exist outputs $u_1, \dots, u_m \in O$ with $(u, u_1) \in \mathcal{E}^{syn}$, $(u_m, v) \in \mathcal{E}^{syn}$, and $(u_i, u_{i+1}) \in \mathcal{E}^{syn}$ for all $i \in \mathbb{N}_0$ with $1 \leq i < m$, and either $(u, u_1) \in \mathcal{E}_F^{syn}$, $(u_m, v) \in \mathcal{E}_F^{syn}$, or $(u_i, u_{i_1}) \in \mathcal{E}_F^{syn}$ for some $i \in \mathbb{N}_0$ with $1 \leq i < m$. If $(u,v) \in \mathcal{E}_F^{syn}$, then we annotate the corresponding edge in $\mathbb{C}(\mathcal{E}_F^{syn})$ with $a(u,v)$. If $(u,v) \notin \mathcal{E}_F^{syn}$ and if there are $j, \ell \in \mathbb{N}_0$ with $j < \ell$ and $1 \leq j < m$ such that $u_j = u_\ell$ and $(u_i, u_{i+1}) \in \mathcal{E}_F^{syn}$ holds for all $i \in \mathbb{N}_0$ with $j \leq i < \ell$, then we annotate (u,v) in $\mathbb{C}(\mathcal{E}_F^{syn})$ with ∞. Otherwise, we annotate the edge (u,v) in $\mathbb{C}(\mathcal{E}_F^{syn})$ with $a(u, u_1) + a(u_m, v) + \sum_{1 \leq i < m} a(u_i, u_{i+1})$.

Intuitively, we thus add edges between output variables between which no syntactic dependency exists but which are connected via syntactic dependencies of other output variables to the dependency graph by building the transitive output closure. We obtain additional present edges if all of the connecting dependencies are present dependencies. Otherwise, we obtain additional future edges. The annotation of the future edges is computed from the annotations of the connecting edges. If the connecting edges contain a cycle of solely future edges, then the annotation is ∞ as the effect of changing the valuation of the variable represented by the source node of the edge can be delayed indefinitely. Otherwise, the annotation is the sum of the annotations of all connecting edges.

Example 6.9. Reconsider the autonomous car from the running example in Section 6.1 and its specification φ_{car}. Recall that, as outlined in Example 6.7, the specification φ_{car} induces, among others, syntactic present dependencies between acc and dec as well as a a syntactic future dependency from acc to itself with offset 1 and two future dependencies from acc to g_1 with offsets 1 and 2, respectively. These dependencies constitute present and future edges in the syntactic dependency graph $\mathcal{D}^{syn}_{\varphi_{car}} = (\mathcal{V}^{syn}, \mathcal{E}^{syn})$. Building the transitive output closure $\mathbb{C}(\mathcal{D}^{syn}_{\varphi_{car}})$ of the syntactic dependency graph $\mathcal{D}^{syn}_{\varphi_{car}}$ adds the following edges: two future edges $(dec, g_1) \in \mathbb{C}(\mathcal{E}^{syn}_F)$ with annotations $a(dec, g_1) = 1$ and $a(dec, g_1) = 2$, respectively, a future edge $(acc, acc) \in \mathbb{C}(\mathcal{E}^{syn}_F)$ with annotation $a(acc, acc) = \infty$, a future edge $(acc, g_1) \in \mathbb{C}(\mathcal{E}^{syn}_F)$ with annotation $a(acc, g_1) = \infty$, a future edge $(acc, dec) \in \mathbb{C}(\mathcal{E}^{syn}_F)$ with annotation $a(acc, dec) = \infty$, a future edge $(dec, acc) \in \mathbb{C}(\mathcal{E}^{syn}_F)$ with annotation $a(dec, acc) = \infty$, and, lastly, a future edge $(dec, g_1) \in \mathbb{C}(\mathcal{E}^{syn}_F)$ with annotation $a(dec, g_1) = \infty$. Except the first two edges from dec to g_1, all transitive edges are annotated with ∞ since the edge sequences from which they are built contain the future edge from acc to itself and thus a cycle. Note that when considering the full syntactic dependency graph induced by the specification φ_{car} and not only a few of its edges, some of the edges derived with transitivity above might already be contained in the initial graph due to dependencies induced by other conjuncts. Then, they are not added during the transitive closure and we will not refer to them as transitive edges in the following but as original dependency edges. △

The transitive output closure $\mathbb{C}(\mathcal{D}^{syn}_{\varphi})$ of the syntactic dependency graph $\mathcal{D}^{syn}_{\varphi}$ of an LTL specification φ reflects the syntactic dependencies induced by φ as well as the immediate transitive dependencies resulting from the syntactic dependencies. Transitive dependencies are a first step toward recognizing semantic dependencies that range over multiple conjuncts.

In a second step, we further capture the synergies of dependencies. Intuitively, synergies arise if two variables $u, v \in V$ both depend syntactically on the future valuation of a third variable $w \in V$. Then, further dependencies between u and v, which are not derived by the transitive output closure, can exist. If a change in the valuation of u at a point in time $k \geq 0$ requires a change in the valuation of w at a later point in time k' with $k' > k$, this might, for instance, induce the need for a change of the valuation of v at a point in time k'' with $k \leq k'' \leq k'$.

Example 6.10. Consider a system with three output variables $u, v, w \in V$ and the LTL specification $\varphi = \square(u \rightarrow \bigcirc w) \wedge \square(\bigcirc w \rightarrow v)$. It is clearly satisfied by the sequence $\sigma = \emptyset^\omega$, while it is violated by the sequence $\sigma' = \{u\}\emptyset^\omega$ since $\sigma' \not\models \square(u \rightarrow \bigcirc w)$ holds. The only possibility to satisfy the first conjunct of φ is to set w to *true* in the second time step. Then, however, v needs to be set to *true* in the first time step as well. Hence, $P = \{v\}$, $F = \{w\}$ is a minimal satisfying change set for φ, σ', u, and 0 and therefore u also depends semantically on the current valuation of v. Hence, we need to derive a further dependency to capture this synergy between the dependencies from u to w and from v to w. △

The synergies arising from two such dependencies depend on the scope of the dependencies in the sense of whether the dependencies are due to bounded or unbounded temporal operators. Therefore, the synergies depend on the annotation of the respective edges since the annotations accurately capture the "distance" of the dependency, which is given by ∞ for unbounded operators and by the number of $\bigcirc$-operators otherwise. Formally, we derive further dependencies between variables based on existing edges in the transitive closure of the syntactic dependency graph as follows:

Definition 6.8 (Extended Syntactic Dependency Graph).
Let φ be an LTL formula over atomic propositions V. Let $\mathcal{D}_\varphi^{syn} = (\mathcal{V}^{syn}, \mathcal{E}^{syn})$ be the syntactic dependency graph of φ and let $\mathbb{C}(\mathcal{D}_\varphi^{syn}) = (\mathcal{V}^{syn}, \mathbb{C}(\mathcal{E}^{syn})$ with $\mathbb{C}(\mathcal{E}^{syn}) = \mathbb{C}(\mathcal{E}_P^{syn}) \cup \mathbb{C}(\mathcal{E}_F^{syn})$ be its transitive output closure. The *extended syntactic dependency graph* $\mathcal{D}_\varphi^{xsyn}$ is defined by $\mathcal{D}_\varphi^{xsyn} = (\mathcal{V}^{syn}, \mathcal{E}^{xsyn})$ with $\mathcal{E}^{xsyn} = \mathcal{E}_P^{xsyn} \cup \mathcal{E}_F^{xsyn}$, where we have $\mathbb{C}(\mathcal{E}_P^{syn}) \subseteq \mathcal{E}_P^{xsyn}$ and $\mathbb{C}(\mathcal{E}_F^{syn}) \subseteq \mathcal{E}_F^{xsyn}$. Furthermore, we add edges to $\mathcal{E}_P^{xsyn}$ and $\mathcal{E}_F^{xsyn}$ as follows. Let $u, v, w \in \mathcal{V}^{syn}$ with $u, w \in O$ and either $u \neq v$ or $u \neq w$ such that $(u, w), (v, w) \in \mathbb{C}(\mathcal{E}_F^{syn})$ holds.

- If $a(u, w) = \infty$, then add $(u, v), (v, u) \in \mathcal{E}_P^{xsyn}$ and add $(u, v), (v, u) \in \mathcal{E}_F^{xsyn}$, both future transitions annotated with ∞ and the present edge annotated with 0.
- If $a(u, w) = a(v, w) \neq \infty$ holds, then add $(u, v), (v, u) \in \mathcal{E}_P^{syn}$ with annotation 0.
- If $\infty \neq a(u, w) < a(v, w) \neq \infty$ holds, then add $(v, u) \in \mathcal{E}_F^{xsyn}$ with annotation $y - x$.
- If $\infty \neq a(u, w) > a(v, w) \neq \infty$ holds, then add $(u, v) \in \mathcal{E}_F^{xsyn}$ with annotation $x - y$.

The extended syntactic dependency graph thus also captures dependencies between two variables that stem from shared dependencies of these two variables to a third variable. Hence, it captures implicit dependencies, which arise from the synergies of several syntactic dependencies, allowing for recognizing semantic dependencies that range over multiple conjuncts. Afterward, we then build the transitive output closure $\mathbb{C}(\mathcal{D}_\varphi^{xsyn})$ of the extended syntactic dependency graph of the LTL specification φ. We proceed with deriving further edges according to Definition 6.8 and building the transitive output closure afterward until a fixpoint is reached, i.e., until neither of the techniques yields new edges. Slightly overloading notation, we denote the resulting dependency graph with $\mathbb{C}(\mathcal{D}_\varphi^{xsyn})$ as well.

Example 6.11. Consider the autonomous car from the running example described in Section 6.1 and its specification φ_{car}. The transitive output closure $\mathbb{C}(\mathcal{D}_\varphi^{xsyn})$ of the extended syntactic dependency induced by φ_{car} is depicted in Figure 6.6. Dashed edges represent present dependencies; solid edges represent future dependencies. Since all present edges are annotated with 0 by construction, we omit the annotations of present edges for the sake of readability. Furthermore,

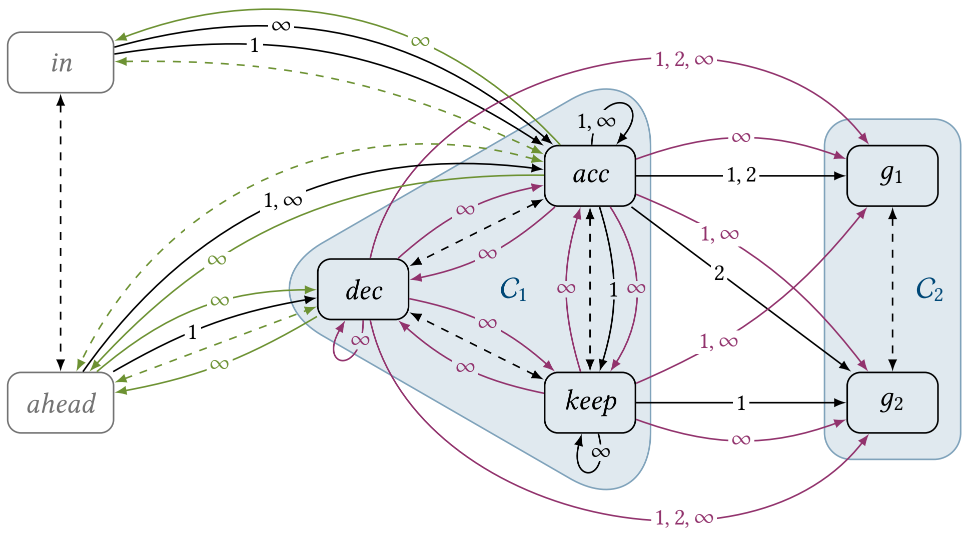


Figure 6.6.: Transitive output closure $\mathbb{C}(\mathcal{D}_\varphi^{xsyn})$ of the extended syntactic dependency graph induced by the specification φ_{car} of the running example. Nodes representing input variables are depicted in gray. Dashed edges represent present edges; solid ones represent future edges. We omit the annotation 0 for present edges for readability and merge future edges with the same source and successor but different annotations into a single one with multiple annotations. Black edges are induced by syntactic dependencies, violet ones are obtained by transitivity, and green ones are derived edges. For readability, we omit derived edges between *acc*, *dec* and *keep* as well as derived edges from *dec* to *in* and from *keep* to both *in* and *ahead*. Furthermore, we omit edges obtained with transitivity after derivation. The strongly connected components of the output variables are highlighted in blue.

we merge future edges with the same source and successor node but different annotations into a single future edge with multiple annotations. The edges are highlighted in different colors, depending on in which step of the construction of $\mathbb{C}(\mathcal{D}_\varphi^{xsyn})$ they are added. Black edges denote syntactic dependencies, i.e., they represent dependencies that are already contained in the syntactic dependency graph $\mathcal{D}_\varphi^{syn}$. Violet edges are added to the graph by building the transitive output closure of $\mathcal{D}_\varphi^{syn}$, i.e., they represent dependencies that are already contained in $\mathbb{C}(\mathcal{D}_\varphi^{syn})$. Green edges are derived edges, i.e., they represent dependencies that are already contained in the extended syntactic dependency graph $\mathcal{D}_\varphi^{xsyn}$. For the sake of readability, we omit derived edges between *acc*, *dec* and *keep* as well as derived edges from *dec* to *in* and from *keep* to both *in* and *ahead*. Furthermore, we omit edges obtained from transitivity after derivation. Unlike the semantic dependency graph induced by the car's specification φ_{car} depicted in Figure 6.3, the transitive output closure $\mathbb{C}(\mathcal{D}_\varphi^{xsyn})$ of the extended syntactic dependency contains outgoing edges from nodes representing input variables of the system, i.e., from the nodes representing *in* and *ahead*. While such dependencies are irrelevant for component selection and thus do not constitute semantic dependencies, they are needed to derive dependencies *to* input variables with the syntactic component selection technique. Observe that, after derivation, the transitive output closure $\mathbb{C}(\mathcal{D}_\varphi^{xsyn})$ of the extended syntactic dependency contains all edges that are contained in the semantic dependency graph depicted in Figure 6.3. Before derivation, however, it misses dependencies from output variables to input variables. △

Utilizing the transitive output closure of the extended syntactic dependency graph of the LTL formula φ, i.e., the graph $\mathbb{C}(\mathcal{D}_\varphi^{xsyn})$, we now identify suitable components and the ranking function $rank_{syn}$, which defines the synthesis order, as for semantic dependencies. That is, we eliminate vertices of $\mathbb{C}(\mathcal{D}_\varphi^{xsyn})$ that represent input variables, then compute the set of strongly connected components, which constitute the components, and then define the ranking function $rank_{syn}$ according to the edges between the strongly connected components. For the autonomous car from the running example, for instance, we obtain the very same two components from the dependency graph depicted in Figure 6.6 as with the semantic decomposition approach, namely one component controlling *acc*, *dec*, and *keep*, i.e., a component modeling the acceleration unit, and a component controlling g_1 and g_2, i.e., a component modeling the gearing unit. The gearing unit has a lower rank in the synthesis order than the acceleration unit.

Note here that we can additionally resolve present dependencies as in the semantic decomposition approach (see Section 6.3.3) before building the strongly connected components of the output variables. In particular, we utilize the implementation order to eliminate present dependency edges unidirectionally whenever possible. For instnace, in the transitive output closure of the extended dependency graph for the autonomous car depicted in Figure 6.6, we can resolve one of the present edges between g_1 and g_2, allowing for splitting the gearing unit into two separate parts, one controlling g_1 and one controlling g_2.

Clearly, as we do not alter the computation of the ranking function $rank_{syn}$ defining the synthesis order with respect to the semantic approach, the resulting ranking function for both variants of the syntactic decomposition technique is compositionality-preserving as well:

Lemma 6.5. *Let φ be an LTL formula over atomic propositions V. Let $\mathbb{D}$ and $rank_{syn}$ be the decomposition and the ranking function computed with syntactic decomposition algorithm. Then, the synthesis order $<_{syn}$ induced by $\mathbb{D}$ and $rank_{syn}$ is compositionality-preserving.*

Therefore, soundness of incremental synthesis when deriving the components as well as the ranking function with the syntactic decomposition algorithm follows, as for the semantic decomposition algorithm, from Lemma 6.5 and Theorem 6.2.

Corollary 6.2. *Let φ be an LTL formula over atomic propositions V. Let $\mathbb{D} = \langle c_1, \ldots, c_n \rangle$ and $rank_{syn}$ be the decomposition and the ranking function computed with the syntacitc decomposition algorithm. Suppose that Algorithm 6.1 returns* $(\texttt{true}, s)$ for input φ, $\mathbb{D}$, and $rank_{syn}$. Then, strategy s is dominant for φ.

Next, we consider completeness of incremental synthesis for the syntactic decomposition algorithm. After deriving further dependencies from the syntactic dependencies constituting edges in the syntactic dependency graph $\mathcal{D}_\varphi^{syn}$ of the specification φ via transitive output closure, building the extended dependency graph, and repeating derivation and transitivity until a fixpoint is reached, every semantic dependency has a syntactic counterpart – even if it ranges over multiple conjuncts. Intuitively, the properties of a minimal satisfying change set, which then constitutes a semantic dependency, induce several syntactic present and future dependencies that only affect single conjuncts of the specification. Utilizing Lemma 6.4, the claim then follows by induction on the number of these separate dependencies.

Theorem 6.7. *Let φ be an LTL formula over atomic propositions V. Let $\mathbb{C}(\mathcal{D}_{\varphi}^{xsyn}) = (V, \mathbb{C}(\mathcal{E}^{xsyn}))$ with $\mathbb{C}(\mathcal{E}^{xsyn}) = \mathbb{C}(\mathcal{E}_P^{xsyn}) \cup \mathbb{C}(\mathcal{E}_F^{xsyn})$ be the transitive output closure of the extended syntactic dependency graph of φ. Let $u \in V$. If u depends semantically on the current valuation of a variable $v \in O$, then we have $(u,v) \in \mathbb{C}(\mathcal{E}_P^{xsyn})$. If u depends semantically on the future valuation of a variable $v \in O$, then we have $(u,v) \in \mathbb{C}(\mathcal{E}_F^{xsyn})$. If u depends semantically on the input, then there exist variables $w \in O$, $w' \in I$ such that $(w,w') \in \mathbb{C}(\mathcal{E}^{xsyn})$ holds.*

Proof. First, suppose that u depends semantically on the current or future valuation of an output variable $v \in O$. Then, there exists a point in time $k \geq 0$ and sequences $\sigma, \sigma' \in (2^V)^\omega$ such that $\sigma_k \cap \{u\} \neq \sigma'_k \cap \{u\}$ holds, while we have $\sigma_k \cap (V \setminus \{u\}) = \sigma'_k \cap (V \setminus \{u\})$ and while $\sigma_j = \sigma'_j$ holds for all $j \geq 0$ with $j < k$, and such that we have $\sigma \models \varphi$ and $\sigma' \not\models \varphi$. Thus, there exists a conjunct φ_i of φ that is violated by σ', while it is satisfied by σ. Furthermore, there are sets $P \subseteq V \setminus \{u\}$ and $F \subseteq V$ such that (P,F) is a minimal satisfying change set for φ, σ', u, and k and we have $v \in P \cup F$. Therefore, by definition of satisfying change sets, there exists a sequence $\sigma'' \in (2^V)^\omega$ that adheres to the requirements defined in Definition 6.2 and that satisfies φ. Hence, the violation of conjunct φ_i by σ' can intuitively be fixed by changing the valuations of the variables in P at point in time k and the valuations of the variables in F at their respective points in time $j \geq 0$ with $j > k$. Note that not all of these changes are necessarily needed to obtain satisfaction of φ_i. Rather, satisfying φ_i may introduce violations of different conjuncts φ_j of the full specification φ, which would yield a violation of φ, and which need to be averted by further changes of variable valuations.

We, therefore, introduce the notion of *violation clusters*. A violation cluster C is a set of conjuncts of φ where all conjuncts are violated by the same change in the valuation of a variable. In particular, the cluster C_1 contains all conjuncts $\varphi_1^1, \ldots \varphi_{m_1}^1$ that are violated by σ', while they are satisfied by σ. To satisfy these conjuncts again, changes in variables are needed that may introduce violations of different conjuncts. The cluster $C_{1 \cdot i}$ contains the conjuncts $\varphi_1^{1 \cdot i}, \ldots, \varphi_{m_{1 \cdot i}}^{1 \cdot i}$ that are violated by the changes needed to satisfy φ_i^1 and so on. This induces a tree-like structure of violation clusters. A conjunct of φ may occur in different violation clusters.

For the formal definition of violation clusters, we utilize a slightly modified version of a minimal satisfying change set. It is defined by a triple (H,P,F) for a specification φ', a sequence $\sigma'''' \in (2^V)^\omega$, a variable $u' \in V$, and a point in time $k' \geq 0$. It poses similar requirements as a minimal satisfying change set. However, it does not require a sequence to agree with σ'''' up to point in time k', but only to point in time k, i.e., the point in time at which σ differs from σ' in u. Furthermore, the change of u at point in time k needs to be preserved. All variables that change their valuations between points in time k and $k'-1$ are then captured in H. Hence, the set H intuitively captures history dependencies. We call such a triple (H,P,F) a history-adapting satisfying change set for φ', σ'''', u', and k'. We denote the set of all sequences $\sigma \in (2^V)^\omega$ that satisfy the requirements of a history-adapting satisfying change set (H,P,F) with $\Sigma^{H,P,F}$.

More precisely, we define violation clusters as follows. The initial violation cluster C_1 is defined by $C_1 = \{\varphi_i \in \varphi \mid \sigma \models \varphi \wedge \sigma' \not\models \varphi_i\}$. Recall here that we also represent conjunctive LTL formulas as the set containing all their conjuncts, and thus $\varphi_i \in \varphi$ denotes that φ_i is a conjunct of φ. For every conjunct $\varphi^1_i \in C_1$, there then exists a minimal history-adapting satisfying change set (H^1_i, P^1_i, F^1_i) for φ^1_i, σ', u, and k such that $P^1_i \subseteq P$ and $F^1_i \subseteq F$ holds. Note that since we consider the point in time k here, which is also the point in time at which σ differs from σ' in u, we have $H^1_i = \emptyset$. Let $\mathcal{S}$ be the set of those minimal history-adapting satisfying change sets. Based on $\mathcal{S}$, we define the violation cluster $C_{1\cdot i}$ as

$$C_{1\cdot i} = \left\{\varphi_j \in \varphi \mid \sigma \models \varphi_j \wedge \exists (H^1_i, P^1_i, F^1_i) \in \mathcal{S}.\, \exists \sigma'' \in \Sigma^{H^1_i, P^1_i, F^1_i}.\, \sigma'' \models \varphi^1_i \wedge \sigma'' \not\models \varphi_j\right\}.$$

Given a violation cluster $C_j = \{\varphi^j_1, \dots, \varphi^j_{m_j}\}$ with $j = \ell \cdot i'$, we define its successor for conjunct $\varphi^j_i \in C_j$ as follows. Let $\sigma^j \in (2^V)^\omega$ be a sequence with $\sigma^j \in \Sigma^{H^\ell_{i'}, P^\ell_{i'}, F^\ell_{i'}}$ such that $\sigma^j \models \varphi^\ell_{i'}$ holds while we have $\sigma^j \not\models \varphi^j_1 \vee \ldots \vee \varphi^j_{m_j}$. Let $u^j_i \in H^\ell_{i'} \cup P^\ell_{i'} \cup F^\ell_{i'}$ be some variable and let k^j_i be some point in time with $k^j_i \geq k$ that establishes that u^j_i is part of the minimal history-adapting satisfying change set $(H^\ell_{i'}, P^\ell_{i'}, F^\ell_{i'})$, i.e., σ^j differs from the sequence σ^ℓ, from which $(H^\ell_{i'}, P^\ell_{i'}, F^\ell_{i'})$ is built. Note that there exists some $\sigma^j \in \Sigma^{H^\ell_{i'}, P^\ell_{i'}, F^\ell_{i'}}$ such that there exists a sequence $\tilde{\sigma}^{j,i}$ such that $\tilde{\sigma}^{j,i}$ differs from σ^j only in the valuation of the variable u^j_i with $u^j_i \in H^\ell_{i'} \cup P^\ell_{i'} \cup F^\ell_{i'}$, which is used for constructing the violation cluster $C_{j\cdot i}$, and such that $\tilde{\sigma}^{j,i} \models \varphi$ holds. By assumption, there exists a sequence that satisfies all conjuncts, and thus, in particular, there is one that satisfies all conjuncts of previous violation clusters as well as φ_i. Suppose that there is no sequence that differs from σ^j only in one variable valuation at a single point in time. Let $\mathcal{M}$ be a minimal set of changes that need to performed to obtain a sequence $\hat{\sigma}$ from σ^j that satisfies φ^j_i. Then, performing only a subset of the reverse changes on $\hat{\sigma}$ cannot result in satisfaction of φ^j_i as otherwise $\mathcal{M}$ would not be minimal. However, in particular, a sequence that is obtained from $\hat{\sigma}$ by only performing a single reverse change of the changes in $\mathcal{M}$ violates φ^j_i; yielding that either $(H^\ell_{i'}, P^\ell_{i'}, F^\ell_{i'})$ is not minimal – if we do not need to perform changes defined by $(H^\ell_{i'}, P^\ell_{i'}, F^\ell_{i'})$ – or that there exists another sequence in $\Sigma^{H^\ell_{i'}, P^\ell_{i'}, F^\ell_{i'}}$ that violated φ^j_i and thus constitutes that φ^j_i lies in C_j – if all changes are addressed, but the variable of the single change lies either in $H^\ell_{i'}$ or in $F^\ell_{i'}$ and thus allows for changes at several positions.

Let σ^j be such a particular sequence with $\sigma^j \in \Sigma^{H^\ell_{i'}, P^\ell_{i'}, F^\ell_{i'}}$ and let $\tilde{\sigma}^{j,i}$ be the respective sequence with $\tilde{\sigma}^{i,j} \models \varphi$. Then, there is a minimal history-adapting satisfying change set (H^j_i, P^j_i, F^j_i) for φ^j_i, σ^j, u^j_i, and k^j_i such that (H^j_i, P^j_i, F^j_i) respects (P, F) in the sense that all variables that require changes at point in time k are contained in P and that all variables that require changes at a later point in time are contained in F. Let $\mathcal{S}$ be the set of such minimal satisfying change sets. Similar to $C_{1\cdot i}$, we then define the violation cluster $C_{j\cdot i}$ as

$$C_{j\cdot i} = \left\{\varphi_j \in \varphi \mid \sigma^\ell \models \varphi_j \wedge \exists (P^j_i, F^j_i) \in \mathcal{S}.\, \exists \sigma'' \in \Sigma^{P^j_i, F^j_i}.\, \sigma'' \models \varphi^j_i \wedge \sigma'' \not\models \varphi_j\right\}.$$

Note that a sequence $\sigma'' \in \Sigma^{H^j_i, P^j_i, F^j_i}$ might rely on changing variable valuations that have already been changed with respect to σ to obtain the sequence σ^j. Then, however, these variables have to be contained in either H_i^j, P_i^j or F_i^j, depending on the point in time at which the change occurs. Hence, the change set (H_i^j, P_i^j, F_i^j) also premits the initial changes. Furthermore, since there exists a sequence that satisfies all conjuncts of φ by assumption, namely one from the set $\Sigma^{P,F}$, we can choose $\sigma^j \in \Sigma^{H_{i'}^\ell, P_{i'}^\ell, F_{i'}^\ell}$ such that only changing the variable valuations back while not performing the changes that yield satisfaction of φ_i^j while keeping the already performed changes with respect to σ is not minimal.

If all branches of the tree-like structure induced by change of the valuation of u at point in time k in σ end since the successor violation cluster is empty, we check whether all conjuncts of φ are contained in a violation cluster. If not, we proceed as follows. Let $\mathcal{V} \subseteq \varphi$ be the set of all conjuncts of φ that are contained in one of the violation clusters. Let $\tilde{\sigma} \in (2^V)^\omega$ be a sequence obtained from incorporating all the variable changes performed in the sequence of the violation clusters. Since there exists a sequence that satisfies all conjuncts by assumption and since $\mathcal{V}$ is a subset of all conjuncts, the existence of such a sequence $\tilde{\sigma}$ is guaranteed. Let $\varphi_i^2 \in \varphi \setminus \mathcal{V}$ be a conjunct that is not contained in any violation cluster. If $\tilde{\sigma} \models \varphi_i^2$, then nothing has to be done in this construction step. Otherwise, let $\tilde{\sigma}^{2,i} \in (2^V)^\omega$ be a sequence such that $\tilde{\sigma}^{2,i} \models \varphi$ and such that $\tilde{\sigma}^{2,i}$ differs from $\tilde{\sigma}$ only in the valuation of a variable, that has been changed from σ to $\tilde{\sigma}$ at a single point in time. Similar to the fact that sequences $\tilde{\sigma}^{j,i}$ with these properties are guaranteed in the first construction round, it follows that such a sequence is guaranteed to exist. The violation cluster C_2 then contains all conjuncts $\varphi_i \in \varphi \setminus \mathcal{V}$, which are violated by $\tilde{\sigma}$, as well as by some sequence $\tilde{\sigma}^{2,i}$ with the above properties. We proceed with constructing violation clusters of the form $C_{2\cdot i}$ analogous to those of the form $C_{1\cdot i}$, i.e., those constructed in the first round. Moreover, we proceed with constructing cluster C_3, C_4 etc. until all conjuncts of φ are contained in one of the violation clusters.

We now show by induction that for every violation cluster $C_{j\cdot i}$ and its associated minimal history-adapting satisfying change set (H_i^j, P_i^j, F_i^j) it holds that if $k_j^i = k$, then there exist edges $(u,w) \in \mathbb{C}(\mathcal{E}_P^{xsyn})$ for all $w \in P_i^j$ and edges $(u,w) \in \mathbb{C}(\mathcal{E}_F^{xsyn})$ for all $w \in F_i^j$, and, if $k_j^i > k$ holds, then there exist edges $(u,w) \in \mathbb{C}(\mathcal{E}_F^{xsyn})$ for all $w \in H_i^j \cup P_i^j \cup F_i^j$ and edges $(u,w) \in \mathbb{C}(\mathcal{E}_F^{xsyn})$ for all $w \in H_i^j$ that are changed at point in time k:

- $j = 1$. Then, for every $\varphi_i \in C_1$, the triple (H_j^1, P_i^1, F_i^1) is a minimal history-adapting satisfying change set for φ_i^1, σ', u, and k. Thus, in particular, $H_j^1 = \emptyset$ holds. Furthermore, σ and σ' only differ in the valuation of u at point in time k. Thus, since φ_i^1 is a single conjunct of φ and since, by assumption, conjunctions of φ itself are conjunction-free, it follows immediately with Lemma 6.4 that $(u,w) \in \mathcal{E}_P^{syn}$ holds for all variables $w \in P_i^1$ and $(u,w) \in \mathcal{E}_F^{syn}$ holds for all variables $w \in F_i^1$. Since transitive output closure as well as deriving further edges in the extended syntactic dependency graph always only add edges but never delete edges, it follows that $(u,w) \in \mathbb{C}(\mathcal{E}_P^{xsyn})$ holds for all variables $w \in P_i^1$ and $(u,w) \in \mathbb{C}(\mathcal{E}_F^{xsyn})$ holds for all variables $w \in F_i^1$.
- $j = \ell$ with $\ell \neq 1$. Then, for every conjunct $\varphi_i \in C_j$, the triple (H_i^j, P_i^j, F_i^j) is a minimal history-adapting satisfying change set for φ_i^1, σ^j, u_i^j, and k_i^j. Furthermore, u_i^j is a variable that lies in the change set $(H_{i'}^\ell, P_{i'}^\ell, F_{i'}^\ell)$ defining the violation cluster C_j. Thus, by induction

hypothesis, we have either $(u, u_i^j) \in \mathbb{C}(\mathcal{E}_P^{xsyn})$ or $(u, u_i^j) \in \mathbb{C}(\mathcal{E}_F^{xsyn})$, depending on whether $k_i^j = k$ or $k_i^j > k$ holds. We distinguish two cases:

- First, suppose that $k_i^j = k$ holds. Then, it follows immediately that $u_i^j \in P_{i'}^{\ell}$ holds. Therefore, there exists a present edge $(u, u_i^j) \in \mathbb{C}(\mathcal{E}_P^{xsyn})$. Moreover, by construction of the violation clusters as well as their associated sequences, variables, and points in time, we have $\tilde{\sigma}^{j,i} \models \varphi_i$, while $\sigma^j \not\models \varphi_i$ holds and $\tilde{\sigma}^{j,i}$ and σ^j only differ on the valuation of u_i^j at a single point in time k_i^j. Hence u_i^j depends semantically on the current valuation of all variables $w \in P_i^j$, and it depends semantically on the future valuation of all variables $w \in F_i^j$. Therefore, it follows immediately with Lemma 6.4 that $(u_i^j, w) \in \mathcal{E}_P^{syn}$ holds for all variables $w \in P_i^1$ and $(u_i^j, w) \in \mathcal{E}_F^{syn}$ holds for all variables $w \in F_i^1$. Since transitive output closure as well as deriving further edges in the extended syntactic dependency graph always only adds edges but never delete edges, it follows that $(u_i^j, w) \in \mathbb{C}(\mathcal{E}_P^{xsyn})$ holds for all variables $w \in P_i^j$ and $(u_i^j, w) \in \mathbb{C}(\mathcal{E}_F^{xsyn})$ holds for all variables $w \in F_i^j$. Since we build the transitive output closure of the syntactic dependency graph as well as the extended syntactic dependency graph, we thus have $(u, w) \in \mathbb{C}(\mathcal{E}_P^{xsyn})$ for all variables $w \in P_i^j$ and $(u, w) \in \mathbb{C}(\mathcal{E}_F^{xsyn})$ for all variables $w \in F_i^j$.

- Second, suppose that $k_i^j > k$ holds. Then, $u_i^j \in F_{i'}^{\ell}$ holds and hence $(u, u_i^j) \in \mathbb{C}(\mathcal{E}_F^{xsyn})$ follows. Moreover, by construction of the violation clusters as well as their associated sequences, variables, and points in time, we have $\tilde{\sigma}^{j,i} \models \varphi_i$, while $\sigma^j \not\models \varphi_i$ holds and $\tilde{\sigma}^{j,i}$ and σ^j only differ on the valuation of u_i^j at a single point in time k_i^j. We distinguish three cases. First, let $w \in P_i^j$. Then, by definition of history-adapting satisfying change sets, the valuation of w is changed at point in time k_i^j. Thus, there exists a semantic present dependency from u_i^j to w. Therefore, $(u_i^j, w) \in \mathcal{E}_P^{syn}$ follows with Lemma 6.4 and hence $(u_i^j, w) \in \mathbb{C}(\mathcal{E}_P^{xsyn})$ holds as well. Since we build the transitive output closure of the extended syntactic dependency graph, $(u, w) \in \mathbb{C}(\mathcal{E}_F^{xsyn})$ follows. Next, let $w \in F_i^j$. Then, the valuation of w is changed at a point in time $k' \geq 0$ with $k' > k_i^j$. Thus, there exists a semantic future dependency from u_i^j to w. Therefore, $(u_i^j, w) \in \mathcal{E}_F^{syn}$ follows with Lemma 6.4 and hence $(u_i^j, w) \in \mathbb{C}(\mathcal{E}_F^{xsyn})$ holds as well. Since we build the transitive output closure of the extended syntactic dependency graph, $(u, w) \in \mathbb{C}(\mathcal{E}_F^{xsyn})$ follows. Lastly, let $w \in H_i^j$. Then, the valuation of w is changed at a point in time $k' \geq 0$ with $k \leq k' < k_i^j$. Hence, there exists a semantic future dependency from w to u_i^j: there is a sequence that lies in $\Sigma^{H_i^j, P_i^j, F_i^j}$ that satisfies φ_i^j. This sequence contains both the changes for u_i^j and w. The sequence obtained from reverting the change of w violates φ_i^j as otherwise, the change in w would not be necessary. Reverting the change of u_i^j as well – possibly together with more changes – yields sequence $\tilde{\sigma}_i^j$ which satisfies φ_i^j. Thus, u_i^j lies in the future-set of a minimal satisfying change set for w and φ_i^j, and thus the existence of a semantic future dependency from w to u_i^j

follows. Thus, by Lemma 6.4, there exists a future edge $(w, u_i^j) \in \mathcal{E}_F^{syn}$ and therefore we have $(w, u_i^j) \in \mathbb{C}(\mathcal{E}_F^{xsyn})$ as well. Since there are semantic future dependencies from u to u_i^j and from w to u_i^j, the valuations of u and w affect the future valuation of u_i^j. Furthermore, by construction of the semantic dependencies, they both affect the valuation of u_i^j at the same point in time. If w affects the future valuation of u_i^j due to an unbounded temporal operator, then we have $a(w, u_i^j) = \infty$ by definition of syntactic dependencies and since the semantic dependency from w to u_i^j is induced by the conjunction-free conjunct φ_i^j. Thus, since $w, u_i^j \in O$, as otherwise u would depend on the input, and since we have both $(u, u_i^j), (w, u_i^j) \in \mathbb{C}(\mathcal{E}_F^{xsyn})$ and since $a(w, u_i^j) = \infty$ holds, we derive further syntactic edges according to Definition 6.8, namely future edges $(u, w), (w, u) \in \mathbb{C}(\mathcal{E}_F^{xsyn})$ as well as present edges $(u, w), (w, u) \in \mathbb{C}(\mathcal{E}_P^{xsyn})$. Thus, irrespective of whether the valuation of w needs to be changed at point in time k or at a point in time k' with $k < k' < k_i^j$, we derive the required edge. Otherwise, w affects the future valuation of u_i^j only due to $\bigcirc$-operators. If, in contrast, u affects the future valuation of u_i^j due to unbounded temporal operators, then there exists some conjunct of φ that establishes that some of the variables that build the link between u and u_i^j are connected via an unbounded temporal operator. Hence, it follows similarly to the previous case that this conjunct then induces a syntactic future dependency with offset ∞. Since the transitive output closure preserves the annotation ∞, we thus have $a(u, u_i^j) = \infty$. Therefore, as in the previous case, we derive both $(u, w), (w, u) \in \mathbb{C}(\mathcal{E}_F^{xsyn})$ and $(u, w), (w, u) \in \mathbb{C}(\mathcal{E}_P^{xsyn})$. Thus, irrespective of whether the valuation of w needs to be changed at point in time k or at a point in time k' with $k < k' < k_i^j$, we derive the required edge. If u also does not affect the future valuation of u_i^j due to unbounded temporal operators but only due to $\bigcirc$-operators, the effect of the change of a valuation of u or w on the valuation of u_i^j is limited to the number of $\bigcirc$-operators. This number, however, is accurately captured by the offset of syntactic dependencies, which is represented by the annotation of the respective edge, and the accuracy of the edge annotations is clearly preserved by transitive output closure as well as derivation. Thus, both edges (u, u_i^j) and (w, u_i^j) are annotated with a natural number that captures the number of $\bigcirc$-operators inducing the semantic dependency. If the valuation of w needs to be changed at point in time k, then, since both dependencies affect the valuation of u_i^j at the same point in time, both dependencies are due to the same number of $\bigcirc$-operators and therefore $a(u, u_i^j) = a(w, u_i^j)$ holds. Hence, we derive, according to Definition 6.8, present edges $(u, w), (w, u) \in \mathbb{C}(\mathcal{E}_P^{xsyn})$. Thus, in particular, we derive the required edge for the present dependency from u to w. Otherwise, the valuation of w needs to be changed at a point in time k' with $k < k' < k_i^j$. Hence, the dependency from w to u_i^j is due to less $\bigcirc$-operators than the dependency from u to u_i^j. Therefore, holds $a(w, u_j^i) < a(u, u_i^j)$ for the annotations of the edges (w, u_i^j) and (u, u_i^j). Consequently, we derive, according to Definition 6.8, the future dependency edge $(u, w) \in \mathbb{C}(\mathcal{E}_F^{xsyn})$, which is the required future edge from u to w.

By construction of the violation clusters, every variable $w \in P \cup F$ is contained in the minimal history-adapting satisfying change set of at least one violation cluster. Thus, in particular, there exists a violation cluster $C_{j \cdot i}$ such that $v \in H_i^j \cup P_i^j \cup F_i^j$ holds. If $v \in P$ holds, then it follows immediately from the property shown above that there exists an edge $(u, v) \in \mathbb{C}(\mathcal{E}_P^{xsyn})$. If $v \in F$ holds, then there exists an edge $(u, v) \in \mathbb{C}(\mathcal{E}_F^{xsyn})$.

Next, suppose that u depends semantically on the current or future valuation of an input variable. Then, there exists a point in time $k \geq 0$ and sequences $\sigma, \sigma' \in (2^V)^\omega$ such that $\sigma_k \cap \{u\} \neq \sigma'_k \cap \{u\}$ holds, while we have $\sigma_k \cap (V \setminus \{u\}) = \sigma'_k \cap (V \setminus \{u\})$ and while $\sigma_j = \sigma'_j$ holds for all $j \geq 0$ with $j < k$, and such that we have $\sigma \models \varphi$ and $\sigma' \not\models \varphi$. Furthermore, we have $(P \cup F) \cap I \neq \emptyset$ for all minimal satisfying change sets (P, F) for φ, σ', u, and k, i.e., there is no minimal satisfying change set that does not contain at least one input variable. Let (P, F) be some minimal satisfying change set for φ, σ', u, and k. Then, there exists a sequence $\sigma'' \in \Sigma^{P,F}$ such that $\sigma'' \models \varphi$ holds. If there exists a conjunct $\varphi_i \in \varphi$ that is violated by σ' and that contains an input variable, then it follows immediately that u dependents semantically on some input variable $w \in I$ when only considering the single conjunct φ_i. Hence, then $(u, w) \in \mathcal{E}^{syn}$ holds by Lemma 6.4 and thus $(u, w) \in \mathbb{C}(\mathcal{E}_F^{xsyn})$ follows. Otherwise, we construct violation clusters as in the first case. Similarly, it follows that we have edges $(u, w) \in \mathbb{C}(\mathcal{E}^{xsyn})$ for all output variables $w \in O$ with $w \in H_i^j \cup P_i^j \cup F_i^j$ for some violation cluster $C_{j \cdot i}$ that did not require a change in any input variable beforehand. That is, for all predecessors of $C_{j \cdot i}$, the respective minimal history-adapting satisfying change sets do not contain any input variable. As soon as we encounter an input variable, we cannot utilize the above result since we do not build the transitive output closure over input variables. Nevertheless, we still obtain that there exists an edge (w, w') from the last output variable $w \in O$ to the first input variable $w' \in I$ with $(w, w') \in \mathbb{C}(\mathcal{E}^{xsyn})$; proving the claim. □

Thus, since all semantic dependencies have a syntactic counterpart, completeness of incremental synthesis, when using syntactic decomposition for deriving the components and the ranking function defining the synthesis order, for LTL specifications φ that do not induce any edge (u, v) in $\mathbb{C}(\mathcal{D}_\varphi^{xsyn})$ from an output variable $u \in O$ of the system to an input variable $v \in I$ of the system, follows from Lemma 6.3, and Theorem 6.7. When utilizing the version of syntactic decomposition that allows for resolving present dependencies, then the result follows from Theorem 6.7 and the respective version of Lemma 6.3, which is based on Theorem 6.6 rather than on Theorem 6.3, for the similarly modified version of semantic decomposition.

Lemma 6.6. *Let φ be an LTL formula over atomic propositions V. Let $\mathbb{D}$ and $rank_{syn}$ be the decomposition and the ranking function computed with syntactic decomposition algorithm. If, for all $u \in O$, output u does not depend semantically on an input variable $v \in I$, then Algorithm 6.1 returns* $(\texttt{true}, s)$ *for input* φ, $\mathbb{D}$, *and* $rank_{syn}$.

Furthermore, we can utilize the same extension of incremental synthesis as for the semantic decomposition algorithm to obtain completeness for general LTL formulas. That is, whenever we encounter a component of non-highest rank for which the synthesis task fails and which contains an output variable that is the source node of an edge in $\mathbb{C}(\mathcal{D}_\varphi^{xsyn})$ to a node representing an input variable, we combine the component with a direct successor in the synthesis order.

We proceed with combining components until either the synthesis task succeeds or until only a single component is left. For this extension of the incremental synthesis algorithms, completeness when utilizing the syntactic decomposition algorithms, in both its versions, then follows immediately from Theorems 6.5 and 6.7, or the respective variant of Theorem 6.5 for the modified semantic decomposition algorithm.

Theorem 6.8. *Let φ be an LTL formula over atomic propositions V. Let $\mathbb{D}$ and $rank_{syn}$ be the decomposition and the ranking function computed with semantic decomposition algorithm. If the extended incremental synthesis algorithm returns* $(\texttt{false}, \textsc{Null})$ for input φ, $\mathbb{D}$, and $rank_{syn}$, then φ is unrealizable.

However, the syntactic analysis is a conservative overapproximation of the semantic dependencies. This can be easily seen when comparing the semantic and syntactic dependency graphs for the self-driving car shown in Figures 6.3 and 6.6 respectively. For instance, there is a present edge from *acc* to *in* in the syntactic graph, while there is no such semantic dependency. Particularly the derivation rule is blamable for the overapproximation. Therefore, the syntactic decomposition algorithm may yield coarser decompositions. Nevertheless, as the syntactic analysis is much cheaper than the semantic one, decomposing the system with the syntactic approach and applying incremental synthesis to the derived components is an adequate first step. In many cases, the syntactic decomposition will already yield sufficiently small components for which solutions can be synthesized in reasonable time. If incremental synthesis does not terminate in reasonable time, however, one can then use the semantic decomposition technique to identify components of the system and, if the decomposition is more fine-grained, rerun incremental synthesis with the semantic decomposition.

We introduce rules for simplifying specifications in the next section to reduce the synthesis time for the individual components further. In particular, we identify in which cases conjuncts of the specification can be omitted when considering an individual component while maintaining soundness and completeness of incremental synthesis.

6.5. Specification Simplification

In this section, we study in which cases the given LTL specification φ for the whole system can be simplified for the individual components that we derived with either the semantic or the syntactic decomposition algorithm introduced in the previous two sections. In particular, we identify conjuncts of φ that are not relevant for the component c_i under consideration to reduce the size of the specification for c_i's synthesis task.

In general, omitting conjuncts of φ for a component c_i is not sound since the missing conjuncts may invalidate admissibility of the specification [DF14]: consider, for instance, the LTL formula $\varphi = \square(i \leftrightarrow \bigcirc o_1) \land \square(o_1 \leftrightarrow \bigcirc o_2)$, where i is an input variable, and both o_1 and o_2 are output variables, i.e., we have $I = \{i\}$ and $O = \{o_1, o_2\}$. Individually, both conjuncts $\square(i \leftrightarrow \bigcirc o_1)$ and $\square(o_2 \leftrightarrow \bigcirc o_1)$ are admissible for the full system. For conjunct $\square(i \leftrightarrow \bigcirc o_1)$, a strategy that sets o_1 to *true* if, and only if, i is *true* in the previous time step is dominant and even winning. For conjunct $\square(o_2 \leftrightarrow \bigcirc o_1)$, a strategy that, for instance, never sets o_1 or o_2 to *true* is winning.

However, the full specification φ is not admissible as a strategy would need to predict the valuation of i in order to set the valuation of o_2 correctly [DF14]. However, non-admissible conjuncts of an LTL formula cannot become admissible by leaving out conjuncts that do not refer to output variables of the system:

Theorem 6.9 ([DF14])**.** *Let φ be an LTL formula over atomic propositions V with $\varphi = \psi \wedge \psi'$, where ψ is an LTL formula over atomic propositions V and where ψ' is an LTL formula over atomic propositions $V \setminus O$. If ψ is admissible, then φ is admissible as well.*

Intuitively, this monotonicity property holds since ψ' does not range over output variables of the system and thus its truth value is solely determined by input variables of the system. In particular, a strategy for the system thus cannot influence the satisfaction of ψ'. A strategy s, however, can only be dominant for ψ but not dominant for φ if there is some input sequence on which the strategy satisfies ψ and violates φ, while there exists an alternative strategy t that satisfies φ on this input sequence. By construction of φ, then s needs to violate ψ' on this input sequence, while t does not, contradicting that a strategy for the system cannot influence the satisfaction of ψ'. For more details on this monotonicity property and the proof of its correctness, we refer to [DF14]. In the following, we extend this simplification result to individual components of the system:

Lemma 6.7. *Let c_i be a component with output variables O_i. Let φ be an LTL formula over atomic propositions V with $\varphi = \psi \wedge \psi'$, where ψ is an LTL formula over atomic propositions V and where ψ' is an LTL formula over atomic propositions $V \setminus O_i$. Let s_i be a strategy for c_i. If s_i is dominant for ψ and c_i, then s_i is dominant for φ and c_i as well.*

Proof. Suppose that s_i is not dominant for φ. Then, there exists an input sequence $\gamma \in (2^{I_i})^\omega$ and an alternative strategy t_i for c_i such that $comp(s_i, \gamma) \not\models \varphi$ holds, while we have $comp(t_i, \gamma) \models \varphi$. Since ψ' is an LTL formula over atomic propositions $V \setminus O_i$, it only refers to variables outside of the control of c_i. Hence, its truth value is solely determined by the valuations of the input variables of c_i. Therefore, in particular, we have $comp(s_i, \gamma) \models \psi'$ if, and only if, $comp(t_i, \gamma) \models \psi'$ holds. Hence, since $comp(t_i, \gamma) \models \varphi$ holds by assumptions and thus $comp(t_i, \gamma) \models \psi'$ follows with the semantics of conjunction, we have $comp(s_i, \gamma) \models \psi'$ as well. Therefore, $comp(s_i, \gamma) \not\models \psi$ follows from the assumption that $comp(s_i, \gamma) \not\models \varphi$ holds. But then we have $comp(s_i, \gamma) \not\models \psi$, while $comp(t_i, \gamma) \models \psi$ follows from the assumption that $comp(t_i, \gamma) \models \varphi$ holds and the definition of φ, contradicting that s_i is dominant for ψ and c_i. □

For components that do not depend syntactically on any variables outside their control, i.e., neither on input variables of the system, nor on output variables of other components, omitting such conjuncts is even complete. That is, whenever the full specification is admissible, so is the one obtained from omitting conjuncts. This relies on the fact that the conjunct of an LTL formula can never have *more* dependencies than the full LTL formula when deriving dependencies with the syntactic decomposition approach, i.e., with computing syntactic dependencies and then deriving further edges in the syntactic dependency graph. Note here that for completeness it is not required that the omitted specification does not range over output variables of the considered component:

Lemma 6.8. *Let c_i be a component with output variables O_i. Let φ be an LTL formula over atomic propositions V with $\varphi = \psi \wedge \psi'$. Let $\mathbb{C}(\mathcal{D}_\varphi^{xsyn}) = (\mathcal{V}^{syn}, \mathbb{C}(\mathcal{E}^{xsyn}))$ be the transitive closure of the extended syntactic dependency graph of φ. If, for all $u \in O_i$ and all $v \in V \setminus O_i$, we have $(u, v) \notin \mathbb{C}(\mathcal{E}^{xsyn})$, then ψ is admissible for c_i.*

Proof. Let $(u, v) \notin \mathbb{C}(\mathcal{E}^{xsyn})$ hold for all $u \in O_i$ and all $v \in V \setminus O_i$. Then, since, by construction of the dependency graph in the syntactic decomposition algorithm, dependencies are never removed, neither by adding conjuncts nor in one of the later steps, i.e., transitive closure or deriving dependencies, it follows immediately from the construction of φ that there do not exist edges (u, v) for any $u \in O_i$ and any $v \in V \setminus O_i$ in the transitive closure of the extended syntactic dependency graph of ψ. Then, it follows from Theorem 6.7, ψ does not induce any semantic dependencies from an output $u \in O_i$ to a variable $v \in V \setminus O_i$. Therefore, ψ is admissible for c_i by Theorem 6.3. □

Hence, together with Lemma 6.7, it follows immediately that for components that do not have any edges from one of their output variables to any variable outside their control in the transitive closure of the extended syntactic dependency graph of the original LTL specification φ, conjuncts of φ that do not contain any output variable of the considered component can be omitted without losing admissibility. Furthermore, it is guaranteed that every dominant strategy for the simplified specification is dominant for the original one as well.

Corollary 6.3. *Let c_i be a component with output variables O_i. Let φ be an LTL formula over atomic propositions V with $\varphi = \psi \wedge \psi'$. Let $\mathbb{C}(\mathcal{D}_\varphi^{xsyn}) = (\mathcal{V}^{syn}, \mathbb{C}(\mathcal{E}^{xsyn}))$ be the transitive closure of the extended syntactic dependency graph of φ, where ψ is an LTL formula over atomic propositions V and where ψ' is an LTL formula over atomic propositions $V \setminus O_i$. If, for all $u \in O_i$ and all $v \in V \setminus O_i$, we have $(u, v) \notin \mathbb{C}(\mathcal{E}^{xsyn})$, then ψ is admissible for c_i and every dominant strategy for ψ and c_i is also dominant for φ and c_i.*

For the autonomous car from the running example from Section 6.1, for instance, we identified a component describing the gearing unit, i.e., a component controlling the variables g_1 and g_2 with the syntactic decomposition algorithm. This component does not contain variables that depend semantically on any variable outside the control of the component (see Example 6.7). Furthermore, none of the conjuncts of the conjunct φ_{acc} of the car's specification φ_{car} contains variable g_1 or variable g_2. Therefore, it follows from Corollary 6.3 that it suffices to use φ_{gear} as the specification for the synthesis task of the component describing the gearing unit instead of the full specification φ_{car} with $\varphi_{car} = \varphi_{acc} \wedge \varphi_{gear}$.

Moreover, it follows immediately from the results of the previous chapter – in particular Sections 5.2 and 5.3 – that if a specification with two conjuncts that induces two components such that the conjuncts do not range over the outputs of the other component is realizable, then there are winning strategies for the components for the respective conjuncts:

Theorem 6.10. *Let $\varphi = \psi \wedge \psi'$ be an LTL formula over atomic propositions V that induces two components c_i and c_j with output variables O_i and O_j, respectively, and such that ψ and ψ' are LTL formula over atomic propositions $V \setminus O_j$ and $V \setminus O_i$, respectively. If φ is realizable, then there are winning strategies s_i and s_j for c_i and c_j for ψ and ψ', respectively, such that $s_i \mid\mid s_j \models \varphi$.*

Proof. Let φ be realizable. By construction of the specification, $\mathcal{L}(\psi) \mid\mid \mathcal{L}(\psi') = \mathcal{L}(\varphi)$ follows with Lemma 5.7. Furthermore, by construction of the decomposition algorithms, we have $O = O_1 \cup O_j$ and $O_i \cap O_j = \emptyset$. Thus, $V \setminus O_j = I \cup O_i$ and $V \setminus O_i = I \cup O_j$ holds. Therefore, since we have $prop(\psi) \subseteq V \setminus O_j$ and $prop(\psi') \subseteq V \setminus O_i$ by constructions, $prop(\psi) \cap prop(\psi') \subseteq I$ follows. Hence, by Lemma 5.8, $\mathcal{L}(\psi)$ and $\mathcal{L}(\psi')$ are non-contradictory and therefore $\mathcal{L}(\psi)$ and $\mathcal{L}(\psi')$ are independent sublanguages according to Definition 5.6. Thus, since φ is realizable by assumption, it follows with Lemma 5.6, that both ψ and ψ' are realizable for c_i and c_j as well. Let s_i and s_j be strategies for c_i and c_j, respectively, such that $s_i \models \psi$ and $s_j \models \psi'$ holds. Then, since we represent strategies by deterministic and complete finite-state transducers, $s_i \mid\mid s_j \models \varphi$ holds by Lemma 5.4. □

Furthermore, in incremental synthesis, the strategies of components with a lower rank in the synthesis order are provided to the component c_i under consideration. Hence, if these strategies are winning for a conjunct of the specification, then the conjunct may be eliminated from the specification for c_i since its satisfaction is already guaranteed:

Theorem 6.11. *Let φ be an LTL formula over atomic propositions V with $\varphi = \psi \wedge \psi'$. Let $\mathbb{D}$ be a decomposition of (I, O) and let $c_i \in \mathbb{D}$ be a component. Let c be the parallel composition of the components $c_j \in \mathbb{D}$ with $c_j <_{syn} c_i$ and let s be the parallel composition of their synthesized strategies. If s is winning for ψ', then there is a strategy s_i for c_i such that $s \mid\mid s_i$ is dominant for ψ and $c \mid\mid c_i$ if, and only if there is a strategy s_i for c_i such that $s \mid\mid s_i$ is dominant for φ and $c \mid\mid c_i$.*

Proof. Let O_{prev} denote the outputs of c, i.e., let $O_{prev} = \bigcup_{c_j \in C} O_j$, where the set of all components of $\mathbb{D}$ with a lower rank in the synthesis order than c_i is denoted with $C = \{c_j \in \mathbb{D} \mid c_j <_{syn} c_i\}$. Let s be winning for ψ'. Then, we have $comp(s, \gamma) \models \psi'$ for every $\gamma \in (2^{V \setminus O_{prev}})^\omega$. Thus, in particular, $comp(s \mid\mid s_i, \gamma) \models \psi'$ holds for every strategy s_i for c_i and every $\gamma \in (2^{V \setminus (O_{prev} \cup O_i)})^\omega$. Hence, by construction of φ, it holds that for every strategy s_i for c_i and every $\gamma \in (2^{V \setminus (O_{prev} \cup O_i)})^\omega$, we have $comp(s \mid\mid s_i, \gamma) \models \varphi$ if, and only if $comp(s \mid\mid s_i, \gamma) \models \psi$ holds.

First, let there exist a strategy s_i for c_i such that $s \mid\mid s_i$ is dominant for ψ and $c \mid\mid c_i$. Suppose that s_i is not dominant for φ and $c \mid\mid c_i$. Then, there exists an input sequence $\gamma \in (2^{V \setminus (O_{prev} \cup O_i)})^\omega$ and an alternative strategy t for $c \mid\mid c_i$ such that $comp(s \mid\mid s_i, \gamma) \not\models \varphi$ holds, while we have $comp(t_i, \gamma) \models \varphi$. As shown above, we then have $comp(s \mid\mid s_i, \gamma) \not\models \psi$ as well. However, by construction of φ and by the semantics of conjunction, we also have $comp(t_i, \gamma) \models \psi$, contradicting the assumption that $s \mid\mid s_i$ is dominant for ψ and $c \mid\mid c_i$.

Second, let there exist a strategy s_i for c_i such that $s \mid\mid s_i$ is dominant for φ and $c \mid\mid c_i$. Suppose that s_i is not dominant for ψ and $c \mid\mid c_i$. Then, there exists an input sequence $\gamma \in (2^{V \setminus (O_{prev} \cup O_i)})^\omega$ and an alternative strategy t for $c \mid\mid c_i$ such that $comp(s \mid\mid s_i, \gamma) \not\models \psi$ holds, while we have $comp(t_i, \gamma) \models \psi$. As shown above, we then have $comp(t_i, \gamma) \models \varphi$ as well. However, by construction of φ and by the semantics of conjunction, we also have $comp(s \mid\mid s_i, \gamma) \not\models \varphi$, contradicting the assumption that $s \mid\mid s_i$ is dominant for φ and $c \mid\mid c_i$. □

Hence, we can simplify the specifications for the individual synthesis tasks in incremental synthesis under certain conditions outlined in this section. In particular, omitting conjuncts that do not range over output variables of the considered component is always sound. That is, as long as the omitted conjuncts do not contain output variables of the considered component, a strategy synthesized for the simplified specification is guaranteed to be dominant for the full specification as well. Therefore, even if none of the conditions needed for completeness of specification simplification is satisfied, it is valid to *try* to omit such conjuncts. If the synthesis task for the simplified specification succeeds, then the strategy is guaranteed to be correct for the full specification as well. Even though we cannot conclude non-admissibility of the full specification if the synthesis task for the simplified one fails, we can iteratively add conjuncts until either the synthesis task succeeds or the full specification is reached.

For instance, omitting conjuncts for a component c_i that do not contain any output variables of c_i often succeeds even if for some output variable of c_i, there exists an edge to some input variable of the whole system in the transitive closure of the extended dependency graph, as long as no edges to output variables of the system outside of c_i's control exist. Although admissibility of the simplified formula is then not guaranteed, there exist dominant strategies for the specifications obtained from this kind of simplification in all our benchmarks (see Section 6.6), where it was applicable. This is partly due to the observation that dependencies to input variables that do not prevent admissibility of the full specification, also do not prevent admissibility of conjuncts in many cases.

6.6. Experimental Evaluation

We have implemented a prototype of the incremental synthesis algorithm. It expects an LTL specification as well as a decomposition of the system and a synthesis order as input. Our prototype extends the bounded synthesis tool BoSy [FFT17] to the synthesis of dominant strategies. Particularly, it utilizes Steiger's rewriting-based approach [Ste13] as described in Section 2.8.2. Furthermore, it converts the synthesized strategy from the Aiger circuit produced by our extension of BoSy into an equivalent LTL formula that is added to the specification of the next component. This encodes the synthesis task in line 7 of Algorithm 6.1 into the framework of monolithic synthesis. The implementation order has been realized in the monolithic synthesis framework by encoding a corresponding delay into the LTL specification.

We compare our prototype to the original version of BoSy on four scalable benchmarks. The results are presented in Table 6.1. We used a machine with a 3.1 GHz Dual-Core Intel Core i5 processor and 16 GB of RAM and a timeout of 60 minutes. The first two benchmarks, the *n-ary latch* and the *generalized buffer*, stem from the annual reactive synthesis competition SyntComp [BEJ14, JBB+17b, JBB+15, JBB+16, JB16, JBB+17a, JBC+19, JPA+22]. The latch is parameterized in the number of bits. The generalized buffer is parameterized in the number of receivers. For a more detailed description of these benchmarks, we refer to [JBC+19]. For the n-ary latch, both the semantic and the syntactic component selection algorithms identify n separate components, one for each bit of the latch. For the generalized buffer, both decomposition techniques identify two components, one for the communication with the senders and one for the communication with the receivers. After simplifying the specifications by omitting conjuncts that do not contain output variables of the considered component, we are able to synthesize separate winning strategies for the components for both benchmarks, making use

Table 6.1.: Experimental results on four scalable benchmarks. Reported is the parameter and the synthesis time in seconds. The timeout is 60 minutes.

Benchmark	Parameter	BoSy	Incremental Synthesis
n-ary Latch	2	**2.61**	4.76
	3	**3.66**	6.58
	4	11.55	**8.74**
	5	TO	**10.98**
	6	TO	**12.52**
	...	...	...
	1104	TO	**3599.04**
Generalized Buffer	1	37.04	**5.08**
	2	TO	**6.21**
	3	TO	**66.03**
Sensors	2	**1.99**	6.08
	3	**2.31**	8.79
	4	**6.99**	11.73
	5	92.79	**16.99**
	6	TO	**43.50**
	7	TO	**2293.85**
Robot Fleet	2	**2.49**	6.25
	3	TO	**10.51**
	4	TO	**269.09**

of Theorem 6.10. Note that both the specification for the latch and the one for the buffer induce dependencies to input variables. Thus, omitting the conjuncts is not complete in general. For these specifications, however, it succeeds. The incremental synthesis approach clearly outperforms BoSy's classical bounded synthesis approach for the generalized buffer in all cases. Particularly noteworthy is parameter $n = 2$, for which BoSy does not terminate within one hour, while incremental synthesis synthesizes a solution in less than seven seconds. For the n-ary latch, the advantage of incremental synthesis becomes clear from parameter $n = 4$ on. Encouragingly, incremental synthesis succeeds up to parameter $n = 1104$, while BoSy already fails in synthesizing a solution for parameter $n = 5$ within one hour.

In addition to the n-ary latch and the generalized buffer, we consider a benchmark describing n sensors and a managing unit. The latter requests and collects sensor data. The managing unit may receive the direction to check the data of all sensors, denoted with the input variable *check*. It may request the i-th sensor data using the output variable $request_i$. The i-th sensor may send data to the managing unit using the output variable $data_i$. Hence, the system consists of a single input variable, namely *check*, and $2n$ output variables, where the n variables $request_i$ are controlled by the managing unit, and the n variables $data_i$ are controlled by the corresponding sensors. Whenever the system receives the direction to check the data of all sensors, $request_i$ must be set to *true* eventually for all sensors. However, mutual exclusion of the requests needs to be ensured as the managing unit utilizes a single wire for communicating from the managing unit to all sensors. Once sensor i receives the direction to check its data via $request_i$, it must send its data eventually, utilizing output variable $data_i$. Lastly, data can only be sent one step after a request has been received. This ensures that data is not sent without request and, together with the mutual exclusion of the requests, that mutual exclusion of data is guaranteed as well, accounting for the existence of only a single communication wire from all sensors to the managing unit. We formalize these requirements in the following LTL specification φ for a system with n sensors and their managing unit:

$$\varphi = \bigwedge_{1\leq i\leq n} \bigwedge_{\substack{1\leq j\leq n \\ i\neq j}} \square(request_i \rightarrow \neg request_j) \ \wedge \bigwedge_{1\leq i\leq n} \square(check \rightarrow \diamond\, request_i)$$
$$\wedge \bigwedge_{1\leq i\leq n} \square(request_i \rightarrow \diamond\, data_i) \ \wedge \bigwedge_{1\leq i\leq n} \square((\bigcirc\, data_i) \rightarrow request_i),$$

For the sensor specification, the semantic decomposition algorithm identifies n separate components for the sensors as well as a component for the managing unit that depends on the other components. For this decomposition, the incremental synthesis approach outperforms BoSy from parameter $n = 5$ on, i.e., if we consider five or more sensors. Most notably, for parameter $n = 6$, incremental synthesis derives a solution in less than 45 seconds while BoSy does not terminate within one hour. The syntactic component selection technique, however, does not identify the separability of the sensors from the managing unit due to the overapproximation in the transitivity and derivation rules.

Lastly, we consider a benchmark describing a fleet of n robots that must not collide with another robot crossing their way. Upon receiving the starting signal, denoted with the input variable *ready*, the additional robot starts moving. The i-th robot in the fleet may stop, move left, or move right, denoted with the output variables $stop_i$, $left_i$, or $right_i$, respectively. The additional robot outside the fleet may notify the i-th robot of the fleet, denoted with the output variable $robot\_ahead_i$, that a collision is ahead if the fleet robot does not change its course. We used the LTL specification $\varphi = (\square\diamond\neg ready) \rightarrow \psi$ for the robot fleet benchmark with n robots in the fleet and one additional robot, where

$$\psi = \bigwedge_{1\leq i\leq n} \neg stop_i \ \wedge \bigwedge_{1\leq i\leq n} \square\diamond\neg stop_i \ \wedge \bigwedge_{1\leq i\leq n} \square\neg(left_i \wedge right_i)$$
$$\wedge \bigwedge_{1\leq i\leq n} \square(ready \rightarrow \diamond\, robot\_ahead_i)$$
$$\wedge \bigwedge_{1\leq i\leq n} \square(robot\_ahead_i \rightarrow \bigcirc(left_i \vee right_i \vee stop_i)).$$

The first two conjuncts ensure that the fleet robots start moving in the very first step and that they move infinitely often. Mutual exclusion between moving left and right is established by the third conjunct. Upon receiving the starting signal, a collision between the additional robot and each fleet robot is ahead eventually, formalized by the fourth conjunct. This models that the additional robot starts moving and crosses the way of each fleet robot. The fifth conjunct then ensures that the fleet robots react by either moving left, moving right, or stopping if a collision with the additional robot is ahead.

Both the semantic and the syntactic techniques identify n separate components for the robots in the fleet as well as a component for the additional robot. The latter component depends on the former ones. Incremental synthesis clearly outperforms BoSy from parameter $n = 3$ on. Most notably, it still synthesizes a solution in less than 10 seconds for parameter $n = 4$, while BoSy does not terminate within one hour anymore.

All in all, the results of our experimental evaluation clearly demonstrate the advantage of incremental synthesis over classical monolithic synthesis approaches. Our prototype significantly outperforms the bounded synthesis approach implemented in BoSy when the specifications grow. Incremental synthesis is particularly beneficial for parameterized specifications, in which increasing the parameter results in an increased number of components.

6.7. Summary

We have presented an incremental synthesis algorithm that reduces the complexity of synthesis by decomposing large monolithic systems into several components. Unlike classical compositional approaches, which aim at synthesizing strategies for the components completely independently, our algorithm proceeds in an incremental fashion. In addition to the decomposition of the system into components, it computes the order in which strategies for the components should be synthesized. The synthesis task of a component can then rely on the components with a lower rank in the synthesis order do not deviate from their strategies, which have been synthesized previously. This allows for applying incremental synthesis also to systems with more interconnected specifications for which the fully compositional synthesis approach introduced in the previous chapter fails.

We have introduced two algorithms to select the components, one based on a semantic dependency analysis of the output variables of the system and one based on a syntactic analysis of the specification. Both decomposition techniques further define the order in which strategies for the components are synthesized. Soundness and completeness of incremental synthesis are guaranteed for the decomposition and synthesis order computed with both the semantic and the syntactic decomposition algorithm. Furthermore, we have presented rules for simplifying the specifications for the individual components while maintaining correctness of incremental synthesis and, in particular, success of the individual synthesis tasks. We have implemented a prototype of the incremental synthesis algorithm and compared it to the monolithic bounded synthesis tool BoSy. Our experiments clearly demonstrate the advantage of incremental synthesis over classical synthesis for larger systems as our prototype significantly outperforms BoSy when the specifications grow.

Chapter 7

Conclusions

In this thesis, we have developed automated techniques for the compositional synthesis of both distributed and monolithic reactive systems. Our algorithms automate the extensive manual interventions that have so far been required from the developer for applying compositional concepts to reactive synthesis.

Summary. For distributed systems, pinpointing what system processes need to know about other processes and their behavior in order to be able to satisfy the specification is fundamental. So far, identifying this knowledge and incorporating it into the synthesis tasks of the individual processes has been primarily a manual task. We developed two approaches for automatically deriving the required assumptions about other processes. For the first algorithm, we introduced delay-dominance, a new requirement for strategies. Delay-dominance is a best-effort notion for strategies that, in contrast to winning strategies, permits the violation of the specification in certain situations. Intuitively, delay-dominance allows every process to implicitly assume that the other processes will not maliciously violate the shared goal. We proved that it overcomes the shortcomings of existing variants of dominance and is thus a suitable notion for compositional synthesis. We presented a synthesis algorithm for delay-dominant strategies for monolithic systems and extended it to a compositional synthesis approach for distributed systems. The second algorithm relies on explicit assumptions on the concrete behavior of other processes. It is thus suitable also for distributed systems with more complex inter-process dependencies. Our approach automatically constructs valid assume-guarantee contracts between the system processes, which provide essential information for the synthesis tasks of the individual processes. The behavioral guarantees additionally enable modularity of the synthesized system, allowing for safely exchanging process strategies as long as they still meet the contract.

For monolithic systems, one of the key challenges is to identify a suitable decomposition of the single process of the system into several components that then constitute synthesis subtasks. Until now, such independent components had to be recognized manually by the developer. Therefore, we introduced two approaches for automatically identifying decompositions of monolithic systems that ensure both soundness and completeness of their respective synthesis algorithms. The first approach identifies completely independent components, i.e., components for which separate synthesis tasks succeed without the need for making assumptions about the behavior of other components. Hence, an individual synthesis task is a classical monolithic synthesis problem. Therefore, our decomposition algorithm can be seen as a preprocessing technique for a wide range of existing synthesis approaches and tools. Furthermore, we demonstrated the applicability of our approach to specifications of smart contracts, indicating that our algorithms are particularly beneficial for specific domains. The second approach allows for finding more fine-grained decompositions of a monolithic system than the first one. Similar to our first approach for compositional distributed systems, it seeks dominant strategies rather than winning ones, thus allowing for implicitly assuming that the other components will not maliciously violate the shared goal. Furthermore, we employ an incremental rather than a fully compositional synthesis algorithm. In incremental synthesis, we synthesize a dominant strategy for a component under the assumption that previously synthesized components do not deviate from their synthesized strategies. Hence, our second approach combines implicit assumptions from dominant strategies with explicit assumptions on previously synthesized components. The decomposition algorithm takes both the dominance of the desired strategies and the incremental nature of the employed synthesis algorithm into account.

Our experimental evaluation shows that our compositional algorithms for both distributed and monolithic synthesis automate the manual efforts that have previously been required for compositional synthesis. Hence, they constitute fully automated compositional synthesis algorithms. Furthermore, our approaches significantly outperform classical, non-compositional synthesis algorithms on scalable benchmarks.

Conclusions and Future Directions. The algorithms introduced in this thesis are a fundamental step toward the compositional and, thus, more scalable synthesis of distributed and monolithic reactive systems. They lay theoretical foundations in the areas of best-effort strategies and assumption generation, as well as in sound and complete system decomposition for different types of synthesis algorithms. Our prototype implementations demonstrate the practical potential of our algorithms. In particular, the fact that our decomposition algorithm for entirely compositional synthesis of monolithic systems has been integrated into the most recent release of state-of-the-art reactive synthesis tool ltlsynt [MC18, RSDP22] as a preprocessing technique showcases the relevance of our work.

However, this thesis clearly does not complete the massive undertaking of compositional reactive synthesis: none of the approaches introduced in this thesis is *the one* compositional synthesis algorithm. Instead, all of them have their advantages and disadvantages and excel for different system types and specification classes. Automatically constructing assume-guarantee contracts for compositional distributed synthesis, for instance, is particularly beneficial for systems with complex inter-process dependencies but small interfaces between the processes. Utilizing delay-dominance for compositional synthesis of distributed systems, in contrast, is fitting for large systems with many but not too complex inter-process dependencies. For monolithic systems, decomposing the system into independent components for which winning strategies can be synthesized separately only has an advantage over classical monolithic synthesis algorithms if the system's specification contains completely independent parts. Incremental synthesis with dominant strategies, in contrast, allows for finding decompositions of specifications that contain dependencies. However, both the decomposition and synthesis algorithms are more complex than in the purely compositional monolithic synthesis approach. Hence, which algorithm to choose highly depends on the considered system and its specification.

Consequently, identifying classes of systems and specifications for which the algorithms introduced in this thesis perform particularly well is a logical next step. For instance, one could think of identifying specific system architectures or LTL fragments that allow for efficient usage of one of our compositional synthesis algorithms. Ideally, this can result in a guideline for developers on how they should design their system and how they should specify the system requirements in order to allow for successful compositional synthesis. Furthermore, restricting the specification to specific fragments, e.g., safety specifications or the GR(1) fragment [PPS06, BJP+12, KP10] of LTL, has proven to result in more efficient non-compositional synthesis procedures. Following this successful idea, the restriction of compositional synthesis to particular system types and specification classes might allow for more targeted algorithms, thus further increasing the efficiency and scalability.

In a similar direction, recognizing domains for which compositional synthesis approaches are beneficial is a crucial step toward the applicability of synthesis and particularly compositional synthesis in practice. Doubtlessly, there will always be large and highly interconnected systems for which compositional synthesis approaches are not applicable in the sense that they do not have an advantage over classical synthesis methods. However, there are, most likely, domains for which compositional synthesis excels. For instance, we observed in our experimental evaluation of the decomposition algorithm for monolithic systems that identifies completely independent components that it performs particularly well on those benchmarks that stem from the Syntroids [GHKF19] case study, the fully synthesized realization of an arcade game on an FPGA. This indicates that hardware components might be a promising candidate for a suitable domain for compositional synthesis. Similarly, as illustrated in Section 5.7, certain classes of smart contracts frequently exhibit a natural decomposition into their temporal control flow and the required actions for particular function calls.

An intriguing open research question is the existence of a semantic notion of dominance-style strategies that ensures compositionality also for liveness properties. Both bounded dominance [DF14] and delay-dominance, presented in this thesis, are syntactic notions as they are defined on the structure of an ω-automaton representing the specification. This has the disadvantage that, for an LTL specification, there is not necessarily a unique answer to the question of whether or not a strategy is bounded dominant or delay-dominant, respectively, as this depends on the considered automaton. A semantic notion overcomes this weakness and has thus the potential to induce a more elegant compositional synthesis approach.

Furthermore, research on the practical synthesis of remorsefree dominant strategies has, so far, been limited. While algorithms for synthesizing such strategies exist [DF14, Ste13], they rely on altering the automaton used for synthesis to recognize remorsefree dominant rather than winning strategies. The automaton construction is easy to implement as it heavily utilizes algorithms for constructing ω-automata from LTL formulas, for which well-engineered tools such as Spot [DLF+16, DRC+22] exist. However, it produces large intermediate automata during construction, which can negatively affect the construction's efficiency. Avoiding the intermediate automata by immediately incorporating the concept of dominance into the automaton construction instead of first making a detour over automata for winning strategies and then obtaining dominance by universal projection might boost the performance of synthesis of dominant strategies, which has an immediate impact on the performance of our incremental synthesis algorithm for monolithic systems.

In summary, while this thesis lays the foundations for fully automated compositional synthesis algorithms for distributed and monolithic systems, this line of research is far from complete. Compositional synthesis is a mammoth task, and putting it into practice will require plenty of further research. Exploring the research directions mentioned above, as well as advancing the underlying tools for reactive synthesis and automata translations and manipulations, can significantly increase the performance and the applicability of automatic compositional methods for distributed and monolithic synthesis.

Bibliography

[ABB16] Lacramioara Astefanoaei, Saddek Bensalem, and Marius Bozga. A compositional approach to the verification of hybrid systems. In *Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*, Vol. 9660 of *Lecture Notes in Computer Science*, pp. 88–103. Springer, 2016. doi: 10.1007/978-3-319-30734-3_8.

[ACG^{+}08] Suzana Andova, Cas Cremers, Kristian Gjøsteen, Sjouke Mauw, Stig Fr. Mjølsnes, and Sasa Radomirovic. A framework for compositional verification of security protocols. *Inf. Comput.*, 206(2-4):425–459, 2008. doi: 10.1016/j.ic.2007.07.002.

[AGL^{+}20] Benjamin Aminof, Giuseppe De Giacomo, Alessio Lomuscio, Aniello Murano, and Sasha Rubin. Synthesizing strategies under expected and exceptional environment behaviors. In *29th International Joint Conference on Artificial Intelligence, IJCAI 2020, Proceedings*, pp. 1674–1680, 2020. doi: 10.24963/ijcai.2020/232.

[AGMR18] Benjamin Aminof, Giuseppe De Giacomo, Aniello Murano, and Sasha Rubin. Synthesis under assumptions. In *Principles of Knowledge Representation and Reasoning - 16th International Conference, KR 2018, Proceedings*, pp. 615–616. AAAI Press, 2018.

[AGR21] Benjamin Aminof, Giuseppe De Giacomo, and Sasha Rubin. Best-effort synthesis: Doing your best is not harder than giving up. In *13th International Joint Conference on Artificial Intelligence, IJCAI 2021, Proceedings*, pp. 1766–1772, 2021. doi: 10.24963/ijcai.2021/243.

[AK20] Shaull Almagor and Orna Kupferman. Good-enough synthesis. In *Computer Aided Verification - 32nd International Conference, CAV 2020, Proceedings*, Vol. 12225 of *Lecture Notes in Computer Science*, pp. 541–563. Springer, 2020. doi: 10.1007/978-3-030-53291-8_28.

[AKRV17] Shaull Almagor, Orna Kupferman, Jan Oliver Ringert, and Yaron Velner. Quantitative assume guarantee synthesis. In *Computer Aided Verification - 29th International Conference, CAV 2017, Proceedings*, Vol. 10427 of *Lecture Notes in Computer Science*, pp. 353–374. Springer, 2017. doi: 10.1007/978-3-319-63390-9_19.

[AMT13] Rajeev Alur, Salar Moarref, and Ufuk Topcu. Counter-strategy guided refinement of GR(1) temporal logic specifications. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Proceedings*, pp. 26–33. IEEE, 2013. DOI: 10.1109/FMCAD.2013.6679387.

[AMT15] Rajeev Alur, Salar Moarref, and Ufuk Topcu. Pattern-based refinement of assume-guarantee specifications in reactive synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Proceedings*, Vol. 9035 of *Lecture Notes in Computer Science*, pp. 501–516. Springer, 2015. DOI: 10.1007/978-3-662-46681-0_49.

[AMT18] Rajeev Alur, Salar Moarref, and Ufuk Topcu. Compositional and symbolic synthesis of reactive controllers for multi-agent systems. *Inf. Comput.*, 261:616–633, 2018. DOI: 10.1016/j.ic.2018.02.021.

[AS87] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Comput.*, 2(3):117–126, 1987. DOI: 10.1007/BF01782772.

[Bau17] Jan E. Baumeister. Encodings of bounded synthesis for distributed systems. Bachelor's thesis, Saarland University, 2017.

[BBF$^+$12] Aaron Bohy, Véronique Bruyère, Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. Acacia+, a tool for LTL synthesis. In *Computer Aided Verification - 24th International Conference, CAV 2012, Proceedings*, Vol. 7358 of *Lecture Notes in Computer Science*, pp. 652–657. Springer, 2012. DOI: 10.1007/978-3-642-31424-7_45.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Tools and Algorithms for Construction and Analysis of Systems - 5th International Conference, TACAS 1999, Proceedings*, Vol. 1579 of *Lecture Notes in Computer Science*, pp. 193–207. Springer, 1999. DOI: 10.1007/3-540-49059-0_14.

[BCJ18] Roderick Bloem, Krishnendu Chatterjee, and Barbara Jobstmann. Graph games and reactive synthesis. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pp. 921–962. Springer, 2018. DOI: 10.1007/978-3-319-10575-8_27.

[BCJK15] Roderick Bloem, Krishnendu Chatterjee, Swen Jacobs, and Robert Könighofer. Assume-guarantee synthesis for concurrent reactive programs with partial information. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Proceedings*, Vol. 9035 of *Lecture Notes in Computer Science*, pp. 517–532. Springer, 2015. DOI: 10.1007/978-3-662-46681-0_50.

[BDG$^+$04] Guillaume P. Brat, Doron Drusinsky, Dimitra Giannakopoulou, Allen Goldberg, Klaus Havelund, Michael R. Lowry, Corina S. Pasareanu, Arnaud Venet, Willem Visser, and Richard Washington. Experimental evaluation of verification and validation tools on martian rover software. *Formal Methods Syst. Des.*, 25(2-3):167–198, 2004. DOI: 10.1023/B:FORM.0000040027.28662.a4.

[BEJ14] Roderick Bloem, Rüdiger Ehlers, and Swen Jacobs. The synthesis competition. `http://www.syntcomp.org`, 2014. Accessed: 2023-01-16.

[BEJK14] Roderick Bloem, Rüdiger Ehlers, Swen Jacobs, and Robert Könighofer. How to handle assumptions in synthesis. In *3rd Workshop on Synthesis, SYNT 2014, Proceedings*, Vol. 157 of *EPTCS*, pp. 34–50, 2014. DOI: 10.4204/EPTCS.157.7.

[Ber07] Dietmar Berwanger. Admissibility in infinite games. In *24th Annual Symposium on Theoretical Aspects of Computer Science, STACS 2007, Proceedings*, Vol. 4393 of *Lecture Notes in Computer Science*, pp. 188–199. Springer, 2007. DOI: 10.1007/978-3-540-70918-3_17.

[BFH19] Raven Beutner, Bernd Finkbeiner, and Jesko Hecking-Harbusch. Translating asynchronous games for distributed synthesis. In *30th International Conference on Concurrency Theory, CONCUR 2019, Proceedings*, Vol. 140 of *LIPIcs*, pp. 26:1–26:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. DOI: 10.4230/LIPIcs.CONCUR.2019.26.

[BGJ+07] Roderick Bloem, Stefan J. Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. Automatic hardware synthesis from specifications: a case study. In *Design, Automation and Test in Europe Conference and Exposition, DATE 2007, Proceedings*, pp. 1188–1193. EDA Consortium, 2007. DOI: 10.1109/DATE.2007.364456.

[BGS+22] Suguman Bansal, Giuseppe De Giacomo, Antonio Di Stasio, Yong Li, Moshe Y. Vardi, and Shufang Zhu. Compositional safety LTL synthesis. In *Verified Software. Theories, Tools and Experiments - 14th International Conference, VSTTE 2022, Revised Selected Papers*, Vol. 13800 of *Lecture Notes in Computer Science*, pp. 1–19. Springer, 2022. DOI: 10.1007/978-3-031-25803-9_1.

[Bie21] Armin Biere. Bounded model checking. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, Vol. 336 of *Frontiers in Artificial Intelligence and Applications*, pp. 739–764. IOS Press, 2021. DOI: 10.3233/FAIA201002.

[BJP+12] Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive(1) designs. *J. Comput. Syst. Sci.*, 78(3):911–938, 2012. DOI: 10.1016/j.jcss.2011.08.007.

[BKK11] Christel Baier, Joachim Klein, and Sascha Klüppelholz. Modeling and verification of components and connectors. In *Formal Methods for Eternal Networked Software Systems - 11th International School on Formal Methods for the Design of Computer, Advanced Lectures*, Vol. 6659 of *Lecture Notes in Computer Science*, pp. 114–147. Springer, 2011. DOI: 10.1007/978-3-642-21455-4_4.

[BKRS12] Tomás Babiak, Mojmír Kretínský, Vojtech Rehák, and Jan Strejcek. LTL to büchi automata translation: Fast and more deterministic. In *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Proceedings*, Vol. 7214 of *Lecture Notes in Computer Science*, pp. 95–109. Springer, 2012. DOI: 10.1007/978-3-642-28756-5_8.

[BL69] J. Richard Büchi and Lawrence H. Landweber. Solving sequential conditions by finite-state strategies. *Trans. Am. Math. Soc.*, 138:295–311, 1969. DOI: 10.1007/978-1-4613-8928-6_29.

[BPRS17] Romain Brenguier, Arno Pauly, Jean-François Raskin, and Ocan Sankur. Admissibility in games with imperfect information (invited talk). In *28th International Conference on Concurrency Theory, CONCUR 2017, Proceedings*, Vol. 85 of *LIPIcs*, pp. 2:1–2:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. DOI: 10.4230/LIPIcs.CONCUR.2017.2.

[BR06] Samik Basu and C. R. Ramakrishnan. Compositional analysis for verification of parameterized systems. *Theor. Comput. Sci.*, 354(2):211–229, 2006. DOI: 10.1016/j.tcs.2005.11.016.

[BRS14] Romain Brenguier, Jean-François Raskin, and Mathieu Sassolas. The complexity of admissibility in omega-regular games. In *Joint Meeting of the 23rd EACSL Annual Conference on Computer Science Logic and the 29th Annual ACM/IEEE Symposium on Logic in Computer Science, CSL-LICS 2014, Proceedings*, pp. 23:1–23:10. ACM, 2014. DOI: 10.1145/2603088.2603143.

[BRS17] Romain Brenguier, Jean-François Raskin, and Ocan Sankur. Assume-admissible synthesis. *Acta Informatica*, 54(1):41–83, 2017. DOI: 10.1007/s00236-016-0273-2.

[CDS13] Chia Yuan Cho, Vijay D'Silva, and Dawn Song. BLITZ: compositional bounded model checking for real-world programs. In *28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Proceedings*, pp. 136–146. IEEE, 2013. DOI: 10.1109/ASE.2013.6693074.

[CFGR16] Rodica Condurache, Emmanuel Filiot, Raffaella Gentilini, and Jean-François Raskin. The complexity of rational synthesis. In *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, Proceedings*, Vol. 55 of *LIPIcs*, pp. 121:1–121:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. DOI: 10.4230/LIPIcs.ICALP.2016.121.

[CGP02] Satish Chandra, Patrice Godefroid, and Christopher Palm. Software model checking in practice: an industrial case study. In *24th International Conference on Software Engineering, ICSE 2002, Proceedings*, pp. 431–441. ACM, 2002. DOI: 10.1145/581339.581393.

[CGP03] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu. Learning assumptions for compositional verification. In *Tools and Algorithms for the Construction and Analysis of Systems - 9th International Conference, TACAS 2003, Proceedings*, Vol. 2619 of *Lecture Notes in Computer Science*, pp. 331–346. Springer, 2003. DOI: 10.1007/3-540-36577-X_24.

[CH07] Krishnendu Chatterjee and Thomas A. Henzinger. Assume-guarantee synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems - 13th International Conference, TACAS 2007, Proceedings*, Vol. 4424 of *Lecture Notes in Computer Science*, pp. 261–275. Springer, 2007. DOI: 10.1007/978-3-540-71209-1_21.

[CHJ06] Krishnendu Chatterjee, Thomas A. Henzinger, and Marcin Jurdzinski. Games with secure equilibria. *Theor. Comput. Sci.*, 365(1-2):67–82, 2006. DOI: 10.1016/j.tcs.2006.07.032.

[CHJ08] Krishnendu Chatterjee, Thomas A. Henzinger, and Barbara Jobstmann. Environment assumptions for synthesis. In *Concurrency Theory, 19th International Conference, CONCUR 2008, Proceedings*, Vol. 5201 of *Lecture Notes in Computer Science*, pp. 147–161. Springer, 2008. DOI: 10.1007/978-3-540-85361-9_14.

[Chu57] Alonzo Church. Applications of Recursive Arithmetic to the Problem of Circuit Synthesis. In *Summaries of the Summer Institute of Symbolic Logic*, pp. 3–50, 1957.

[CLM89] Edmund M. Clarke, David E. Long, and Kenneth L. McMillan. Compositional model checking. In *Fourth Annual Symposium on Logic in Computer Science, LICS 1989, Proceedings*, pp. 353–362. IEEE Computer Society, 1989. DOI: 10.1109/LICS.1989.39190.

[CLM91] Edmund M. Clarke, David E. Long, and Kenneth L. McMillan. A language for compositional specification and verification of finite state hardware controllers. *Proc. IEEE*, 79(9):1283–1292, 1991. DOI: 10.1109/5.97298.

[CMP92] Edward Y. Chang, Zohar Manna, and Amir Pnueli. Characterization of temporal property classes. In *Automata, Languages and Programming - 19th International Colloquium, ICALP 1992, Proceedings*, Vol. 623 of *Lecture Notes in Computer Science*, pp. 474–486. Springer, 1992. DOI: 10.1007/3-540-55719-9_97.

[CMP94] Edward Y. Chang, Zohar Manna, and Amir Pnueli. Compositional verification of real-time systems. In *Ninth Annual Symposium on Logic in Computer Science LICS 1994, Proceedings*, pp. 458–465. IEEE Computer Society, 1994. DOI: 10.1109/LICS.1994.316045.

[Cre04] Cas Cremers. Compositionality of security protocols: A research agenda. In *First International Workshop on Views on Designing Complex Architectures, VODCA@FOSAD 2004, Proceedings*, Vol. 142 of *Electronic Notes in Theoretical Computer Science*, pp. 99–110. Elsevier, 2004. DOI: 10.1016/j.entcs.2004.12.047.

[CS10] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, 2010. DOI: 10.3233/JCS-2009-0393.

[DF11] Werner Damm and Bernd Finkbeiner. Does it pay to extend the perimeter of a world model? In *Formal Methods - 17th International Symposium on Formal Methods, FM 2011, Proceedings*, Vol. 6664 of *Lecture Notes in Computer Science*, pp. 12–26. Springer, 2011. DOI: 10.1007/978-3-642-21437-0_4.

[DF14] Werner Damm and Bernd Finkbeiner. Automatic compositional synthesis of distributed systems. In *Formal Methods - 19th International Symposium, FM 2014, Proceedings*, Vol. 8442 of *Lecture Notes in Computer Science*, pp. 179–193. Springer, 2014. DOI: 10.1007/978-3-319-06410-9_13.

[DFR16] Werner Damm, Bernd Finkbeiner, and Astrid Rakow. What you really need to know about your neighbor. In *Fifth Workshop on Synthesis, SYNT@CAV 2016, Proceedings*, Vol. 229 of *EPTCS*, pp. 21–34, 2016. DOI: 10.4204/EPTCS.229.4.

[DGM03] Giorgio Delzanno, Maurizio Gabbrielli, and Maria Chiara Meo. Compositional verification of infinite state systems. In *Logic Programming - 19th International Conference, ICLP 2003, Proceedings*, Vol. 2916 of *Lecture Notes in Computer Science*, pp. 47–48. Springer, 2003. DOI: 10.1007/978-3-540-24599-5_4.

[DH01] Werner Damm and David Harel. LSCs: Breathing life into message sequence charts. *Formal Methods Syst. Des.*, 19(1):45–80, 2001. DOI: 10.1023/A:1011227529550.

[DLF$^+$16] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. Spot 2.0 - A framework for LTL and ω-automata manipulation. In *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Proceedings*, Vol. 9938 of *Lecture Notes in Computer Science*, pp. 122–129, 2016. DOI: 10.1007/978-3-319-46520-3_8.

[DR18] Rohit Dureja and Kristin Yvonne Rozier. More scalable LTL model checking via discovering design-space dependencies (D^3). In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Proceedings*, Vol. 10805 of *Lecture Notes in Computer Science*, pp. 309–327. Springer, 2018. DOI: 10.1007/978-3-319-89960-2_17.

[DRC$^+$22] Alexandre Duret-Lutz, Etienne Renault, Maximilien Colange, Florian Renkin, Alexandre Gbaguidi Aisse, Philipp Schlehuber-Caissier, Thomas Medioni, Antoine Martin, Jérôme Dubois, Clément Gillard, and Henrich Lauko. From spot 2.0 to spot 2.10: What's new? In *Computer Aided Verification - 34th International Conference, CAV 2022, Proceedings*, Vol. 13372 of *Lecture Notes in Computer Science*, pp. 174–187. Springer, 2022. DOI: 10.1007/978-3-031-13188-2_9.

[dRdBH$^+$01] Willem P. de Roever, Frank S. de Boer, Ulrich Hannemann, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency Verification: Introduction* *to Compositional and Noncompositional Methods*, Vol. 54 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2001.

[dRH89] Willem P. de Roever and Jozef Hooman. Design and verification in real-time distributed computing: an introduction to compositional methods. In *9th International Symposium on Protocol Specification, Testing and Verification, 1989, Proceedings*, pp. 37–56. North-Holland, 1989.

[dRLP98] Willem P. de Roever, Hans Langmaack, and Amir Pnueli, editors. *Compositionality: The Significant Difference, International Symposium, COMPOS 1997, Revised Lectures*, Vol. 1536 of *Lecture Notes in Computer Science*. Springer, 1998. DOI: 10.1007/3-540-49213-5.

[Ehl11] Rüdiger Ehlers. Unbeast: Symbolic bounded synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Proceedings*, Vol. 6605 of *Lecture Notes in Computer Science*, pp. 272–275. Springer, 2011. DOI: 10.1007/978-3-642-19835-9_25.

[Ehl12] Rüdiger Ehlers. Symbolic bounded synthesis. *Formal Methods Syst. Des.*, 40(2):232–262, 2012. DOI: 10.1007/s10703-011-0137-x.

[EJ91] E. Allen Emerson and Charanjit S. Jutla. Tree automata, mu-calculus and determinacy. In *32nd Annual Symposium on Foundations of Computer Science, FOCS 1991, Proceedings*, pp. 368–377. IEEE Computer Society, 1991. DOI: 10.1109/SFCS.1991.185392.

[Eth23] Intro to ethereum. `https://ethereum.org/en/developers/docs/intro-to-ethereum/`, 2023. Accessed: 2023-02-08.

[Fae09] Marco Faella. Admissible strategies in infinite games over graphs. In *Mathematical Foundations of Computer Science 2009 - 34th International Symposium, MFCS 2009, Proceedings*, Vol. 5734 of *Lecture Notes in Computer Science*, pp. 307–318. Springer, 2009. DOI: 10.1007/978-3-642-03816-7_27.

[FFRT17] Peter Faymonville, Bernd Finkbeiner, Markus N. Rabe, and Leander Tentrup. Encodings of bounded synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Proceedings*, Vol. 10205 of *Lecture Notes in Computer Science*, pp. 354–370, 2017. DOI: 10.1007/978-3-662-54577-5_20.

[FFT17] Peter Faymonville, Bernd Finkbeiner, and Leander Tentrup. Bosy: An experimentation framework for bounded synthesis. In *Computer Aided Verification - 29th International Conference, CAV 2017, Proceedings*, Vol. 10427 of *Lecture Notes in Computer Science*, pp. 325–332. Springer, 2017. DOI: 10.1007/978-3-319-63390-9_17.

[FG17] Bernd Finkbeiner and Paul Gölz. Synthesis in distributed environments. In *37th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2017, Proceedings*, Vol. 93 of *LIPIcs*, pp. 28:1–28:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. DOI: 10.4230/LIPIcs.FSTTCS.2017.28.

[FGHO17] Bernd Finkbeiner, Manuel Gieseking, Jesko Hecking-Harbusch, and Ernst-Rüdiger Olderog. Symbolic vs. bounded synthesis for petri games. In *Sixth Workshop on Synthesis, SYNT@CAV 2017, Proceeings*, Vol. 260 of *EPTCS*, pp. 23–43, 2017. DOI: 10.4204/EPTCS.260.5.

[FGHO22] Bernd Finkbeiner, Manuel Gieseking, Jesko Hecking-Harbusch, and Ernst-Rüdiger Olderog. Global winning conditions in synthesis of distributed systems with causal memory. In *30th EACSL Annual Conference on Computer Science Logic, CSL 2022, Proceedings*, Vol. 216 of *LIPIcs*, pp. 20:1–20:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. DOI: 10.4230/LIPIcs.CSL.2022.20.

[FGP21a] Bernd Finkbeiner, Gideon Geier, and Noemi Passing. Specification decomposition for reactive synthesis. In *NASA Formal Methods - 13th International Symposium, NFM 2021, Proceedings*, Vol. 12673 of *Lecture Notes in Computer Science*, pp. 113–130. Springer, 2021. DOI: 10.1007/978-3-030-76384-8_8.

[FGP21b] Bernd Finkbeiner, Gideon Geier, and Noemi Passing. Specification decomposition for reactive synthesis (full version). 2021, arXiv: 2103.08459.

[FGP22] Bernd Finkbeiner, Gideon Geier, and Noemi Passing. Specification decomposition for reactive synthesis. *Innov. Syst. Softw. Eng.*, 2022. DOI: 10.1007/s11334-022-00462-6.

[FHKP22] Bernd Finkbeiner, Jana Hofmann, Florian Kohn, and Noemi Passing. Reactive synthesis of smart contract control flows. 2022, arXiv: 2205.06039.

[FHKP23] Bernd Finkbeiner, Jana Hofmann, Florian Kohn, and Noemi Passing. Reactive synthesis of smart contract control flows. In *Automated Technology for Verification and Analysis - 21st International Symposium, ATVA 2023, Proceedings*, 2023. (To appear).

[FHP21] Bernd Finkbeiner, Philippe Heim, and Noemi Passing. Temporal stream logic modulo theories (full version). 2021, arXiv: 2104.14988.

[FHP22] Bernd Finkbeiner, Philippe Heim, and Noemi Passing. Temporal stream logic modulo theories. In *Foundations of Software Science and Computation Structures - 25th International Conference, FOSSACS 2022, Proceedings*, Vol. 13242 of *Lecture Notes in Computer Science*, pp. 325–346. Springer, 2022. DOI: 10.1007/978-3-030-99253-8_17.

[Fin15] Bernd Finkbeiner. Bounded synthesis for petri games. In *Correct System Design - Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday, Proceedings*, Vol. 9360 of *Lecture Notes in Computer Science*, pp. 223–237. Springer, 2015. DOI: 10.1007/978-3-319-23506-6_15.

[Fin16] Bernd Finkbeiner. Synthesis of reactive systems. In *Dependable Software Systems Engineering*, Vol. 45 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pp. 72–98. IOS Press, 2016. DOI: 10.3233/978-1-61499-627-9-72.

[FJR09] Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. An antichain algorithm for LTL realizability. In *Computer Aided Verification, 21st International Conference, CAV 2009, Proceedings*, Vol. 5643 of *Lecture Notes in Computer Science*, pp. 263–277. Springer, 2009. DOI: 10.1007/978-3-642-02658-4_22.

[FJR10] Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. Compositional algorithms for LTL synthesis. In *Automated Technology for Verification and Analysis - 8th International Symposium, ATVA 2010, Proceedings*, Vol. 6252 of *Lecture Notes in Computer Science*, pp. 112–127. Springer, 2010. DOI: 10.1007/978-3-642-15643-4_10.

[FJR11] Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. Antichains and compositional algorithms for LTL synthesis. *Formal Methods Syst. Des.*, 39(3):261–296, 2011. DOI: 10.1007/s10703-011-0115-3.

[FK16] Bernd Finkbeiner and Felix Klein. Bounded cycle synthesis. In *Computer Aided Verification - 28th International Conference, CAV 2016, Proceedings*, Vol. 9779 of *Lecture Notes in Computer Science*, pp. 118–135. Springer, 2016. DOI: 10.1007/978-3-319-41528-4_7.

[FK17] Bernd Finkbeiner and Felix Klein. Reactive synthesis: Towards output-sensitive algorithms. In *Dependable Software Systems Engineering*, Vol. 50 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pp. 25–43. IOS Press, 2017. DOI: 10.3233/978-1-61499-810-5-25.

[FKBV14] Andreas Fröhlich, Gergely Kovásznai, Armin Biere, and Helmut Veith. idq: Instantiation-based DQBF solving. In *Fifth Pragmatics of SAT workshop, 2014, Proceedings*, Vol. 27 of *EPiC Series in Computing*, pp. 103–116. EasyChair, 2014. DOI: 10.29007/1s5k.

[FKL10] Dana Fisman, Orna Kupferman, and Yoad Lustig. Rational synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems - 16th International Conference, TACAS 2010, Proceedings*, Vol. 6015 of *Lecture Notes in Computer Science*, pp. 190–204. Springer, 2010. DOI: 10.1007/978-3-642-12002-2_16.

[FKP10] Lu Feng, Marta Z. Kwiatkowska, and David Parker. Compositional verification of probabilistic systems using learning. In *7th International Conference on the*

Quantitative Evaluation of Systems, QEST 2010, Proceedings, pp. 133–142. IEEE Computer Society, 2010. DOI: 10.1109/QEST.2010.24.

[FKPS19] Bernd Finkbeiner, Felix Klein, Ruzica Piskac, and Mark Santolucito. Temporal stream logic: Synthesis beyond the bools. In *Computer Aided Verification - 31st International Conference, CAV 2019, Proceedings*, Vol. 11561 of *Lecture Notes in Computer Science*, pp. 609–629. Springer, 2019. DOI: 10.1007/978-3-030-25540-4_35.

[FMM22] Bernd Finkbeiner, Niklas Metzger, and Yoram Moses. Information flow guided synthesis. In *Computer Aided Verification - 34th International Conference, CAV 2022, Proceedings*, Vol. 13372 of *Lecture Notes in Computer Science*, pp. 505–525. Springer, 2022. DOI: 10.1007/978-3-031-13188-2_25.

[FMS97] Bernd Finkbeiner, Zohar Manna, and Henny Sipma. Deductive verification of modular systems. In *Compositionality: The Significant Difference, International Symposium, COMPOS 1997, Revised Lectures*, Vol. 1536 of *Lecture Notes in Computer Science*, pp. 239–275. Springer, 1997. DOI: 10.1007/3-540-49213-5_9.

[FO17] Bernd Finkbeiner and Ernst-Rüdiger Olderog. Petri games: Synthesis of distributed systems with causal memory. *Inf. Comput.*, 253:181–203, 2017. DOI: 10.1016/j.ic.2016.07.006.

[FP20a] Bernd Finkbeiner and Noemi Passing. Dependency-based compositional synthesis. In *Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Proceedings*, Vol. 12302 of *Lecture Notes in Computer Science*, pp. 447–463. Springer, 2020. DOI: 10.1007/978-3-030-59152-6_25.

[FP20b] Bernd Finkbeiner and Noemi Passing. Dependency-based compositional synthesis (full version). 2020, arXiv: 2007.06941.

[FP21a] Bernd Finkbeiner and Noemi Passing. Compositional synthesis of modular systems. In *Automated Technology for Verification and Analysis - 19th International Symposium, ATVA 2021, Proceedings*, Vol. 12971 of *Lecture Notes in Computer Science*, pp. 303–319. Springer, 2021. DOI: 10.1007/978-3-030-88885-5_20.

[FP21b] Bernd Finkbeiner and Noemi Passing. Compositional synthesis of modular systems (full version). 2021, arXiv: 2106.14783.

[FP22a] Bernd Finkbeiner and Noemi Passing. Compositional synthesis of modular systems. *Innov. Syst. Softw. Eng.*, 18(3):455–469, 2022. DOI: 10.1007/s11334-022-00450-w.

[FP22b] Bernd Finkbeiner and Noemi Passing. Synthesizing dominant strategies for liveness. In *42nd IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2022, Proceedings*, Vol. 250 of *LIPIcs*, pp. 37:1–37:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. DOI: 10.4230/LIPIcs.FSTTCS.2022.37.

[FP22c] Bernd Finkbeiner and Noemi Passing. Synthesizing dominant strategies for liveness (full version). 2022, arXiv: 2210.01660.

[FPS08] Bernd Finkbeiner, Hans-Jörg Peter, and Sven Schewe. RESY: requirement synthesis for compositional model checking. In *Tools and Algorithms for the Construction and Analysis of Systems - 14th International Conference, TACAS 2008, Proceedings*, Vol. 4963 of *Lecture Notes in Computer Science*, pp. 463–466. Springer, 2008. DOI: 10.1007/978-3-540-78800-3_35.

[FS05] Bernd Finkbeiner and Sven Schewe. Uniform distributed synthesis. In *20th IEEE Symposium on Logic in Computer Science, LICS 2005, Proceedings*, pp. 321–330. IEEE Computer Society, 2005. DOI: 10.1109/LICS.2005.53.

[FS07] Bernd Finkbeiner and Sven Schewe. SMT-based synthesis of distributed systems. In *Second Workshop on Automated Formal Methods, AFM 2007, Proceedings*, pp. 69–76, 2007. DOI: 10.1145/1345169.1345178.

[FS13] Bernd Finkbeiner and Sven Schewe. Bounded synthesis. *Int. J. Softw. Tools Technol. Transf.*, 15(5-6):519–539, 2013. DOI: 10.1007/s10009-012-0228-z.

[FSB06] Bernd Finkbeiner, Sven Schewe, and Matthias Brill. Automatic synthesis of assumptions for compositional model checking. In *Formal Techniques for Networked and Distributed Systems - 26th International Conference, FORTE 2006, Proceedings*, Vol. 4229 of *Lecture Notes in Computer Science*, pp. 143–158. Springer, 2006. DOI: 10.1007/11888116_12.

[FW05] Carsten Fritz and Thomas Wilke. Simulation relations for alternating büchi automata. *Theor. Comput. Sci.*, 338(1-3):275–314, 2005. DOI: 10.1016/j.tcs.2005.01.016.

[GCH13] Yashdeep Godhal, Krishnendu Chatterjee, and Thomas A. Henzinger. Synthesis of AMBA AHB from formal specification: a case study. *Int. J. Softw. Tools Technol. Transf.*, 15(5-6):585–601, 2013. DOI: 10.1007/s10009-011-0207-9.

[Gei20] Gideon S. Geier. Specification decomposition for reactive synthesis. Bachelor's thesis, Saarland University, 2020.

[GGMW13] Blaise Genest, Hugo Gimbert, Anca Muscholl, and Igor Walukiewicz. Asynchronous games over tree architectures. In *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Proceedings*, Vol. 7966 of *Lecture Notes in Computer Science*, pp. 275–286. Springer, 2013. DOI: 10.1007/978-3-642-39212-2_26.

[GHKF19] Gideon Geier, Philippe Heim, Felix Klein, and Bernd Finkbeiner. Syntroids: Synthesizing a game for fpgas using temporal logic specifications. In *Formal Methods in Computer Aided Design, FMCAD 2019, Proceedings*, pp. 138–146. IEEE, 2019. DOI: 10.23919/FMCAD.2019.8894261.

[GHY21] Manuel Gieseking, Jesko Hecking-Harbusch, and Ann Yanich. A web interface for petri nets with transits and petri games. In *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Proceedings*, Vol. 12652 of *Lecture Notes in Computer Science*, pp. 381–388. Springer, 2021. DOI: 10.1007/978-3-030-72013-1_22.

[Gim17] Hugo Gimbert. On the control of asynchronous automata. In *37th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2017, Proceedings*, Vol. 93 of *LIPIcs*, pp. 30:1–30:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. DOI: 10.4230/LIPIcs.FSTTCS.2017.30.

[Gim22] Hugo Gimbert. Distributed asynchronous games with causal memory are undecidable. *Log. Methods Comput. Sci.*, 18(3), 2022. DOI: 10.46298/lmcs-18(3:30)2022.

[GK13] Joel Greenyer and Ekkart Kindler. Compositional synthesis of controllers from scenario-based assume-guarantee specifications. In *Model-Driven Engineering Languages and Systems - 16th International Conference, MODELS 2013, Proceedings*, Vol. 8107 of *Lecture Notes in Computer Science*, pp. 774–789. Springer, 2013. DOI: 10.1007/978-3-642-41533-3_47.

[GLZ04a] Paul Gastin, Benjamin Lerman, and Marc Zeitoun. Distributed games and distributed control for asynchronous systems. In *Theoretical Informatics - 6th Latin American Symposium, LATIN 2004, Proceedings*, Vol. 2976 of *Lecture Notes in Computer Science*, pp. 455–465. Springer, 2004. DOI: 10.1007/978-3-540-24698-5_49.

[GLZ04b] Paul Gastin, Benjamin Lerman, and Marc Zeitoun. Distributed games with causal memory are decidable for series-parallel systems. In *Foundations of Software Technology and Theoretical Computer Science - 24th International Conference, FSTTCS 2004, Proceedings*, Vol. 3328 of *Lecture Notes in Computer Science*, pp. 275–286. Springer, 2004. DOI: 10.1007/978-3-540-30538-5_23.

[GM09] Alwyn Goodloe and César A. Muñoz. Compositional verification of a communication protocol for a remotely operated vehicle. In *Formal Methods for Industrial Critical Systems - 14th International Workshop, FMICS 2009, Proceedings*, Vol. 5825 of *Lecture Notes in Computer Science*, pp. 86–101. Springer, 2009. DOI: 10.1007/978-3-642-04570-7_8.

[GMF08] Anubhav Gupta, Kenneth L. McMillan, and Zhaohui Fu. Automated assumption generation for compositional verification. *Formal Methods Syst. Des.*, 32(3):285–301, 2008. DOI: 10.1007/s10703-008-0050-0.

[GNP18] Dimitra Giannakopoulou, Kedar S. Namjoshi, and Corina S. Pasareanu. Compositional reasoning. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pp. 345–383. Springer, 2018. DOI: 10.1007/978-3-319-10575-8_12.

[GO01] Paul Gastin and Denis Oddoux. Fast LTL to büchi automata translation. In *Computer Aided Verification - 13th International Conference, CAV 2001, Proceedings*, Vol. 2102 of *Lecture Notes in Computer Science*, pp. 53–65. Springer, 2001. DOI: 10.1007/3-540-44585-4_6.

[GPB02] Dimitra Giannakopoulou, Corina S. Pasareanu, and Howard Barringer. Assumption generation for software component verification. In *17th IEEE International Conference on Automated Software Engineering, ASE 2002, Proceedings*, pp. 3–12. IEEE Computer Society, 2002. DOI: 10.1109/ASE.2002.1114984.

[GPM+20] Dimitra Giannakopoulou, Thomas Pressburger, Anastasia Mavridou, Julian Rhein, Johann Schumann, and Nija Shi. Formal requirements elicitation with FRET. In *Joint Proceedings of REFSQ-2020 Workshops co-located with the 26th International Conference on Requirements Engineering: Foundation for Software Quality, REFSQ 2020*, Vol. 2584 of *CEUR Workshop Proceedings*, 2020.

[GV13] Giuseppe De Giacomo and Moshe Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *23rd International Joint Conference on Artificial Intelligence, IJCAI 2013, Proceedings*, pp. 854–860. IJCAI/AAAI, 2013.

[HM08] David Harel and Shahar Maoz. Assert and negate revisited: Modal semantics for UML sequence diagrams. *Softw. Syst. Model.*, 7(2):237–252, 2008. DOI: 10.1007/s10270-007-0054-z.

[HM19] Jesko Hecking-Harbusch and Niklas O. Metzger. Efficient trace encodings of bounded synthesis for asynchronous distributed systems. In *Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Proceedings*, Vol. 11781 of *Lecture Notes in Computer Science*, pp. 369–386. Springer, 2019. DOI: 10.1007/978-3-030-31784-3_22.

[Hoo91] Jozef Hooman. *Specification and Compositional Verification of Real-Time Systems*, Vol. 558 of *Lecture Notes in Computer Science*. Springer, 1991. DOI: 10.1007/3-540-54947-1.

[HP84] David Harel and Amir Pnueli. On the development of reactive systems. In *Logics and Models of Concurrent Systems, Proceedings*, Vol. 13 of *NATO ASI Series*, pp. 477–498. Springer, 1984. DOI: 10.1007/978-3-642-82453-1_17.

[HSLL97] Klaus Havelund, Arne Skou, Kim Guldstrand Larsen, and Kristian Lund. Formal modeling and analysis of an audio/video protocol: an industrial case study using UPPAAL. In *18th IEEE Real-Time Systems Symposium, RTSS 1997, Proceedings*, pp. 2–13. IEEE Computer Society, 1997. DOI: 10.1109/REAL.1997.641264.

[HT73] John E. Hopcroft and Robert Endre Tarjan. Efficient algorithms for graph manipulation. *Commun. ACM*, 16(6):372–378, 1973. DOI: 10.1145/362248.362272.

[JB16] Swen Jacobs and Roderick Bloem. The reactive synthesis competition: SYNTCOMP 2016 and beyond. In *Fifth Workshop on Synthesis, SYNT@CAV 2016, Proceedings*, Vol. 229 of *EPTCS*, pp. 133–148, 2016. DOI: 10.4204/EPTCS.229.11.

[JBB+15] Swen Jacobs, Roderick Bloem, Romain Brenguier, Robert Könighofer, Guillermo A. Pérez, Jean-François Raskin, Leonid Ryzhyk, Ocan Sankur, Martina Seidl, Leander Tentrup, and Adam Walker. The second reactive synthesis competition (SYNTCOMP 2015). In *Fourth Workshop on Synthesis, SYNT 2015, Proceedings*, Vol. 202 of *EPTCS*, pp. 27–57, 2015. DOI: 10.4204/EPTCS.202.4.

[JBB+16] Swen Jacobs, Roderick Bloem, Romain Brenguier, Ayrat Khalimov, Felix Klein, Robert Könighofer, Jens Kreber, Alexander Legg, Nina Narodytska, Guillermo A. Pérez, Jean-François Raskin, Leonid Ryzhyk, Ocan Sankur, Martina Seidl, Leander Tentrup, and Adam Walker. The 3rd reactive synthesis competition (SYNTCOMP 2016): Benchmarks, participants & results. In *Fifth Workshop on Synthesis, SYNT@CAV 2016, Proceedings*, Vol. 229 of *EPTCS*, pp. 149–177, 2016. DOI: 10.4204/EPTCS.229.12.

[JBB+17a] Swen Jacobs, Nicolas Basset, Roderick Bloem, Romain Brenguier, Maximilien Colange, Peter Faymonville, Bernd Finkbeiner, Ayrat Khalimov, Felix Klein, Thibaud Michaud, Guillermo A. Pérez, Jean-François Raskin, Ocan Sankur, and Leander Tentrup. The 4th reactive synthesis competition (SYNTCOMP 2017): Benchmarks, participants & results. In *Sixth Workshop on Synthesis, SYNT@CAV 2017, Proceedings*, Vol. 260 of *EPTCS*, pp. 116–143, 2017. DOI: 10.4204/EPTCS.260.10.

[JBB+17b] Swen Jacobs, Roderick Bloem, Romain Brenguier, Rüdiger Ehlers, Timotheus Hell, Robert Könighofer, Guillermo A. Pérez, Jean-François Raskin, Leonid Ryzhyk, Ocan Sankur, Martina Seidl, Leander Tentrup, and Adam Walker. The first reactive synthesis competition (SYNTCOMP 2014). *Int. J. Softw. Tools Technol. Transf.*, 19(3):367–390, 2017. DOI: 10.1007/s10009-016-0416-3.

[JBC+19] Swen Jacobs, Roderick Bloem, Maximilien Colange, Peter Faymonville, Bernd Finkbeiner, Ayrat Khalimov, Felix Klein, Michael Luttenberger, Philipp J. Meyer, Thibaud Michaud, Mouhammad Sakr, Salomon Sickert, Leander Tentrup, and Adam Walker. The 5th reactive synthesis competition (SYNTCOMP 2018): Benchmarks, participants & results. 2019, arXiv: 1904.07736.

[JGK+15] Jean-Baptiste Jeannin, Khalil Ghorbal, Yanni Kouskoulas, Ryan W. Gardner, Aurora C. Schmidt, Erik Zawadzki, and André Platzer. Formal verification of ACAS x, an industrial airborne collision avoidance system. In *International Conference on Embedded Software, EMSOFT 2015, Proceedings*, pp. 127–136. IEEE, 2015. DOI: 10.1109/EMSOFT.2015.7318268.

[JKA+20] Jingwei Ji, Ranjay Krishna, Ehsan Adeli, Juan C. Niebles, Olga Russakovsky, and Fei-Fei Li. Compositionality in computer vision. `https://ai.stanford.edu/~jingweij/cicv/`, 2020. Accessed: 2023-01-16.

[JKS16] Swen Jacobs, Felix Klein, and Sebastian Schirmer. A high-level LTL synthesis format: TLSF v1.1. In *Fifth Workshop on Synthesis, SYNT@CAV 2016, Proceedings*, Vol. 229 of *EPTCS*, pp. 112–132, 2016. DOI: 10.4204/EPTCS.229.10.

[JM01] Ranjit Jhala and Kenneth L. McMillan. Microarchitecture verification by compositional model checking. In *Computer Aided Verification - 13th International Conference, CAV 2001, Proceedings*, Vol. 2102 of *Lecture Notes in Computer Science*, pp. 396–410. Springer, 2001. DOI: 10.1007/3-540-44585-4_40.

[Job07] Barbara Jobstmann. *Applications and Optimizations for LTL synthesis*. PhD thesis, Graz University of Technology, 2007.

[Jon83] Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983. DOI: 10.1145/69575.69577.

[JPA+22] Swen Jacobs, Guillermo A. Pérez, Remco Abraham, Véronique Bruyère, Michaël Cadilhac, Maximilien Colange, Charly Delfosse, Tom van Dijk, Alexandre Duret-Lutz, Peter Faymonville, Bernd Finkbeiner, Ayrat Khalimov, Felix Klein, Michael Luttenberger, Klara J. Meyer, Thibaud Michaud, Adrien Pommellet, Florian Renkin, Philipp Schlehuber-Caissier, Mouhammad Sakr, Salomon Sickert, Gaëtan Staquet, Clément Tamines, Leander Tentrup, and Adam Walker. The reactive synthesis competition (SYNTCOMP): 2018-2021. 2022, arXiv: 2206.00251.

[Jur00] Marcin Jurdzinski. Small progress measures for solving parity games. In *17th Annual Symposium on Theoretical Aspects of Computer Science, STACS 2000, Proceedings*, Vol. 1770 of *Lecture Notes in Computer Science*, pp. 290–301. Springer, 2000. DOI: 10.1007/3-540-46541-3_24.

[KB17] Ayrat Khalimov and Roderick Bloem. Bounded synthesis for streett, rabin, and CTL*. In *Computer Aided Verification - 29th International Conference, CAV 2017, Proceedings*, Vol. 10427 of *Lecture Notes in Computer Science*, pp. 333–352. Springer, 2017. DOI: 10.1007/978-3-319-63390-9_18.

[KF18] Abhishek Ninad Kulkarni and Jie Fu. A compositional approach to reactive games under temporal logic specifications. In *Annual American Control Conference, ACC 2018, Proceedings*, pp. 2356–2362. IEEE, 2018. DOI: 10.23919/ACC.2018.8431867.

[Kha21] Ayrat Khalimov. sdf-hoa. `https://github.com/5nizza/sdf-hoa`, 2021. Accessed: 2023-01-16.

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[KP10] Uri Klein and Amir Pnueli. Revisiting synthesis of GR(1) specifications. In *Hardware and Software: Verification and Testing - 6th International Haifa Verification*

Conference, HVC 2010, Revised Selected Papers, Vol. 6504 of *Lecture Notes in Computer Science*, pp. 161–181. Springer, 2010. DOI: 10.1007/978-3-642-19583-9_16.

[KPV06] Orna Kupferman, Nir Piterman, and Moshe Y. Vardi. Safraless compositional synthesis. In *Computer Aided Verification - 18th International Conference, CAV 2006, Proceedings*, Vol. 4144 of *Lecture Notes in Computer Science*, pp. 31–44. Springer, 2006. DOI: 10.1007/11817963_6.

[KPV14] Orna Kupferman, Giuseppe Perelli, and Moshe Y. Vardi. Synthesis with rational environments. In *Multi-Agent Systems – 12th European Conference, EUMAS 2014, Revised Selected Papers*, Vol. 8953 of *Lecture Notes in Computer Science*, pp. 219–235. Springer, 2014. DOI: 10.1007/978-3-319-17130-2_15.

[KS09] Hillel Kugler and Itai Segall. Compositional synthesis of reactive systems from live sequence chart specifications. In *Tools and Algorithms for the Construction and Analysis of Systems - 15th International Conference, TACAS 2009, Proceedings*, Vol. 5505 of *Lecture Notes in Computer Science*, pp. 77–91. Springer, 2009. DOI: 10.1007/978-3-642-00768-2_9.

[KV00] Orna Kupferman and Moshe Y. Vardi. Synthesis with incomplete informatio. *Advances in Temporal Logic*, 16:109–127, 2000. DOI: 10.1007/978-94-015-9586-5_6.

[KV01] Orna Kupferman and Moshe Y. Vardi. Synthesizing distributed systems. In *16th Annual IEEE Symposium on Logic in Computer Science, LICS 2001, Proceedings*, pp. 389–398. IEEE Computer Society, 2001. DOI: 10.1109/LICS.2001.932514.

[KV05] Orna Kupferman and Moshe Y. Vardi. Safraless decision procedures. In *46th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2005, Proceedings*, pp. 531–542. IEEE Computer Society, 2005. DOI: 10.1109/SFCS.2005.66.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977. DOI: 10.1109/TSE.1977.229904.

[Lam97] Leslie Lamport. Composition: A way to make proofs harder. In *Compositionality: The Significant Difference - International Symposium, COMPOS 1997, Revised Lectures*, Vol. 1536 of *Lecture Notes in Computer Science*, pp. 402–423. Springer, 1997. DOI: 10.1007/3-540-49213-5_15.

[LDS11] Wenchao Li, Lili Dworkin, and Sanjit A. Seshia. Mining assumptions for synthesis. In *9th IEEE/ACM International Conference on Formal Methods and Models for Codesign, MEMOCODE 2011, Proceedings*, pp. 43–50. IEEE, 2011. DOI: 10.1109/MEMCOD.2011.5970509.

[LL95] François Laroussinie and Kim Guldstrand Larsen. Compositional model checking of real time systems. In *Concurrency Theory - 6th International Conference, CONCUR 1995, Proceedings*, Vol. 962 of *Lecture Notes in Computer Science*, pp. 27–41. Springer, 1995. DOI: 10.1007/3-540-60218-6_3.

[LM92] Kim Guldstrand Larsen and Robin Milner. A compositional protocol verification using relativized bisimulation. *Inf. Comput.*, 99(1):80–108, 1992. DOI: 10.1016/0890-5401(92)90025-B.

[LMM21] Frédéric Lang, Radu Mateescu, and Franco Mazzanti. Compositional verification of concurrent systems by combining bisimulations. *Formal Methods Syst. Des.*, 58(1-2):83–125, 2021. DOI: 10.1007/s10703-021-00360-w.

[LMS20] Michael Luttenberger, Philipp J. Meyer, and Salomon Sickert. Practical synthesis of reactive systems from LTL specifications via parity games. *Acta Informatica*, 57(1-2):3–36, 2020. DOI: 10.1007/s00236-019-00349-3.

[LS92] Kim Guldstrand Larsen and Arne Skou. Compositional verification of probabilistic processes. In *Third International Conference on Concurrency Theory, CONCUR 1992, Proceedings*, Vol. 630 of *Lecture Notes in Computer Science*, pp. 456–471. Springer, 1992. DOI: 10.1007/BFb0084809.

[LT91] Kim Guldstrand Larsen and Bent Thomsen. Partial specifications and compositional verification. *Theor. Comput. Sci.*, 88(1):15–32, 1991. DOI: 10.1016/0304-3975(91)90071-9.

[LTVZ21] Yong Li, Andrea Turrini, Moshe Y. Vardi, and Lijun Zhang. Synthesizing good-enough strategies for ltlf specifications. In *30th International Joint Conference on Artificial Intelligence, IJCAI 2021, Proceedings*, pp. 4144–4151, 2021. DOI: 10.24963/ijcai.2021/570.

[Mal20] Kaushik Mallik. Agnes. `https://github.com/kmallik/Agnes`, 2020. Accessed: 2023-01-17.

[MC81] Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Trans. Software Eng.*, 7(4):417–426, 1981. DOI: 10.1109/TSE.1981.230844.

[MC18] Thibaud Michaud and Maximilien Colange. Reactive synthesis from ltl specification with spot. *Proceedings of SYNT@CAV*, 2018.

[McM99] Kenneth L. McMillan. Verification of infinite state systems by compositional model checking. In *Correct Hardware Design and Verification Methods - 10th Advanced Research Working Conference, CHARME 1999, Proceedings*, Vol. 1703 of *Lecture Notes in Computer Science*, pp. 219–234. Springer, 1999. DOI: 10.1007/3-540-48153-2_17.

[McM01] Kenneth L. McMillan. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In *Correct Hardware Design and Verification Methods - 11th Advanced Research Working Conference, CHARME 2001, Proceedings*, Vol. 2144 of *Lecture Notes in Computer Science*, pp. 179–195. Springer, 2001. DOI: 10.1007/3-540-44798-9_17.

[MH84] Satoru Miyano and Takeshi Hayashi. Alternating finite automata on ω-words. *Theor. Comput. Sci.*, 32:321–330, 1984. DOI: 10.1016/0304-3975(84)90049-5.

[MKG+21] Anastasia Mavridou, Andreas Katis, Dimitra Giannakopoulou, David Kooi, Thomas Pressburger, and Michael W. Whalen. From partial to global assume-guarantee contracts: Compositional realizability analysis in FRET. In *Formal Methods - 24th International Symposium, FM 2021, Proceedings*, Vol. 13047 of *Lecture Notes in Computer Science*, pp. 503–523. Springer, 2021. DOI: 10.1007/978-3-030-90870-6_27.

[ML18] Anastasia Mavridou and Aron Laszka. Designing secure ethereum smart contracts: A finite state machine based approach. In *Financial Cryptography and Data Security - 22nd International Conference, FC 2018, Revised Selected Papers*, Vol. 10957 of *Lecture Notes in Computer Science*, pp. 523–540. Springer, 2018. DOI: 10.1007/978-3-662-58387-6_28.

[MLSD19] Anastasia Mavridou, Aron Laszka, Emmanouela Stachtiari, and Abhishek Dubey. Verisolid: Correct-by-design smart contracts for ethereum. In *Financial Cryptography and Data Security - 23rd International Conference, FC 2019, Revised Selected Papers*, Vol. 11598 of *Lecture Notes in Computer Science*, pp. 446–465. Springer, 2019. DOI: 10.1007/978-3-030-32101-7_27.

[MM22] Gary Marcus and Raphaël Millière. The challenge of compositionality for ai, workshop. `https://compositionalintelligence.github.io`, 2022. Accessed: 2023-01-16.

[MMSZ20] Rupak Majumdar, Kaushik Mallik, Anne-Kathrin Schmuck, and Damien Zufferey. Assume-guarantee distributed synthesis. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 39(11):3215–3226, 2020. DOI: 10.1109/TCAD.2020.3012641.

[MSL18] Philipp J. Meyer, Salomon Sickert, and Michael Luttenberger. Strix: Explicit reactive synthesis strikes back! In *Computer Aided Verification - 30th International Conference, CAV 2018, Proceedings*, Vol. 10981 of *Lecture Notes in Computer Science*, pp. 578–586. Springer, 2018. DOI: 10.1007/978-3-319-96145-3_31.

[MSS88] David E. Muller, Ahmed Saoudi, and Paul E. Schupp. Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time. In *Third Annual Symposium on Logic in Computer Science, LICS 1988, Proceedings*, pp. 422–427. IEEE Computer Society, 1988. DOI: 10.1109/LICS.1988.5139.

[MT02] P. Madhusudan and P. S. Thiagarajan. A decidable class of asynchronous distributed controllers. In *Concurrency Theory, 13th International Conference, CONCUR 2002, Proceedings*, Vol. 2421 of *Lecture Notes in Computer Science*, pp. 145–160. Springer, 2002. DOI: 10.1007/3-540-45694-5_11.

[MTY05] P. Madhusudan, P. S. Thiagarajan, and Shaofa Yang. The MSO theory of connectedly communicating processes. In *Foundations of Software Technology and Theoretical Computer Science - 25th International Conference, FSSTCS 2005, Proceedings*, Vol. 3821 of *Lecture Notes in Computer Science*, pp. 201–212. Springer, 2005. DOI: 10.1007/11590156_16.

[MW03] Swarup Mohalik and Igor Walukiewicz. Distributed games. In *Foundations of Software Technology and Theoretical Computer Science, 23rd Conference, FSTTCS 2003, Proceedings*, Vol. 2914 of *Lecture Notes in Computer Science*, pp. 338–351. Springer, 2003. DOI: 10.1007/978-3-540-24597-1_29.

[MW14] Anca Muscholl and Igor Walukiewicz. Distributed synthesis for acyclic architectures. In *34th International Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2014, Proceedings*, Vol. 29 of *LIPIcs*, pp. 639–651. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2014. DOI: 10.4230/LIPIcs.FSTTCS.2014.639.

[MWW08] Björn Metzler, Heike Wehrheim, and Daniel Wonisch. Decomposition for compositional verification. In *Formal Methods and Software Engineering - 10th International Conference on Formal Engineering Methods, ICFEM 2008, Proceedings*, Vol. 5256 of *Lecture Notes in Computer Science*, pp. 105–125. Springer, 2008. DOI: 10.1007/978-3-540-88194-0_9.

[NAS20] NASA. Fret: Formal requirements elicitation tool. `https://software.nasa.gov/software/ARC-18066-1`, 2020. Accessed: 2023-01-17.

[NMA08] Wonhong Nam, P. Madhusudan, and Rajeev Alur. Automatic symbolic compositional verification by learning assumptions. *Formal Methods Syst. Des.*, 32(3):207–234, 2008. DOI: 10.1007/s10703-008-0055-8.

[NPW81] Mogens Nielsen, Gordon D. Plotkin, and Glynn Winskel. Petri nets, event structures and domains, part I. *Theor. Comput. Sci.*, 13:85–108, 1981. DOI: 10.1016/0304-3975(81)90112-2.

[NRTV07] Noam Nisan, Tim Roughgarden, Éva Tardos, and Vijay V. Vazirani, editors. *Algorithmic Game Theory*. Cambridge University Press, 2007. DOI: 10.1017/CBO9780511800481.

[NT16] Kedar S. Namjoshi and Richard J. Trefler. Parameterized compositional model checking. In *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Proceedings*, Vol. 9636 of *Lecture Notes in Computer Science*, pp. 589–606. Springer, 2016. DOI: 10.1007/978-3-662-49674-9_39.

[Old91] Ernst-Rüdiger Olderog. *Nets, Terms and Formulas: Three Views of Concurrent Processes and Their Relationship*. Cambridge University Press, 1991. DOI: 10.1017/CBO9780511526589.

[Pla11] André Platzer. Logic and compositional verification of hybrid systems - (invited tutorial). In *Computer Aided Verification - 23rd International Conference, CAV 2011, Proceedings*, Vol. 6806 of *Lecture Notes in Computer Science*, pp. 28–43. Springer, 2011. DOI: 10.1007/978-3-642-22110-1_4.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, FOCS 1977, Proceedings*, pp. 46–57. IEEE Computer Society, 1977. DOI: 10.1109/SFCS.1977.32.

[Pnu84] Amir Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and Models of Concurrent Systems, 1984, Proceedings*, Vol. 13 of *NATO ASI Series*, pp. 123–144. Springer, 1984. DOI: 10.1007/978-3-642-82453-1_5.

[PPS06] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive(1) designs. In *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Proceedings*, Vol. 3855 of *Lecture Notes in Computer Science*, pp. 364–380. Springer, 2006. DOI: 10.1007/11609773_24.

[PR89a] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages, POPL 1989, Proceedings*, pp. 179–190. ACM Press, 1989. DOI: 10.1145/75277.75293.

[PR89b] Amir Pnueli and Roni Rosner. On the synthesis of an asynchronous reactive module. In *Automata, Languages and Programming - 16th International Colloquium, ICALP 1989, Proceedings*, Vol. 372 of *Lecture Notes in Computer Science*, pp. 652–671. Springer, 1989. DOI: 10.1007/BFb0035790.

[PR90] Amir Pnueli and Roni Rosner. Distributed reactive systems are hard to synthesize. In *31st Annual Symposium on Foundations of Computer Science, FOCS 1990, Proceedings*, pp. 746–757. IEEE Computer Society, 1990. DOI: 10.1109/FSCS.1990.89597.

[Rab72] Michael O. Rabin. Automata on infinite objects and chruch's problem. *American Mathematical Society*, 1972. DOI: 10.1090/cbms/013.

[Rei85] Wolfgang Reisig. *Petri Nets: An Introduction*, Vol. 4 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985. DOI: 10.1007/978-3-642-69968-9.

[RSDP22] Florian Renkin, Philipp Schlehuber, Alexandre Duret-Lutz, and Adrien Pommellet. Improvements to ltlsynt. 2022, arXiv: 2201.05376.

[Saf88] Shmuel Safra. On the complexity of omega-automata. In *29th Annual Symposium on Foundations of Computer Science, FOCS 1988, Proceedings*, pp. 319–327. IEEE Computer Society, 1988. DOI: 10.1109/SFCS.1988.21948.

[SB00] Fabio Somenzi and Roderick Bloem. Efficient büchi automata from LTL formulae. In *Computer Aided Verification - 12th International Conference, CAV 2000, Proceedings*, Vol. 1855 of *Lecture Notes in Computer Science*, pp. 248–263. Springer, 2000. DOI: 10.1007/10722167_21.

[SC07] Nishant Sinha and Edmund M. Clarke. Sat-based compositional verification using lazy learning. In *Computer Aided Verification - 19th International Conference, CAV 2007, Proceedings*, Vol. 4590 of *Lecture Notes in Computer Science*, pp. 39–54. Springer, 2007. DOI: 10.1007/978-3-540-73368-3_8.

[Sch14] Sven Schewe. Distributed synthesis is simply undecidable. *Inf. Process. Lett.*, 114(4):203–207, 2014. DOI: 10.1016/j.ipl.2013.11.012.

[SF07] Sven Schewe and Bernd Finkbeiner. Semi-automatic distributed synthesis. *Int. J. Found. Comput. Sci.*, 18(1):113–138, 2007. DOI: 10.1142/S0129054107004590.

[Sol16] Solidity documentation. `https://docs.soliditylang.org/en/v0.8.18/`, 2016. Accessed: 2023-01-23.

[SS13] Saqib Sohail and Fabio Somenzi. Safety first: A two-stage algorithm for the synthesis of reactive systems. *Int. J. Softw. Tools Technol. Transf.*, 15(5-6):433–454, 2013. DOI: 10.1007/s10009-012-0224-3.

[Ste13] Dominik Steiger. Synthesis of dominant strategies. Bachelor's thesis, Saarland University, 2013.

[SYN20] SYNTCOMP20. The reactive synthesis competition 2020 - benchmarks, participants, and results. `http://www.syntcomp.org/syntcomp-2020-results/`, 2020. Accessed: 2023-01-31.

[SYN22] SYNTCOMP22. The reactive synthesis competition 2022 - benchmarks, participants, and results. `http://www.syntcomp.org/syntcomp-2022-results/`, 2022. Accessed: 2023-01-31.

[Ten19] Leander Tentrup. *Symbolic Reactive Synthesis*. PhD thesis, Saarland University, 2019.

[Tho09] Wolfgang Thomas. Facets of synthesis: Revisiting church's problem. In *Foundations of Software Science and Computational Structures - 12th International Conference, FOSSACS 2009, Proceedings*, Vol. 5504 of *Lecture Notes in Computer Science*, pp. 1–14. Springer, 2009. DOI: 10.1007/978-3-642-00596-1_1.

[TR19] Leander Tentrup and Markus N. Rabe. Clausal abstraction for DQBF. In *Theory and Applications of Satisfiability Testing - 22nd International Conference, SAT 2019, Proceedings*, Vol. 11628 of *Lecture Notes in Computer Science*, pp. 388–405. Springer, 2019. DOI: 10.1007/978-3-030-24258-9_27.

[Tri16] Stavros Tripakis. Compositionality in the science of system design. *Proc. IEEE*, 104(5):960–972, 2016. DOI: 10.1109/JPROC.2015.2510366.

[VW94] Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Inf. Comput.*, 115(1):1–37, 1994. DOI: 10.1006/inco.1994.1092.

[WLC$^+$19] Yuepeng Wang, Shuvendu K. Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, Immad Naseer, and Kostas Ferles. Formal verification of workflow policies for smart contracts in azure blockchain. In *Verified Software. Theories, Tools, and Experiments - 11th International Conference, VSTTE 2019, Revised Selected Papers*, Vol. 12031 of *Lecture Notes in Computer Science*, pp. 87–106. Springer, 2019. DOI: 10.1007/978-3-030-41600-3_7.

[Woo14] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, eip-150 revision. 2014. URL: https://gavwood.com/paper.pdf.

[ZH95] Ping Zhou and Jozef Hooman. Formal specification and compositional verification of an atomic broadcast protocol. *Real Time Syst.*, 9(2):119–145, 1995. DOI: 10.1007/BF01088854.

[Zie87] Wieslaw Zielonka. Notes on finite asynchronous automata. *RAIRO Theor. Informatics Appl.*, 21(2):99–135, 1987. DOI: 10.1051/ita/1987210200991.

Index