# Synthesizing Stream Control

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

by
Felix Klein

Saarbrücken, 2020

| | |
|---|---|
| Tag des Kolloquiums: | 8. September 2020 |
| Dekan: | Prof. Dr. Thomas Schuster |
| Vorsitzender des Prüfungsausschuss: | Prof. Dr. Sebastian Hack |
| Gutachter: | Prof. Bernd Finkbeiner, Ph.D.<br>Prof. Orna Kupferman, Ph.D.<br>Prof. Ruzica Piskac, Ph.D.<br>Barbara Jobstmann, Ph.D. |
| akademischer Mitarbeiter: | Dr. Rahul Gopinath |

# Abstract

For the management of reactive systems, controllers must coordinate time, data streams, and data transformations, all joint by the high level perspective of their control flow. This control flow is required to drive the system correctly and continuously, which turns the development into a challenge. The process is error-prone, time consuming, unintuitive, and costly. An attractive alternative is to synthesize the system instead, where the developer only needs to specify the desired behavior. The synthesis engine then automatically takes care of all the technical details. However, while current algorithms for the synthesis of reactive systems are well-suited to handle control, they fail on complex data transformations due to the complexity of the comparably large data space. Thus, to overcome the challenge of explicitly handling the data we must separate data and control.

We introduce *Temporal Stream Logic* (TSL), a logic which exclusively argues about the control of the controller, while treating data and functional transformations as interchangeable black-boxes. In TSL it is possible to specify control flow properties independently of the complexity of the handled data. Furthermore, with TSL at hand a synthesis engine can check for realizability, even without a concrete implementation of the data transformations. We present a modular development framework that first uses synthesis to identify the high level control flow of a program. If successful, the created control flow then is extended with concrete data transformations in order to be compiled into a final executable.

Our results also show that the current synthesis approaches cannot replace existing manual development work flows immediately. During the development of a reactive system, the developer still may use incomplete or faulty specifications at first, that need the be refined after a subsequent inspection. In the worst case, constraints are contradictory or miss important assumptions, which leads to unrealizable specifications. In both scenarios, the developer needs additional feedback from the synthesis engine to debug errors for finally improving the system specification. To this end, we explore two further possible improvements. On the one hand, we consider output sensitive synthesis metrics, which allow to synthesize simple and well structured solutions that help the developer to understand and verify the underlying behavior quickly. On the other hand, we consider the extension of delay, whose requirement is a frequent reason for unrealizability. With both methods at hand, we resolve the aforementioned problems and therefore help the developer in the development phase with the effective creation of a safe and correct reactive system.

# Zusammenfassung

Um reaktive Systeme zu regeln müssen Steuergeräte Zeit, Datenströme und Datentransformationen koordinieren, die durch den übergeordneten Kontrollfluss zusammengefasst werden. Die Aufgabe des Kontrollflusses ist es das System korrekt und dauerhaft zu betreiben. Die Entwicklung solcher Systeme wird dadurch zu einer Herausforderung, denn der Prozess ist fehleranfällig, zeitraubend, unintuitiv und kostspielig. Eine attraktive Alternative ist es stattdessen das System zu synthetisieren, wobei der Entwickler nur das gewünschte Verhalten des Systems festlegt. Der Syntheseapparat kümmert sich dann automatisch um alle technischen Details. Während aktuelle Algorithmen für die Synthese von reaktiven Systemen erfolgreich mit dem Kontrollanteil umgehen können, versagen sie jedoch, sobald komplexe Datentransformationen hinzukommen, aufgrund der Komplexität des vergleichsweise großen Datenraums. Daten und Kontrolle müssen demnach getrennt behandelt werden, um auch große Datenräumen effizient handhaben zu können.

Wir präsentieren *Temporal Stream Logic* (TSL), eine Logik die ausschließlich die Kontrolle einer Steuerung betrachtet, wohingegen Daten und funktionale Datentransformationen als austauschbare Blackboxen gehandhabt werden. In TSL ist es möglich Kontrollflusseigenschaften unabhängig von der Komplexität der zugrunde liegenden Daten zu beschreiben. Des Weiteren kann ein auf TSL beruhender Syntheseapparat die Realisierbarkeit einer Spezifikation prüfen, selbst ohne die konkreten Implementierungen der Datentransformationen zu kennen. Wir präsentieren ein modulares Grundgerüst für die Entwicklung. Es verwendet zunächst den Syntheseapparat um den übergeordneten Kontrollfluss zu erzeugen. Ist dies erfolgreich, so wird der resultierende Kontrollfluss um die konkreten Implementierungen der Datentransformationen erweitert und anschließend zu einer ausführbare Anwendung kompiliert.

Wir zeigen auch auf, dass bisherige Syntheseverfahren bereits existierende manuelle Entwicklungsprozesse noch nicht instantan ersetzen können. Im Verlauf der Entwicklung ist es auch weiterhin möglich, dass der Entwickler zunächst unvollständige oder fehlerhafte Spezifikationen erstellt, welche dann erst nach genauerer Betrachtung des synthetisierten Systems weiter verbessert werden können. Im schlimmsten Fall sind Anforderungen inkonsistent oder wichtige Annahmen über das Verhalten fehlen, was zu unrealisierbaren Spezifikationen führt. In beiden Fällen benötigt der Entwickler zusätzliche Rückmeldungen vom Syntheseapparat, um Fehler zu identifizieren und die Spezifikation schlussendlich zu verbessern. In diesem Zusammenhang untersuchen wir zwei mögliche Erweiterungen. Zum einen betrachten wir ausgabeabhängige Metriken, die es dem Entwickler erlauben einfache

und wohlstrukturierte Lösungen zu synthetisieren die verständlich sind und deren Verhalten einfach zu verifizieren ist. Zum anderen betrachten wir die Erweiterung um Verzögerungen, welche eine der Hauptursachen für Unrealisierbarkeit darstellen. Mit beiden Methoden beheben wir die jeweils zuvor genannten Probleme und helfen damit dem Entwickler während der Entwicklungsphase auch wirklich das reaktive System zu kreieren, dass er sich auch tatsächlich vorstellt.

# Acknowledgements

Over the past six years I met many people who shared their time working with me and supported me in writing this thesis in some or another way. Even if not especially listed here, I want to thank all of them.

Special thanks go to my supervisor *Bernd Finkbeiner*, who motivated me to work on all the fascinating topics, offered me great teaching experiences, always helped me with his outstanding expertise, and offered me a lot of different opportunities to broaden my perspectives.

Many thanks also go to *Martin Zimmermann* and *Swen Jacobs* who supported me during my first years with their expertise and gave me a great start for diving into many important topics.

I thank *Mark Santolucito* who always supported me during the lean times of our work, gave me a great experience with FRP, and tought me all the details about the American culture.

I thank *Ruzica Piskac* for introducing me to mark Mark and for teaching me all the social skills that are needed to survive in computer science.

I thank all members of the Reactive Systems Group for our joint talks, cakes, and sharing experiences. Especially, I thank *Leander Tentrup*, *Michael Gerke*, and *Norine Coenen* for tolerating me as their office mate. I thank *Christa Schäfer* for always helping me with all the administrative problems, I encountered regularly.

Finally, I thank my family for supporting me during all the time of my studies. However, the most thanks go to my significant other: *Melanie*, who supported me with all the non-scientific problems, helped me with managing my live and always motivated me to continue this work until the end.

# Contents

# Chapter I

# Introduction

The number of computational devices surrounding us today is exploding and has revolutionized the regular interaction with our everyday environment. While these devices allow us to precisely manage different tasks, most of them are already out of scope to any human assistant. Their importance covers a wide range of applications, such as mobile apps [133], embedded devices [61], robots [75], hardware circuits [15], GUIs [33], and interactive multimedia [126]. Especially safety critical systems, such as vehicles, aircrafts or large industrial plants heavily rely on a safe and secure control [125, 82], but also common devices, like mobile phones, are required to work reliable and continuously over long term periods [133]. As a consequence, the correct and secure design of these systems has turned into one of the major challenges for humanity in the $21^{\text{th}}$ century.

Regarding the current development, more and more of these devices are of a reactive nature. For their permanent operation, it does not suffice any longer, if they only take a single input, which then is turned into a single output. Instead, they need to continuously interact with their environment to serve users at any time and under all possible circumstances, yielding an infinitely ongoing input-to-output behavior. Unfortunately, especially the development of these so called *reactive systems* turns out to be much harder than creating their classical terminating counterparts [58]. The combination of repetitive tasks, irregular user request, and different "modus operandi" induces a vast rise in complexity, which cannot be handled by traditional algorithms and approaches.

While research on classical algorithmic transformations has revealed many strong insights leading to a fundamental understanding of the underlying problems, the design of reactive systems still challenges people and companies all around the world [58, 87, 125]. For mastering these challenges, we need better fundamental insights that enable the prevention the deceitful traps along the development process, while at the same time being expressive enough to cover all of the users' requirements. To this end, developers need new tools that utilize these models effectively and provide insights up to any required amount of precision.

One of the first major advances towards this desire, is the principle of *verification*. Already the roman politician and lawyer Cicero stated:

*"Cuiusvis hominis est errare, nullius nisi insipientis in errore perseverare."*

`Anyone can err, but only the fool persists in his fault.`

Cicero teaches us that as long as a human developer is involved we always have to expect that there will be errors that are made. Verification is a mechanism to countervail this behavior by providing methods and tools to identify and understand these errors by revealing them to the user.

Under the scope of reactive systems, the general methodology behind verification can be summarized as follows: Given an existing implementation of a reactive system, we first extract a simplified model that (ideally) behaves exactly the same as the original, but abstracts from unnecessary technical details that do not influence the generated behavior. At the same time, all desired properties of the input/output behavior are formalized using a suitable specification formalism, such as a temporal logic. Applying the established theoretical machinery, we then verify that every possible execution following an arbitrary sequence of uncontrollable inputs from the environment satisfies the behavioral specification.

A rough overview over the verification process is depicted in Figure 1. First, a simplified model is abstracted from the manually created system implementation, which then is transformed into an automaton. On the other hand, the specification is translated to an automaton as well, which then is complemented to cover all violating behavior. Both steps may involve intermediate translations via other automata models or specification logics, depending on the exact specification formalism used. Finally, the intersection between both automata is built. If the language of the resulting cross-product automaton is empty, then there is no path violating the property and, thus, the property is satisfied. If the language is non-empty, the existing member provides a counter-example for the violation of the property. A slight modification of the verification process is also known as *model checking*, where we elide the task of first creating a system abstraction and instead directly start from the abstracted system model.

If verification succeeds, then it is formally proven that the implementation satisfies the formalized requirement and, thus, is *error free*. Otherwise, an input sequence is revealed that violates one or multiple of the given requirements, which is assumed to be violated as well, when executed on the original implementation. If this is case, then the verification tool has found a bug and the developers need to revise the created implementation. If the bug cannot occur in reality, then their is a mismatch with the formalized

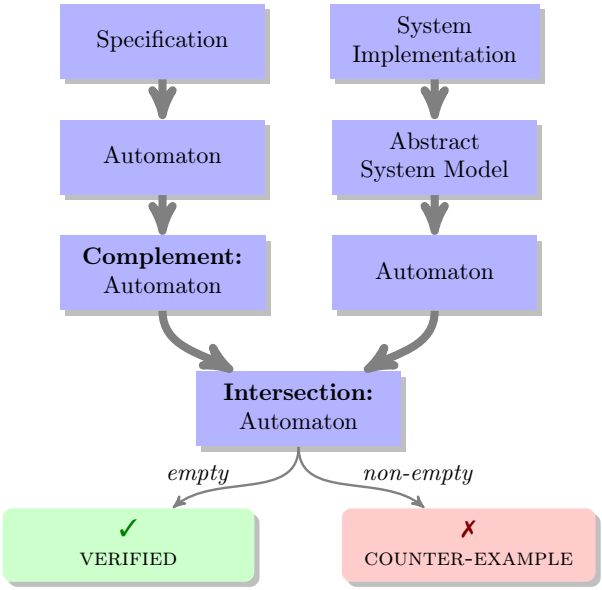



Figure 1: The verification process.

requirements and the underlying system model demanding a refinement of the system abstraction. Verification already has proven it's success in industry [18, 24, 54, 60] and has been explored extensively, both in terms of theory [8, 29, 37, 38, 86, 116, 119] and practice with respect to scientific tools [10, 55, 59, 63].

Verification gives us a mighty tool to encounter development failures. However, despite its obvious advantages, it also introduces an additional burden. Instead of only manually creating the requested system by hand, developers now also are required to create a specification, using a suitable formalism to be understood by a machine and a matching system abstraction. Both steps consume additional time and produce additional costs, which may not countervail simpler methods, such as testing or manual inspection. Likewise, verification does not free us from the task of creating the system under consideration in the first place. We still need programmers and engineers for creating it. Correspondingly, technical issues and other cumbersome considerations remain to be resolved through manual interaction in the end.

Motivated by these drawbacks, there are some more extensive thoughts: May it be possible to *automatically* create the system under consideration? Just from the given specification?

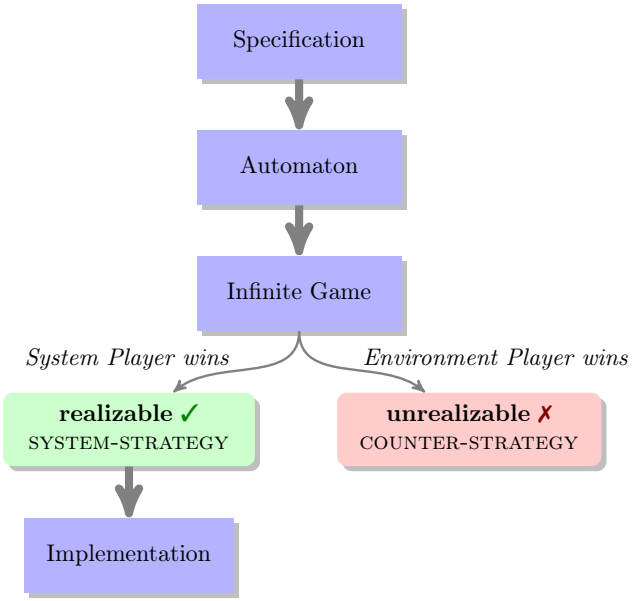



Figure 2: Overview over the synthesis process.

The question is known as the *synthesis problem* and was first formulated for reactive systems by Alonzo Church [48] in 1963. Synthesis elides the tedious task of manually creating a system by hand and instead asks for automatically creating it from a behavioral specification. Similar to the process of verification, as depicted in Figure 1, synthesis first translates the specification into an automaton. However, for synthesis, the inputs from the environment are not determined by a single property that needs to be violated. Instead, the synthesized system must satisfy the specification against every possible input behavior. Correspondingly, the problem does not reduce to an emptyness-check of an automaton, but leads to an infinite game, played between the system and its environment. With respect to this game, the specification then is realizable if and only if the system player wins against every possible behavior of the environment. The final system implementation is then given by the winning strategy of the system player in this game. An overview of the synthesis process is illustrated in Figure 2.

While the synthesis approach itself sounds promising, unsurprisingly, it also comes at a price: in general, the synthesis of reactive systems is computationally much harder than their verification. In a nutshell, it is required to

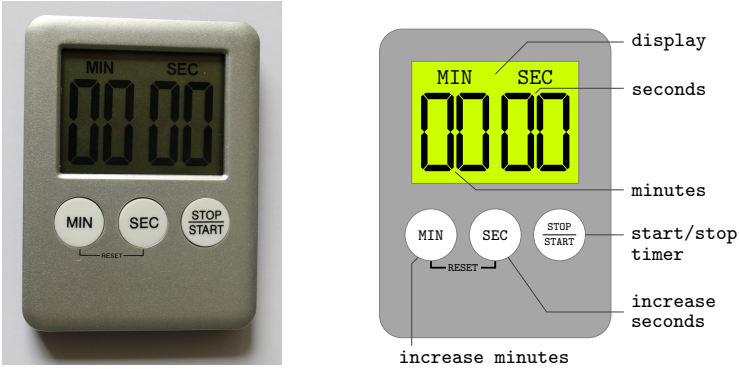



Figure 3: A kitchen timer, whose real implementation is depicted on the left, while it's main interfaces are explained on the right.

unfold the strategy tree along every possible sequence of inputs, resulting in a much higher complexity of the synthesis algorithm. Synthesis mostly received attention from the theoretical perspective for a long time. However, recent improvements of computational devices and new algorithmic solutions have revived the practical relevance of synthesis again.

Many achievements in the area of verification can be re-utilized for synthesis, like for example the extensive automata theoretical backend. Therefore, it is not very astonishing that recent advances on the underlying automata theory, originally developed for the purpose of verification, also can be leveraged for synthesis. What comes at a surprise, however, is that there still are no big success stories, which show that synthesis approaches are ready for real-world development as well. While there are *some* success stories, such as the synthesis of the AMBA AHB bus arbiter, an open industrial standard for the on-chip communication and management of functional blocks in system-on-a-chip (SoC) designs or the IBM generalized buffer [14], they still are limited to small hardware examples that cannot compete with complex system designs, as given by most the applications considered in practice today.

This is especially surprising, since for verification such success stories do exist [18, 30, 115, 135, 78], raising the question: Why is this the case? Especially, as both techniques utilize the same automata theoretic backends. In order to find an answer to this question consider the following experiment, which reveals an important, but missing principle for the practical application of synthesis. We consider a compact reactive application that everybody already has been used before. It is a simple kitchen timer device, which allows

to set up a timer that keeps track of time-dependent processes in the kitchen. It consists of three buttons for setting a time and starting or stopping a timer, a screen to display the remaining time, and a buzzer to notify the user if the time is over. Figure 3 shows a real-world implementation of such a device and it's components, together with a short overview of its functionality.

Our goal will be to synthesize an implementation for the device from a behavior specification only, instead of creating it manually by hand. If it is indeed legal to assume that current synthesis frameworks can handle such an application, then the corresponding design and synthesis process should be easy. Therefore, the behavior of the timer is fixed according to the following list of requirements, where we used the real-world application of Figure 3 for obtaining a reference behavior. We do not go into detail for all of the expressed properties. Nevertheless, we still provide the complete list to give an intuition of to process of creating reactive system specifications.

1. *Whenever the* `MIN` *and* `SEC` *buttons are pressed simultaneously, the timer is reset, meaning the time is set to zero and the system stays idle until the next button gets pressed.*

2. *If only the* `SEC` *button is pressed and the timer is not currently counting up or down, then the currently set time is increased by one second.*

3. *If only the* `MIN` *button is pressed and the timer is not currently counting up or down, then the currently set time is increased by one minute.*

4. *As long as no time greater than zero has been set and the system is idle: if* `START/STOP` *gets pressed and the timer is not already counting up or down, then it starts counting up until it is stopped by any button pressed.*

5. *If a time has been set and the* `START/STOP` *button is pressed while the timer is not currently counting up or down, then the timer starts counting down until it is stopped by any button pressed.*

6. *The timer can only be started by pressing start.*

7. *The timer can always be stopped by pressing any button while counting up or down.*

8. *It is possible to start the timer and to set some time simultaneously.*

9. *The buzzer beeps on any button press and after the counter reaches zero while counting down.*

10. *The display always shows the time currently set.*

First, we collect the inputs and outputs of the system, as given by its physical interface. The application consists of three binary inputs, one for each button, and 29 binary outputs, one for the buzzer, controlled via a square wave of varying frequency, and 28 to display the four digits, each consisting of a seven-segment display. In order to bring the aforementioned informal requirements into a machine readable format, we also need a formalism for specifying the input/output behavior. The most popular formalism for such tasks is Linear-time Temporal Logic (LTL) [116]. The logic combines Boolean and temporal reasoning, which is everything that we need for specifying the aforementioned properties. In LTL, inputs and output of the system are handled as atomic proposition, *i.e.*, as named literals that evaluate at each point in time either to true or false. LTL is also used in the annual synthesis competition SYNTCOMP [70, 68, 71]. Hence, it not only is supported by many existing and efficient synthesis tools [105, 41], but also represents an accepted standard for the evaluation of our considerations, as selected by the research community. With LTL at hand, we are ready to specify the first property:

*Whenever the* `MIN` *and* `SEC` *buttons are pressed simultaneously the timer is reset, meaning the time is set to zero and the system stays idle until the next button gets pressed.*

In LTL the property of `Min` and `SEC` being pressed simultaneously is expressed by $i_{min} \wedge i_{sec}$, which states that the input atomic propositions $i_{min}$ and $i_{sec}$ must both evaluate to true at the current point in time. The reset to zero is observed at the output by every digit displaying a zero. Let $o_d^s$ be the output atomic propositions for each digit $0 \leq d < 4$ and segment element $0 \leq s < 7$. Then the property of all digits displaying zero is specified by:

$$\varphi_{\text{ZERO}} := \bigwedge_{0 \leq d < 4} \left( \neg o_d^3 \wedge \bigwedge_{\substack{0 \leq s < 7 \\ s \neq 3}} o_d^s \right)$$

Furthermore, the system being idle is specified by stating that the output does not change from the current to the next execution in time:

$$\varphi_{\text{IDLE}} := \bigwedge_{\substack{0 \leq d < 4 \\ 0 \leq s < 7}} \left( o_d^s \leftrightarrow \bigcirc o_d^s \right)$$

The temporal $\bigcirc$-operator allows us to move one time step into the future. The final specification of the first property then is given by:

$$\square\Big(i_{\text{MIN}} \wedge i_{\text{SEC}} \rightarrow \varphi_{\text{ZERO}} \wedge \big( \bigvee_{b \in \{\text{MIN, SEC, STASTO}\}} (\neg i_b \wedge \bigcirc i_b)\big)\, \mathcal{R}\, \varphi_{\text{IDLE}}\Big)$$

The temporal operator $\square$ denotes that the respective sub-property must be satisfied in every time step, while the temporal operator $\mathcal{R}$ states that the sub-property $\varphi_{\text{IDLE}}$ must be satisfied as long as it is not released by a button being pressed, which might potentially never be the case. Note that the "*gets pressed*" property is formalized by a button input changing from the not being pressed state to being pressed. This completes the specification of the first property. We conclude that it is indeed possible to specify the stated behavior with LTL and are motivated to continue with the second property:

*If only the* `SEC` *button is pressed and the timer is not currently counting up or down, then the currently set time is increased by one second.*

We immediately recognize that the expressed behavior covers multiple situations to be reflected within the specification. First, consider that the one second increase produces many different displayed values depending on the currently displayed time. In particular, it implies that we need to consider every possible of these situations using a big case distinction, since LTL can only relate Boolean inputs with Boolean outputs. At this point, we leave it to the reader to calculate the exact number of these cases and to write them all down. We only note to take care that also all cases are covered, for which the minutes need to be increased as well due to an overflow.

The mindful reader, however, already has recognized that this turns out to be tedious task, which does not even countervail the work to be spent into a manual implementation. However, even worse, even if we are patient and manage to finish the task eventually, then we probably still will not get a solution, since the utilized synthesis tool most likely cannot handle the Boolean complexity that we have introduced with the encoding.

Exactly this restriction will face us regularly, introducing a fundamental challenge for current synthesis approaches. Being restricted to the Boolean data domain, we always need to break down complex data to this level at first. The result is an additional overhead for both: the developer, required to break down complex data to the Boolean level, and the synthesis tool, required to capture the Boolean representation to bring it back into context of the overall behavior. This clearly introduces unnecessary overhead, which we should try to avoid instead.

But how to circumvent the problem? We previously stated that verification is successfully used in practice. Hence, isn't there any existing method

already used in verification that is able to help us with synthesis as well? If we only need to verify the second property against an existing implementation, then we probably would formalize it differently:

$$\neg i_{\text{SEC}} \wedge \bigcirc i_{\text{SEC}} \wedge \neg\big(\neg i_{\text{MIN}} \wedge \bigcirc i_{\text{MIN}} \vee \neg i_{\text{STASTO}} \wedge \bigcirc i_{\text{STARTSTOP}}\big) \wedge \bigcirc \neg o_{\text{COUNTUPDOWN}} \rightarrow \bigcirc\big(time_{t+1} \equiv time_t + 1\big)$$

This formalization first checks that only the `SEC` button is pressed, using a simple Boolean combination over the input buttons. Then, it checks whether the counter is currently counting up or down. To this end, we assume that all regions of the verified code are labeled through an additional output, highlighting whether the counter is currently counting up/down or not. Adding such an additional output is easy for the programmer or, depending on the programming framework, can even be done automatically. Furthermore, we need to verify whether the time was increased by a second, where we use the same trick: we label all regions of the code, where time gets increased by a second, and then provide this labeling as an additional output. Thereafter, we reduced the verification of a huge number of possible combinations to checking only a few additionally produced outputs, and, thus, avoid the corresponding state space explosion problem. We already illustrated this process in Figure 1 through the initial reduction from the actual system implementation to an abstract model that only covers the relevant information.

Hence, why do we not apply the same idea for synthesis too? The reason is simple. We do not have an existing implementation for marking the regions, since creating this implementation is the synthesis task to be executed in the first place. Can this circumstance be resolved in order to synthesize our kitchen timer? Or does synthesis indeed force us to encode all possible combinations of increasing time by a second in the Boolean data domain? Is this Boolean overhead an unavoidable burden, which synthesis must handle in order to automatically create an implementation from a specification? Or does there exist a better solution?

In this thesis, we show that there indeed is a counterpart to the intermediate abstraction step, as leveraged for verification, that can be utilized for synthesis as well. Remember that for verification we rely on the circumstance that there exists an additional abstraction that hides unnecessary details from the specification, while for synthesis, such an abstraction does not exist. For resolving this mismatch, our solution is to use a universally quantified abstraction. Hence, instead of relying on a specific abstraction that is provided by the user, we utilize a symbolic representation such that the synthesized implementation must be correct for any possible matching instantiation.

# 1 Temporal Stream Logic

Our solution is Temporal Stream Logic (TSL), a specification language that leverages a clean separation of the implementation's search space into data and control. This provides many advantages. It allows developers to focus on the reactive control, while complex data and their transformations are hidden behind the universal abstraction. A similar concept is also used in programming languages, where programmers hide the details of a complex routine behind a procedure or a function call. Therefore, it is not necessary to consider all of the inner workings of every written procedure in order to assemble the final program, but only to have an understanding of every procedure's effect or their behavior. The same principle is utilized for synthesis with TSL.

The immediate advantage of a universal abstraction is that developers do not need to have a real implementation of the data entities and their transformations at hand, since TSL synthesis guarantees a correct behavior, independently of the finally used implementations. This is especially important for initial design phases, where developers need to evaluate different concepts and interaction possibilities, even before they want to consider every detail of how data structures and the conversations between them need to be represented.

TSL uses the same temporal and Boolean operators as LTL, but atomic propositions are exchanged with a more expressive formalism. In TSL, inputs and outputs are data streams of arbitrary type. To argue about values of input streams, universally quantified pure predicates are used. For example, the predicate `p i` checks for the property `p` on the input stream `i`. The predicates reduce properties of the data to Boolean decisions, used to guide the control flow of the reactive application. However, for the production of outputs TSL relies on a different approach. The logic not immediately argues about values or their properties, as they are produced by the system, but instead about how and from which sources they are produced. This is in contrast to LTL, where the output values themselves (and, thus, also their properties) are related to the inputs. TSL instead uses the notion of *updates*, like $[\,\mathtt{o} \leftarrowtail \mathtt{f\ i}\,]$, expressing that a pure function `f` is applied to an input `i` and the result is piped to the output `o`. In combination with temporal and Boolean operators TSL therefore is expressive enough to specify the control behavior of reactive systems.

Another feature of TSL is that it leverages the principle of *purity*. Purity ensures that values, as returned by functions and predicates, only depend on the corresponding argument inputs, *e.g.*, if a function `f` is applied to the input `i` and the result is once passed to $\mathtt{o}_1$ ($[\,\mathtt{o}_1 \leftarrowtail \mathtt{f\ i}\,]$) and once to $\mathtt{o}_2$ ($[\,\mathtt{o}_2 \leftarrowtail \mathtt{f\ i}\,]$), then both resulting values, that are passed to $\mathtt{o}_1$ and $\mathtt{o}_2$, are

the same. In other words, a function `f` is not allowed to maintain internal state and must be side effect free. Purity is a powerful principle, with its origins in functional programming languages, which allows to consider function and predicate implementations by their equivalent mathematical counterparts. We use purity in TSL to tame the expressive power of the universal abstraction. If a function or predicate would not be required to be pure, then it could produce different values with every possible application, even if the inputs stay the same, eliminating the idea of using different symbolic representations for different functions and predicates at all. Instead, any application could produce any result at every possible point in time.

Utilizing purity, however, comes at a price: the implicitly induced repeatability of application results makes the synthesis problem undecidable. This is in contrast to LTL, where the synthesis problem is 2ExpTime-complete [117]. Hence, we trade theoretical decidability against practical scalability, which is an acceptable trade-off with respect to the advantage of being able to synthesize realistic systems that finally can be instantiated to practical applications.

To deal with the undecidability result, we present a method for synthesizing TSL that combines Bounded Synthesis [45] with counterexample-guided abstraction refinement (CEGAR) [31]. The method approximates the TSL synthesis problem via LTL queries that allow the environment player to relax the general purity assumption. Our approximation is sound, in the sense that if the weaker LTL specification is realizable, then the encoded TSL specification is realizable as well. However, the reduction is not complete. Thus, it may be that LTL synthesis returns an unrealizability result, even if the specification is realizable. An overview of the respective design process is given in Figure 4. The approximation allows for inconsistencies, since loosing the semantic meaning of predicates and functions in TSL enables the environment player to evaluate the same predicate differently on the same system input when evaluated at different points in time. To fix this, we first analyze the returned counter-strategy in case of unrealizability, for whether it violates purity in order to win in the created LTL approximation. In this case, we call the counter strategy *spurious*. The respective analysis requires an upper bound $n$ on the size of the counter strategy, which is provided using the bounded synthesis approach. Finally, to countervail the spurious behavior, we refine the approximated LTL specification by adding additional assumptions that forbid the spurious behavior and re-execute the LTL synthesis tool afterwards. The CEGAR loop is continued until either a non-spurious counter strategy is found or the specification gets realizable.

If TSL synthesis is successful, then a control flow model (CFM) is returned that implements the control of the specified reactive system. By utilizing LTL

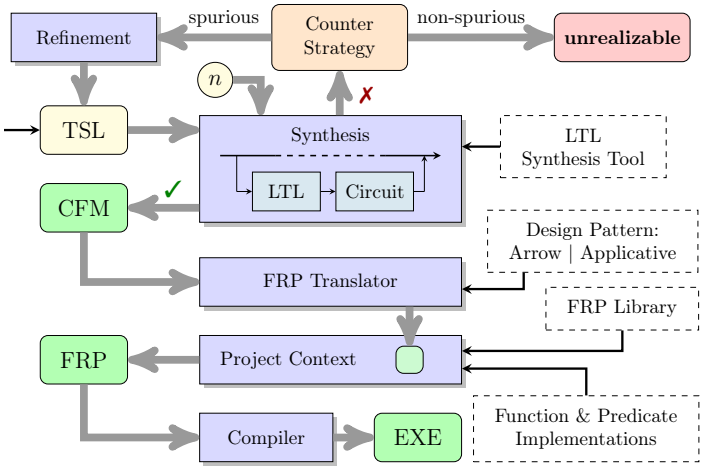



Figure 4: TSL synthesis uses a modular design. Each step takes input from the previous step as well as interchangeable modules (dashed boxes).

synthesis as a backend, the control part of the program is still returned as a Boolean structure, such as a circuit or Mealy machine. This Boolean structure then selects the correct function transformations at the right points in time according to the specification. The CFM, however, does not instantiate the symbolic representation of functions and predicates with concrete implementations yet. Instead, they are provided by the developer afterwards, using implementations that are coded in their preferred functional programming languages or using external libraries and API calls. In order to implement a CFM as part of a larger functional programming context we utilize Functional Reactive Programming (FRP). FRP is a programming paradigm that uses stream processing components, which are connected as part of a control flow network. FRP provides the ideal framework for describing programs that handle infinite data streams of arbitrary and polymorphic type. There are different design patterns used to realize FRP, such as Applicative FRP, Monadic FRP, and Arrowized FRP. Our synthesis engine is compatible with all of them, but depending on the concrete design pattern used, different code needs to be generated from the CFM. Hence, after synthesis, the designer selects his or her favorite FRP library, the corresponding design pattern and implementations for all function and predicate terms. Our framework then puts everything together and produces code that can be compiled to an executable with the corresponding FRP compiler.

```
1  COUNTUP    = [ time <- countup time dt ];
2  COUNTDOWN  = [ time <- countdown time dt ];
3  INCMIN     = [ time <- incMinutes time ];
4  INCSEC     = [ time <- incSeconds time ];
5  IDLE       = [ time <- time ];
6
7  ZERO         npxs= eq time zero();
8  RESET        = Min && Sec;
9  COUNTING     = COUNTUP || COUNTDOWN;
10 ANYKEY       = press Min || press Sec || press StartStop;
11 START        = press StartStop && !press Min
12                  && !press Sec;
13 STARTANDMIN  = press StartStop && press Min
14                  && X !Sec && X X !Sec;
15 STARTANDSEC  = press StartStop && press Sec
16                  && X !Min && X X !Min;
17
18 xor x y          = !(x <-> y);
19 press x          = !x && X x;
20 tillAnyInput x = (x && !ANYKEY) W (RESET || x && ANYKEY);
21
22 initially guarantee {
23   !COUNTING && (X COUNTING -> START);
24   !INCSEC && !INCMIN;
25   [ beep <- false ];
26 }
27
28 always guarantee {
29   RESET <-> [ time <- zero() ];
30
31   !COUNTING && press Sec && X !Min <-> X INCSEC;
32
33   !COUNTING && press Min && X !Sec <-> X INCMIN;
34
35   ZERO -> ((IDLE && START -> X tillAnyInput COUNTUP)
36              W (INCMIN || INCSEC));
37
38   INCMIN || INCSEC
39     -> ((!COUNTING && START -> X tillAnyInput COUNTDOWN) W X ZERO);
40
41   X !COUNTING && X X COUNTING
42     -> X START || STARTANDMIN || STARTANDSEC;
43
44   COUNTING && ANYKEY && X !RESET -> X tillAnyInput IDLE;
45
46   !COUNTING && (STARTANDMIN || STARTANDSEC)
47     -> X X tillAnyInput COUNTDOWN;
48
49   X (COUNTDOWN && ZERO) || ANYKEY <-> X [ beep <- true ];
50   xor [ beep <- true ] [ beep <- false ];
51
52   [ dsp <- display time ];
53 }
```

! ($\neg$) negation && ($\wedge$) conjunction || ($\vee$) disjunction X ($\bigcirc$) next
-> ($\rightarrow$) implication <-> ($\leftrightarrow$) equivalence W ($\mathcal{W}$) weak until

Figure 5: The full kitchen timer specification.

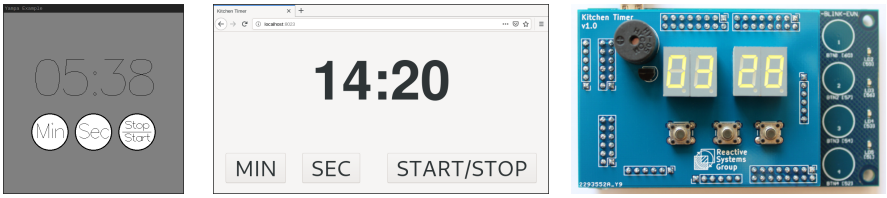



Figure 6: Timer applications: the left picture shows the desktop application built with the `Yampa` FRP library, the center picture shows a web version built with the `Threepenny-GUI` library, and the right picture shows the hardware version built with the functional hardware description language `ClaSH`. All applications have been synthesized from the same specification of Figure 5.

Our experiments on the design and synthesis of reactive systems from TSL specifications reveal extremely positive results. We successfully synthesized systems like the kitchen timer, a music player app and a controller for autonomous vehicles in the Open Race Car Simulator (TORCS) [44]. In general, the systems we designed with TSL range from classic reactive synthesis problems, like escalator control, through programming exercises from functional reactive programming, to novel case studies like an interactive arcade space shooter implemented on an FPGA [50]. In contrast to manually written programs, our specifications only cover the control flow behavior, which is easy to read and easy to extend. In contrast to classical specification logics, the specifications are close to the natural language descriptions, easy to understand, and do not need to encode complex data transformations by using unreadable Boolean encodings.

These propositions are underlined by the full specification of the kitchen timer, given in our textual TSL specification format, in Figure 5. The kitchen timer's control is directly synthesized from this textual representation, resulting in a satisfying CFM. By utilizing FRP, we can instantiate the created CFM using different FRP libraries that embed the application in different environments. For example, we created a desktop application, using the Arrowized FRP library `Yampa`, a web application, using the Monadic FRP library `Threepenny-GUI`, and a hardware version, using the Applicative hardware description language `ClaSH`. The desktop version is a standalone application to be run on your personal computer, while the web version can be accessed over the network using a browser. The hardware version, on the other hand, is implemented on an FPGA that is connected to a physical display, real hardware buttons and a buzzer. The resulting applications are depicted in Figure 6.

TSL synthesis enables the successful creation of reactive applications from temporal logic specifications. The logic leverages a clean separation between data and control such that the synthesis engine can focus on the control, while data instances and their transformations are postponed to be concertized afterwards. A full discussion of the semantics of TSL, the CEGAR based synthesis procedure, and the connection to FRP and real world applications are presented in Chapter III.

# 2 Output Sensitive Synthesis

With the ability to separate the representation of data from the actual control, we made a big step forward towards enabling synthesis as a design method for the creation of reactive systems. Towards our goal of reaching a fully synthesis enabled work flow there are, however, still some more obstacles to overcome. Instead of programming a system by hand, developers now have to create specifications. Although they offer obvious advantages against manually created programs, they also introduce some new challenges. One major challenge is the transition from a deterministic controller description, as implicitly induced by a program, to a broader solution space, of which all instances satisfy the specification. While in theory, the synthesizer can freely choose of any of theses solutions, in practice system designers still may want the possibility to choose the solution that best fits their needs. To this end, we need more than just a single push button technology. Instead we need output sensitive methods that allow to impose additional quality requirements on the result, thus, enabling developers to *choose* among all of the possible satisfying solutions.

One of the most well known output sensitive methods is *Bounded Synthesis* [45], where the solution - usually represented as a Mealy machine - is bounded in the number of Mealy states. Hence, beside the specification itself, the developer additionally provides a positive integer bound that limits the overall search space. Therefore, the developer now has an additional metric that for example allows to favor smaller solutions against larger ones. Nevertheless, just restricting the size of the final solution may not be enough. A Mealy machine - from a simple perspective being nothing else than an edge labeled, directed graph - still allows for more degrees of freedom. For example, in the way of how the edges are connected to vertices. Even with the number of states of the solution being fixed, the synthesized result still may be unnecessary complex, and, thus, should be avoided to be chosen by the developer.

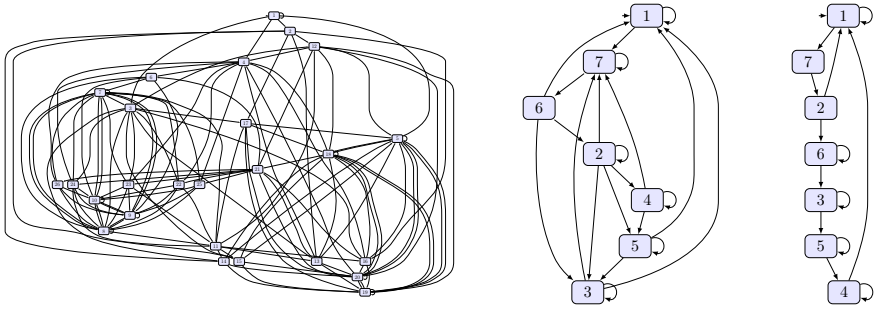



Figure 7: Three implementations of the TBURST4 component of the AMBA bus controller. Standard synthesis with Acacia+ produces the state graph on the left with 14 states and 61 cycles. Bounded synthesis produces the graph in the middle with 7 states and 19 cycles. The graph on the right, produced by bounded cycle synthesis, has 7 states and 7 cycles, which is the minimum.

For an example consider the three Mealy machines depicted in Figure 7. All of them have been synthesized from the same specification, describing the TBURST4 component of the AMBA AHB bus arbiter [14]. The first solution on the left results from a standard game based synthesis approach, as implemented by the Acacia+ tool [17]. The second solution in the middle uses Bounded Synthesis, where the bound has been chosen such that it matches the minimal number of required states of any realizable solution. Finally, the solution on the right also matches a bound on the number of simple cycles that are part of the solution graph. A *simple cycle* is a path through the graph that starts and ends in the same vertex and visits every vertex in between at most once. Such a cycle can be seen as a single instance of the observable infinite behavior, where in general, the overall behavior is composed of switching between multiple such instances. Our intuition tells us, that if a solution contains less simple cycles, then it also has a simpler structure with respect to the solution graph. It turns out that this intuition is indeed confirmed by the theory. Our analysis shows that the number of simple cycles of a solution can be a highly explosive factor. Hence, bounding it to a minimum leads to much *simpler* solutions. Regarding the example of Figure 7 this simplicity is clearly visible. In most cases, a designer would prefer choosing the leftmost solution in favor of the two other ones.

Our algorithm for bounding the number of cycles is based on Tiernan's cycle counting algorithm for directed graphs [138]. The algorithm executes an exhaustive search, while keeping track of the vertices that already have

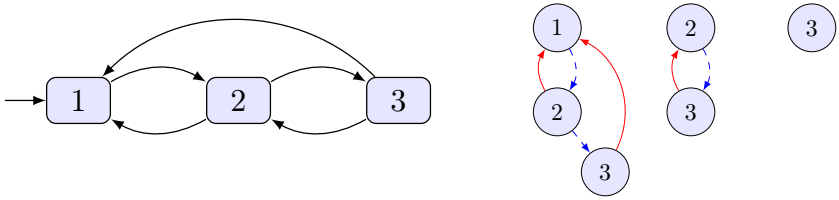



Figure 8: The witness trees for an example state graph with three simple cycles. The state graph is shown on the left. The first graph on the right proves that vertex 1 appears on two cycles (via vertex 2 and vertices 2 and 3). The second graph proves that vertex 2 is on a cycle not containing vertex 1 (via vertex 3) and that there are no more cycles through vertex 3.

been visited. To this end, the graph is unfolded to a tree from some arbitrary root vertex $v$ such that no vertices repeat on every branch. The number of vertices in the tree, leading back to $v$, then is equal to the number of simple cycles containing $v$. Next, the vertex $v$ gets removed from the graph and the algorithm recursively determines the number of cycles that do not contain $v$ on the remaining sub-graph. The overall number of cycles is equal to the sum of both results. The algorithm is linear in the number of cycles of the graph.

Our synthesis procedure combines this algorithm with Bounded Synthesis by using a SAT-solver to simultaneously constrain the number of vertices of the encoded Mealy machine, as well as the number of simple cycles. To this end, we first *guess* three witnesses: the realizing Mealy machine, their cross-product with the specification automaton, and a ranking function that bounds the number of rejecting states visited by the cross-product. These witnesses and their encoding are taken according to the standard bounded synthesis [45] approach. On the other hand, we guess a forest of trees that covers the execution of Tiernan's algorithm and link it to the graph shape that is implied by the Mealy machine. Putting a bound on the number of edges in the witness trees that lead back to the corresponding roots finally completes our encoding.

An example for the guessed witnesses of the bounded cycle encoding is depicted in Figure 8. The left graph depicts the shape of the state graph of a given Mealy machine, while the right graphs represent the resulting witness trees. The first graph witnesses that vertex 1 appears on two cycles (via vertex 2 and vertices 2 and 3). Similarly, the second graph witnesses that vertex 2 is on one more cycle (via vertex 3). Overall, there are no more than three simple cycles in the graph, as witnessed by the last cycle-free witness tree.

Our experiments reveal that *Bounded Cycle Synthesis* performs similar in terms of computational overhead as the standard Bounded Synthesis approach. The overhead of imposing an additional bound on the number of cycles is measured to increase just linear with the given bound. While Bounded Cycle Synthesis clearly cannot handle as many specifications as game based synthesis techniques, our results still show that the structural quality of our solutions is much better. In that sense, our approach is of interest for the synthesis based development of reactive systems, because it enables the synthesis of optimized solutions for decomposed or partial specifications that can be joint to an overall satisfying implementation after they have been synthesized.

Output sensitive synthesis methods extensively explore the solution space of correct solutions, while at the same time introducing selectable quality metrics to the process. Such flexibility is especially desired for the development of reactive applications, since developers acquire the possibility to pick those solutions that best fit their needs [87]. To this end, different strategy representation models have been considered [19]. Indeed, there is no limitation towards rudimentary models, like Mealy machines or basic graphs. More compact representations can be utilized as well. In practice, developers prefer models that can be executed efficiently on the target execution platforms and fit the underlying programming contexts.

Motivated by these observations, we also explore synthesis methods for some of these *more compact* models in their function as potential targets for output sensitive synthesis. Beside bounding the number of simple cycles and the states of the underlying Mealy machine, we consider *Boolean circuits*, where explicit bounds on the number of latches and gates are provided by the user. Furthermore, we explore the efficiency of synthesizing *Boolean programs*, which use standard programming language elements like while loops and conditionals. Our model is inspired by Madhusudan, who used an automata theoretic approach extending standard unbounded synthesis towards synthesizing reactive programs [100]. Finally, we also consider a model for *register programs*, which instead build on more assembler like instructions language that works on registers and with jump instructions. Our results show, that Boolean circuits are best suited for most of today's application cases. Especially, since they provide very natural metrics by the amount of latches and gates that are used. Furthermore, due to their easily parallelizable execution semantics, they best fit with current FRP approaches, since they allow to consider the circuit structure as part of a stream processing network. More details on the encodings that are used for these output sensitive synthesis approaches and their experimental evaluation are presented in Chapter IV.

# 3 Delay Games

If synthesis is successful, then the synthesizer returns a solution that satisfies the specification and can be further processed towards an executable application. However, if synthesis is not successful, then there must be a mismatch among the provided requirements that cannot be resolved by the synthesizer in order to create a satisfying reactive system. In such a case, it is of highly importance that developers are able to identify the problem for refining the specification and taking care of the initial misunderstanding. More concrete, developers need feedback for debugging unrealizability results.

Helping developers in resolving potential issues requires the identification of the error's cause within the specification at first. For reactive system specifications this cause can be analyzed in two stages. Unrealizability either is caused by an inconsistency among the system requirements or by the environment producing a sequence of inputs that results in invalid behavior, no matter how the system responds. In the first case, the specification is *unsatisfiable*, since the inconsistency remains present independently of how the environment behaves. In the second case, however, the specification is assumed to be satisfiable, *i.e.*, there is at least one input sequence for which a satisfying output assignment exists, but it is still rendered *unrealizable* by the environment, *i.e.*, there exits a counter-strategy that beats every possible system implementation.

A first notion for explanations of unrealizability has been considered by Cimatti, Roveri, Schuppan, and Tchaltsev [28], where they identified minimal unsatisfiable cores for the *Generalized Reactivity(1) fragment* of LTL (GR(1)) [16] based on activation variables. The work was later extended towards full LTL by Schuppan, discussing challenges of suitable notions for unsatisfiable and unrealizable cores [129]. Regarding unsatisfiability on its own, Schuppan further analyzed the possibility of extending the unrealizable core with temporal information about the cause of the problem using information on temporal relevance [130] and temporal resolution graphs [131]. Nevertheless, while formal notions of unsatisfiable and unrealizable cores provide locatable targets for corresponding algorithms, the resulting cores still may be hard to understand for developers. The corresponding problem was studied explicitly in the area of autonomous robots [94, 121, 122].

Eliding the task of *explaining* the issue to the user can be achieved by automatically repairing the specification instead. Therefore, existing approaches usually assume that the problem is caused by the environment, *i.e.*, the specification is satisfiable, but not realizable. In this case, the general idea is to weaken the environment through the extension of the specifica-

tion by environment assumptions, which forbid *problematic* inputs. To this end, corresponding assumptions should be extracted automatically from the counter-strategy for the environment witnessing the unrealizability result.

A first realization of specification repair was presented by Chatterjee et al., who extracted minimal assumptions from the underlying game graph [25]. It is, however, not guaranteed that especially those assumptions are selected that are best suited for achieving realizability and can be expressed by simple LTL formulas. Improvements towards these requirements have been provided by template-based classifications, which guide the selection of the assumptions [92]. Furthermore, output sensitive metrics, like the length of the longest path leading to an unsafe state, have been considered [27]. Another consideration for the identification of problem causes, used to guide the specification repair, leverages the notion of *strong satisfiabiliy* [49], which requires that the specification is satisfiable and that there exists at least one satisfiable output sequence for every possible input. Note that is in contrast to a realization via a uniform system strategy [107]. Thus, strong satisfiability can be seen as an intermediate notion placed between satisfiability and realizability. Finally, the application of a CEGAR-based approach for a multi-stage refinement of environment assumption has been considered for GR(1) [2].

Due to the infeasible scalability of the resulting algorithms and high complexity classes, incomplete and approximating methods have been studied as well. For the GR(1) fragment, approximations of possible winning regions in the underlying game graph [83] and countertrace-based heuristics [84] have been used. For full LTL, the problem also has been studied under the assumption of bounded environments [34, 88]. Finally, for cyber-physical systems, like robots acting in a physical environment, online approaches have been applied. For example, run-time monitoring of environment assumptions has been considered for the detection of assumption violations of the environment to trigger recovery transitions that reestablish the correct behavior [149].

Lifting reactive synthesis towards distributed architectures makes the problem undecidable in general. Hence, similar results are implied in the unrealizability case. Nevertheless, for simple architectures, consisting only of two processes, pattern-based refinement techniques generating assume-guarantee specifications from the respective counter-strategies still can be considered [3]. Furthermore, for general architectures, unrealizability can be witnessed via sets of traces, whose elements are utilized for disproving every possible implementation strategy [46].

In summary, the identification of issues in faulty specifications for reactive systems has been experienced to be a challenging task (cf. [85]). The key challenges are the identification of core properties that produce unrealizabil-

ity and the identification of the mismatch between a specified property and the designer's intend. If both of these indicators are revealed, then they can be leveraged together to provide feedback for the user, or even to automatically repair the specification. Automatically deriving the designer's intend is, however, a highly problem specific task and, thus, very hard, if considered to be resolved automatically. Especially, as the resolution of an error according to the designer's intend may cause the requirement of other changes in the specification, which then need to be resolved as well. Handling such chains of required changes thus usually behaves too diverge to be managed by a completely automated process.

A more reasonable approach is to identify classes of problem types for unrealizability instead, which regularly appear during the design of temporal specifications. Therefore, if errors are detected according to multiple such classes, then class-specific explanations of the errors' sources can be provided. Designers then can choose among the presented options to select the one that best fits their needs. Moreover, they get a much better overview of the required tweaks that are necessary in order to archive realizability.

In this sense, we present a method for targeting a specific type of errors leading to unrealizability. We consider problems that are introduced by temporal dependencies between the actions of the system and the environment players. To this end, consider an example where a data value, arriving via an input stream, must be saved whenever a specific property $\mathtt{p}$ of that value changes. On a first instance a developer could formalize this requirement $\varphi$ as follows:

$$\varphi \;:=\; \square\big(\neg(\mathtt{p\ i} \leftrightarrow \bigcirc\, \mathtt{p\ i}) \leftrightarrow [\,\mathtt{s} \leftarrowtail \mathtt{i}\,]\big)$$

As it turns out, the formula $\varphi$ is unrealizable. Regarding the aforementioned approaches, $\varphi$ is satisfiable, as for example witnessed by the case where the input never changes, and even strong satisfiable, since knowing the whole future sequence of inputs in advance allows to schedule the updates at the right positions in time. Furthermore, the whole sub-formula below the $\square$-operator already provides a minimal unrealizable core, since every element of this subformula is relevant for the unrealizability result. Finally, note that $\varphi$ could be fixed by adding environment assumptions, like for example *"the input never changes"*, but regarding the property's intend this is of little help. The problem that the developer introduced here, is that a change of the property $\mathtt{p\ i}$ requires to compare $\mathtt{p\ i}$ at the time of the update with the previous evaluation, which is at the second evaluation of $\mathtt{p\ i}$. In the specification $\varphi$, however, the update is required too early. Hence, the developer missed putting a $\bigcirc$-operator before the update term $[\,\mathtt{s} \leftarrowtail \mathtt{i}\,]$.

For such types of unrealizable specifications developers instead need the right explanation of the cause for unrealizability describing the temporal conflict between the players. Such a kind of mismatch in the temporal dependencies can be identified by using *delay games* as already considered by Hosch and Landweber in 1972 [64]. In a delay game, the requirement of a strict alternation between the system and environment is relaxed. As a result, one of the players can postpone her moves to obtain a lookahead on the opponent's moves. With lookahead at hand, the specification $\varphi$ is realizable, since the system can delay the output by a single time step to match the temporal dependencies. Thus, with the introduction of delay, the developer gets immediate feedback of the error's nature and even a resolution strategy, as delivered by the winning strategy of the delay game.

The exact complexity of solving delay games and the bounds on the required lookahead to be utilized by the system player previously had been unknown. The result of Hosch and Landweber shows that it is decidable whether a delay game with $\omega$-regular winning condition is won with bounded lookahead, but an upper bound on the required lookahead could not be established. The result was improved by Holtmann, Kaiser, and Thomas, who proved a first doubly-exponential upper bound on the lookahead for delay games with parity winning conditions and showed that those games can be solved in doubly-exponential time [62]. We improve on these results by lowering the required lookahead and the complexity to a single exponent and show that both bounds are tight. Furthermore, we also lift the problem to full LTL, *i.e.*, while solving realizability for LTL is doubly-exponential in the size of the formula it rises to triply-exponential with the introduction of delay. Furthermore, triply-exponential lookahead is sufficient, but at the same time may be necessary for the system in order to win. More details about delay games and the corresponding proofs of the aforementioned results are presented in Chapter V.

# 4 The Reader's Guide to the Thesis

We start slowly, with preliminaries in Chapter II, building a common ground for the later considerations. As a warm-up, we introduce the reactive system model, as well as notions for infinite words and trees. Then, we expand our thinking to infinite games together with the notions of the corresponding system and environment strategies. We continue by introducing Mealy machines and conclude with Linear Temporal Logic (LTL), considered to be the traditional specification logic for reactive systems. We fade out with a system example to put ourselves again into the read-to-start positions for the upcoming discussions.

Arriving at Chapter III, we then immediately head over to TSL. We consider the necessary changes to the underlying reactive system models, along which we explore the definition of the logic itself, as well as its realizability and synthesis problems. Then, we revisit the kitchen timer specification from the introduction and formulate it, this time precisely, by using TSL. Beside that, we also consider the specification of a music player app to further advance our experience with the art of TSL system design.

Afterwards, we enter the adventure zone of our trip, which is introduced by the undecidability proof of TSL. We explore different resolution strategies, where we first analyze, whether a reduction to fragments of TSL can help in making the problem decidable. Unfortunately, this excursion does not yield to a satisfactory result. Hence, we try to rescue the situation, by switching to the world of infinite games instead. There, we explore step by step which changes to the standard game semantics are necessary as soon as predicate and update terms are added to the setting. We especially dive into the determinacy question of these games, as well as into the restrictions that are needed to secure ourselves against the rising potential of dangerous infinite branching. Along the way, we further discover the answer to the question of how much memory the players need in order to win a TSL based game. It turns out that both players may need infinite memory in order to win.

Armed with these new insights, we then leave the game world behind, and again tackle the undecidability problem from the logic's side. We introduce our final solution using an approximate reduction of TSL to LTL that is interactively refined in a CEGAR like fashion. We show that the approximation is sound, but incomplete. Therefore, we are able to effectively synthesize reactive control behavior that is guaranteed to satisfy the given TSL specifications by construction. In this context, we then have a closer look into the synthesized results yielding towards our idea of a Control Flow Model (CFM) that

provides an intermediate abstraction for different implementation frameworks.

Next, we connect the abstract CFM implementation model to Functional Reactive Programming (FRP). To this end, we introduce the concept of FRP in general and give an overview of the leveraged programming concepts. We consider the time model of FRP and how it connects to the classical reactive system models and temporal logics. Furthermore, we learn about different FRP instantiations in terms of programming patterns. Therefore, our discussion covers Arrowized, Applicative, and Monadic FRP.

We finish with a selection of the different application domains that can be targeted with TSL synthesis. To this end, we present domain specific examples that demonstrate the flexibility of TSL. At the same time, a detailed analysis of the application specifics and their influence on the synthesis performance has been prepared. The discussion is complemented by an experimental evaluation, where the TSL approximation has been analyzed in combination with different state of the art LTL synthesis tools. The chapter is closed with a short summary of the successfully targeted challenges, as well as the remaining open questions.

Some parts of Chapter III already have been published in previous research papers. The logic TSL originally was introduced by the paper *Temporal Stream Logic - Synthesis beyond the Bools*, which was published at the 31th International Conference on Computer Aided Verification (CAV 2019). The connection to FRP was first explored by the paper *Synthesizing Functional Reactive Programs*, published at the Haskell Symposium 2019. Both papers have been created by the thesis author together with Mark Santolucito, Ruzica Piskac, and Bernd Finkbeiner. They also cover all of the experimental evaluations that are presented in the chapter. All game related considerations, however, and the excursion with respect to fragments of TSL are extending this research and have not been published so far.

Even with the separation of data and control through TSL at hand, there still are other problems that limit synthesis in terms of practical applications. These problems already can be considered at the level of LTL. One of these problems is the inspectability on the synthesized results, which is especially of interest for specification designers in order to verify their design intents. While the synthesis result is ensured to be functionally correct according to the correct-by-construction paradigm, there is still no guarantee that the returned solutions not unnecessarily exhibit a complex representation.

These problems can be targeted with output sensitive synthesis methods, which measure the algorithmic complexity of the problem in terms of output metrics, instead of the input with respect to the specification size. For an

overview, we first revisit the idea of bounded synthesis [45], which actively bounds the solution size with the bound being considered as an additional input to the synthesis problem. The approach sets the basis for our later considered extensions that are based on other metrics than the solution size.

One of these metrics is the number of simple cycles in the solution graph. We pursue the intuition that a large number of simple cycles also increases the amount of potentially varying behavior. Thus, keeping the number of simple cycles at a minimum reduces potential behavior changes, finally leading to simpler solutions in general. We underline this intuition with theory and some experiments. To this end, we prove that the number of simple cycles indeed is an explosive metric. We show that it can explode triply-exponential in the formula size, which is even worse than the maximal explosion in the size of the solution graph that only grows at most doubly-exponentially in the formula size. Thus, keeping the number of simple cycles small is indeed helpful in order to avoid such kind of an explosion. Our experimental evaluation also reveals that the synthesized solutions, for which the number of simple cycles is bounded in addition to the solution size, indeed are better to inspect. Furthermore, according to our evaluation bounding the number of simple cycles introduces no additional overhead in terms of synthesis times.

We further consider output sensitive methods that depend on the representation of the synthesized result. The most simple output model is a Mealy machine, which reflects the configuration graph of a solution only in terms of a flat graph representation. Other models, however, deliver more compact representations as well, like Boolean circuits or programs. Therefore, we also consider output sensitive solutions that target metrics, which are specific to these compact models. We consider the number of gates and latches as part of the circuit representation, the size of the intermediate run graphs, and the number of Boolean variables of assembler like or nested loop-based programs. We provide SAT based encodings for all of these approaches and compare them according to the observable tradeoffs with an experimental evaluation.

Chapter IV covers research that already has been published as part of previous publications. *Bounded Synthesis* originally was introduced by Sven Schewe and Bernd Finkbeiner and is only revisited for the sake of completion. Bounding the number of simple cycles first was explored in the paper *Bounded Cycle Synthesis* published at the 28th International Conference on Computer Aided Verification (CAV 2016). The paper was written by the thesis author together with Bernd Finkbeiner. A first exploration of output sensitive synthesis methods for reactive programs was given by the paper *Bounded Synthesis of Reactive Programs* published at the International Symposium on Automated Technology for Verification and Analysis (ATVA 2018). The pa-

per was written by the thesis author together with Carsten Gerstacker and Bernd Finkbeiner. However, the paper primarily focuses on the comparison of the output sensitive approach with an automata based equivalent, as introduced by Madhusudan. This thesis, instead, focuses on the underlying SAT encoding and the comparison with other output sensitive methods that are based on different representation models.

Verification is used to identify whether a manually created implementation satisfies the specification, while synthesis only creates specifications that are correct by design. However, this does not imply that there always are systems that satisfy the specification. While engineers cannot introduce errors any more, since they have been removed from the equation, the designers still can, because they now are in charge of creating a specification that covers all of their design intents. Nevertheless, there is no guarantee that designers only come with intends that always are conflict free. Consequently, specification developers must be able to fix and debug specifications that are unrealizable as well.

There are already many approaches that deliver feedback for an unrealizable specification, but some specific types of errors still are hard to detect. To this end, we consider the introduction of delay to the underlying infinite game in Chapter V. In a delay game, one player obtains a lookahead on her opponent's moves, which may allow her to win games with lookahead that she would loose otherwise. The introduction of delay, thus, offers new debugging possibilities for errors that are caused by access to inputs that are assumed to arrive too early in time.

Delay games with parity winning conditions already have been considered in the past, but previous results only established a doubly-exponential upper bound on their algorithmic complexity. We improve these bounds to a single exponent and at the same time complement them with matching lower bounds. To this end, we also consider the special cases of reachability and safety winning conditions in more detail. Our final results show that an additional exponential blowup always arises when introducing lookahead to games with $\omega$-regular winning conditions. Thus, using delay as a debugging method for specifications that are only realizable with lookahead comes at an additional cost. The chapter formally introduces the delay game setting and provides proofs for all of the aforementioned results.

Some results of Chapter V already have been published as part of previous publications. The upper and lower bounds for delay games with reachability, safety, and parity conditions first have been introduced by the paper *How much Lookahead is needed to win Infinite Games?* published at the

42nd International Colloquium on Automata, Languages, and Programming (ICALP 2015). In the original paper, two proofs are given for the upper bounds. The first one additionally provides a method for solving delay games with parity winning conditions, while the second one proves a slightly better bound, but does not yield a constructive algorithm. In this work, an adapted version of the first proof is given that matches the better bounds of the aforementioned second proof. Thus, only a single proof is needed. Another difference is that the games of the original work are played implicitly on the languages of $\omega$-automata, while in this work we target a concrete graph based game definition instead, where the edges are labeled by atomic proposition, thus, inducing an $\omega$-regular language instead. The lower bounds for delay games with LTL winning conditions originally are from the paper *Prompt Delay* published at the 36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2016). Both papers have been created by the thesis author together with Martin Zimmermann.

The thesis closes with a final discussion in Chapter VI.

# Chapter II

# Preliminaries

We use the symbol $\mathbb{N}$ to denote the set of *non-negative integers* and $\mathbb{N}^+$ to denote the set of *positive integers*. Elements of $\mathbb{N}$ are abbreviated by small roman letters, preferably $n, m, j$ and $k$. For all $n, m \in \mathbb{N}$ with $n \leq m$ we use $[n, m]$ to denote the set $\{n, n+1, ..., m\}$ and shorten the special case of $[0, n-1]$ by $[n]$, where $[0] = \emptyset$ and $[\infty] = \mathbb{N}$. To denote *sets*, we prefer using big roman letters like $S, P$ and $Q$. The *cardinality* of a set $S$ is denoted by $|S|$, where for infinite sets $S$ we fix $|S| = \infty$. The *power set* of $S$ is denoted by $2^S$.

A *value* is an arbitrary object of arbitrary type. We use $\mathcal{V}$ to denote the set of all values. A special subset is given by the set of *Boolean values* $\mathcal{B} \subseteq \mathcal{V}$, which are either $\mathit{true} \in \mathcal{B}$ or $\mathit{false} \in \mathcal{B}$. An $n$-ary *function* $f \colon \mathcal{V}^n \to \mathcal{V}$ determines a new value from $n$ given values. We denote the set of all functions (of arbitrary arity) by $\mathcal{F}$. The arity of an $n$-ary function $f$ is accessed by $\sharp(f) = n$. *Constants* $c$ are functions of zero arity and at the same time values, *i.e.*, $c \in \mathcal{F} \cap \mathcal{V}$. An $n$-ary *predicate* $p \colon \mathcal{V}^n \to \mathcal{B}$ checks a truth statement on $n$ given values. The set of all predicates (of arbitrary arity) is denoted by $\mathcal{P} \subseteq \mathcal{F}$. For a function $f \colon A \to B$, mapping elements from a domain $A \subseteq \mathcal{V}$ to a co-domain $B \subseteq \mathcal{V}$, and some set $C \subseteq A$ we use $f(C)$ for the set $\{b \in B \mid c \in C \wedge f(c) = b\}$. For the sake of readability, we also use $B^{[A]}$ to denote the set of all total functions with domain $A$ and image $B$.

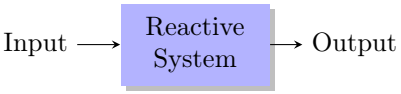
If $X$ is a set and $n \in \mathbb{N}$ we use $X^n$ to denote the set of $n$-ary *tuples* over $X$ and number its components from 0 to $n-1$. The length of an $n$-ary tuple $t$ is given by $\sharp(t) = n$. We use $\sharp$ for the arity of functions, as well as for tuples lengths', since the interpretation is always clear from the context. The *projection* $pr_j$ projects to the $j$-th component of a tuple of arbitrary length for any $j \in \mathbb{N}$. If the tuple has no such component, then $pr_j$ is undefined. To denote concrete tuple instances, we use round brackets, e.g. $(1, 2, 3) \in \mathbb{N}^3$.

Let $X$ be some set of variables. A *Boolean formula* $\varphi$ over $X$ is a formula constructed according to the grammar $\varphi := \mathit{true} \mid x \in X \mid \neg\varphi \mid \varphi \wedge \varphi$. The semantics of Boolean formulas are defined with respect to truth assignments to the variables of $X$. *Truth assignments* are given by subsets $Y \subseteq X$, where all variables in $Y$ are assigned *true* and all variables not in $Y$ are assigned *false*. Semantics then are defined via a *satisfaction relation* $\models$, where $Y \models x$ iff $x \in Y$, $Y \models \neg\varphi$ iff $X \setminus Y \models \varphi$, and $Y \models \varphi \wedge \vartheta$ iff $Y \models \varphi$ and $Y \models \vartheta$.

Additional derived Boolean operations are defined as usual: *false* $:= \neg$*true*, *disjunction* $\varphi \vee \vartheta := \neg(\neg\varphi \wedge \neg\vartheta)$, and *implication* $\varphi \rightarrow \vartheta := \neg\varphi \vee \vartheta$. We use $\mathbb{B}^*(X)$ to denote the set of all *Boolean formulas* over a set of variables $X$. Furthermore, $\mathbb{B}^+(X)$ denotes the set of all *positive Boolean formulas*, *i.e.*, formulas build by only using the primitives *true*, *false*, variables $x \in X$, conjunction $\wedge$, and disjunction $\vee$. For a set of variables $X$, we use the constraint $exactly_n(X)$ to denote: exactly $n \in \mathbb{N}$ variables of $X$ must be satisfied.

# 1 Reactive Systems

Reactive systems are systems that receive input from the environment, manage internal state, and produce output to the environment.




As the systems are reactive, these actions happen repeatedly over the infinite amount of time. In general, this definition matches almost any system in our environment, from biological systems over ecological ones up to the entire universe. However, in the scope of this theses, we only use it to capture physical computation devices, possible equipped with physical sensors and actors, and their respective models used for the formal description and analysis in computer science. In this scope, we consider a simplified model of time, where we assume that time is represented as a countably totally ordered sequence of points in time, as mathematically reflected by $\mathbb{N}$. In computer science, such a time model is usually sufficient, since the physical principles today's computers use to execute calculations depend on discrete state and, thus, need discrete updates as part of their execution semantics.

We further restrict ourselves to the synchronous model of time, where time is discrete and the process of reading input and producing output strictly alternates. In this model, the internal state then either is updated between reading inputs and producing outputs or together with the production of the output. All other models, that do not fit the aforementioned characteristics, are usually considered to be asynchronous.

For the synchronous model, two major mathematical frameworks have been proven useful to express the behavior of reactive systems over time: infinite words and infinite trees, where the latter is an extension of the former. While strongly simplifying the capabilities of real world models, infinite words and trees still allow to analyze important properties of reactive systems and provide fundamental building blocks to express their temporal behavior.

# 2 Infinite Words

An *alphabet* is a non-empty, finite set of symbols, usually denoted by $\Sigma$ or $\Upsilon$. Elements of an alphabet are called *letters*. Let an alphabet $\Sigma$ be given, then the concatenation $w = w_0 w_1 \dots w_{n-1}$ of finitely many letters of $\Sigma$ is called a *finite word* over $\Sigma$, where $n$ defines the length of $w$ also denoted by $|w|$. The only word of length 0 is the *empty word* denoted by $\varepsilon$.

The concatenation of infinitely many letters defines an *infinite word* which has infinite length. We usually use small roman letters $w$ and $v$ to denote finite words and small greek letters $\alpha$, $\beta$, and $\nu$ to denote infinite words. For words that are either finite or infinite we default to denote them by small roman letters $w$ and $v$. The set of all finite words over $\Sigma$ is denoted by $\Sigma^*$ and the set of all infinite words by $\Sigma^\omega$. Their union is denoted by $\Sigma^{*/\omega} = \Sigma^* \cup \Sigma^\omega$. For $\Sigma^* \setminus \{\varepsilon\}$ we also use the shorter notation $\Sigma^+$. Given some word $w \in \Sigma^{*/\omega}$ we have that for all $n \in [|w|]$ the $n$-th letter of $w$ is denoted by $w_n$ and the first letter is at $w_0$. Every function $f \colon \Sigma \to \Sigma'$ is lifted to words $w \in \Sigma^{*/\omega}$ by point-wise application, *i.e.*, $f(w) = f(w_0)f(w_1)\dots \in (\Sigma')^{*/\omega}$.

Like letters, we can concatenate any finite word $v \in \Sigma^*$ with words $w \in \Sigma^{*/\omega}$ to new words $vw$. For a word $w = v_0 v_1$ we call $v_0$ the prefix of $w$ and $v_1$ it's suffix. If $\Sigma$ is an alphabet then each subset of $\Sigma^*$ is a language over finite words and each subset of $\Sigma^\omega$ is a language over infinite words. Respectively, each subset of $\Sigma^{*/\omega}$ is a language over finite and infinite words.

For infinite words $\alpha \in \Sigma^\omega$ we are also interested in letters of $\alpha$ that appear infinitely often. To this end, we utilize the notion of the *infinity set* $\mathrm{Inf}(\alpha)$, defined as $\mathrm{Inf}(\alpha) = \{\sigma \in \Sigma \mid \forall n \in \mathbb{N}.\ \exists m > n.\ \alpha_m = \sigma\}$.

# 3 Infinite Trees

A $\Sigma$-*labeled* $\Upsilon$-*tree* $t \colon \Upsilon^* \rightharpoonup \Sigma$ is a partial function, defined over a domain $\mathcal{D}_t \subseteq \Upsilon^*$, that maps positions in the tree to labels of $\Sigma$. The domain $\mathcal{D}_t$ needs to be *prefixed closed*, *i.e.*, for any $vw \in \mathcal{D}_t$ with $v, w \in \Upsilon^*$ it must be that $v \in \mathcal{D}_t$. If $\mathcal{D}_t = \Upsilon^*$, then we say that $t$ is *input-complete*. The empty word $\varepsilon$ denotes the root of a tree, from which the tree branches according to it's directions $\Upsilon$. A *branch* $\nu \subseteq \Upsilon^{*/\omega}$ is a finite or infinite sequence of directions such that every prefix $w$ of $\nu$ is in the domain of $t$, *i.e.*, $w \in \mathcal{D}_t$. Finite branches $w \in \Upsilon^*$ are not allowed to have further children after the last position of the branch, *i.e.*, $\forall v \in \Upsilon.\ \nu v \notin \mathcal{D}_t$. The *outcome* of a branch $\nu \in \Upsilon^{*/\omega}$ of a tree $t \colon \Upsilon^* \rightharpoonup \Sigma$ is the sequence of labels appearing along the branch, denoted by $t \wr \nu$. It is defined via the unique

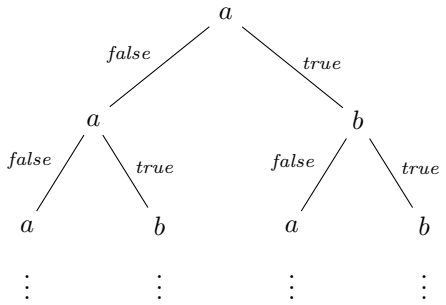



Figure 9: An example for the graphical representation of a tree. The depicted tree branches according to $\mathcal{B}$ and is labeled by letters from $\{a, b\}$.

sequence $t \wr \nu = \alpha$ such that $|\alpha| = |\nu|$ and $\forall n \in [|\nu|]$. $\alpha_n = t(\nu_0 \nu_1 \dots \nu_n)$. Every tree $t$ induces a word language $\mathcal{L}(t) = \{ t \wr \nu \mid \nu \in \mathcal{D}_t \}$. If the tree $t$ is input-complete, then it's induced language $\mathcal{L}(t)$ is input complete as well. If $\Upsilon = \mathcal{B}$, then we call $t$ a binary tree. We depict trees the classical way, where the structure of the tree is given as a graph, positions are identified by the corresponding labels and branches are labeled with the corresponding directions. An example of a tree illustration is given in Figure 9. The tree is an input-complete binary tree that branches according to the Boolean values *true* and *false* and is labeled by the letters $a$ and $b$. In general, we depict trees such that they grow from the top to the bottom.

# 4 Infinite Games

The interaction between the environment and the system can be considered as an infinite, two player game between the *input player* "Player $I$", representing the environment, and the *output player* "Player $O$", representing the system. The rules of the game are given in form of a game arena, which consists of vertices, either owned by Player $I$ or Player $O$, and an edge relation labeled by possible moves. The players play in the arena by moving a single token along the edges, which has been placed on some designated vertex initially. The player that owns the vertex currently holding the token is in charge of moving it to a successor. Therefore, a player can choose among any successor that is allowed by its transition relation. By choosing a successor, the player produces a symbol, as given by the transition relation.

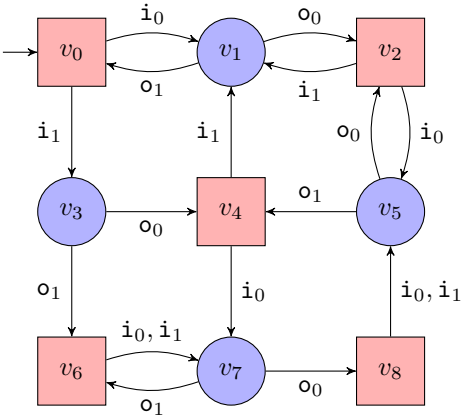



Figure 10: Graphical example of an arena. The round vertices describe the vertices owned by Player $O$ and the angled ones the vertices owned by Player $I$. The edges in between denote the possible moves of the players. The initial vertex is marked by an incoming arrow without a source.

**Definition 1.** An *arena* $\mathcal{A} = (\Sigma_I, \Sigma_O, V_I, V_O, v_I, \delta_I, \delta_O)$ is a tuple where

- $\Sigma_I$ is a non-empty, finite set of inputs,
- $\Sigma_O$ is a non-empty, finite set of outputs,
- $V_I$ is the set of vertices owned by Player $I$,
- $V_O$ is the set of vertices owned by Player $O$,
- $v_I \in V_I$ is the initial vertex,
- $\delta_I \colon V_I \times \Sigma_I \to V_O$ is the transition relation of Player $I$, and
- $\delta_O \colon V_O \times \Sigma_O \to V_I$ is the transition relation of Player $O$.

The size of $\mathcal{A}$, denoted by $|\mathcal{A}|$, is defined to be $|V_I| + |V_O|$. If $V_I$ and $V_O$ are clear from the context, then we also use $V = V_I \cup V_O$ to denote the set of all vertices of an arena.

An example arena $\mathcal{A}^e = (\Sigma_I, \Sigma_O, V_I, V_O, v_0, \delta_I, \delta_O)$ is depicted in Figure 10. It consists of five vertices owned by Player $I$ ($V_I = \{v_0, v_2, v_4, v_6, v_8\}$) and four

vertices owned by Player $O$ ($V_O = \{v_1, v_3, v_5, v_7\}$). The players play in the arena by moving a single token along the edges, which has been placed on $v_0$ initially. The player that owns the vertex currently holding the token is in charge of moving it to a successor. Therefore, a player can choose among any successor that is allowed by its transition relation. By choosing a successor, the player produces a symbol, as given by the transition relation. In the example, Player $I$ produces either the input $\mathtt{i}_0$ or the input $\mathtt{i}_1$ and Player $O$ produces either the output $\mathtt{o}_0$ or the output $\mathtt{o}_1$. Moving the token ad infinitum through the arena produces an infinite play.

**Definition 2.** A *play* $\rho$ in an arena $\mathcal{A} = (\Sigma_I, \Sigma_O, V_I, V_O, v_I, \delta_I, \delta_O)$ is an infinite sequence

$$\rho = (v_0^I, i_0, v_0^O, o_0)(v_1^I, i_1, v_1^O, o_1)\ldots \in (V_I \times \Sigma_I \times V_O \times \Sigma_O)^\omega$$

such that $v_0^I = v_I$ and for all $n \in \mathbb{N}$ we have that $\delta_I(v_n^I, i_n) = v_n^O$ and $\delta_O(v_n^O, o_n) = v_{n+1}^I$. The set of all possible plays on $\mathcal{A}$ is denoted by $\mathrm{Plays}(\mathcal{A})$.

Some plays in the arena $\mathcal{A}^e$ of Figure 10 would be for example:

1. $\rho_e^1 = (v_0, \mathtt{i}_0, v_1, \mathtt{o}_1)(v_0, \mathtt{i}_1, v_3, \mathtt{o}_0)\big((v_4, \mathtt{i}_1, v_1, \mathtt{o}_0)(v_2, \mathtt{i}_0, v_5, \mathtt{o}_1)\big)^\omega$
2. $\rho_e^2 = (v_0, \mathtt{i}_1, v_3, \mathtt{o}_0)\big((v_4, \mathtt{i}_0, v_7, \mathtt{o}_1)(v_6, \mathtt{i}_1, v_7, \mathtt{o}_0)(v_8, \mathtt{i}_0, v_5, \mathtt{o}_1)\big)^\omega$

Each players' decisions of which successors to choose are determined by the player's strategy. A strategy has access to the whole history of a play, *i.e.*, Player $P$ for $P \in \{I, O\}$ places it's decisions as responses to all previous decisions of the opponent Player $P_{\rightleftarrows}$ (where $I_{\rightleftarrows} = O$ and $O_{\rightleftarrows} = I$).

**Definition 3.** A *strategy* for Player $P \in \{I, O\}$ in a given arena $\mathcal{A} = (\Sigma_I, \Sigma_O, V_I, V_O, v_I, \delta_I, \delta_O)$ is an input-complete $\Sigma_P$-labeled $\Sigma_{P_{\rightleftarrows}}$-tree $\sigma_P \colon (\Sigma_{P_{\rightleftarrows}})^* \to \Sigma_P$.

A strategy $\sigma_O^e$ for Player $O$ on the arena $\mathcal{A}^e$ of Figure 10 would be for example:

$$\sigma_O^e(wv) = \begin{cases} \mathtt{o}_0 & \text{if } v \equiv \mathtt{i}_1 \\ \mathtt{o}_1 & \text{if } v \equiv \mathtt{i}_0 \end{cases}$$

Note that our strategy definition only takes into account the input/output symbols that have been chosen by the opponent player, but not the

vertex on which the token is currently placed on. This is sufficient, since the vertex holding the token can always be reconstructed from the previous choices of the opponent and the player's strategy when starting from the initial vertex.

**Definition 4.** A play $\rho = \rho_0\rho_1\ldots$ on an arena $\mathcal{A}$ is *consistent* with a strategy $\sigma_O$ of Player $O$ iff for every $n \in \mathbb{N}$ with $\rho_n = (v_n^I, i_n, v_n^O, o_n)$ we have that $\sigma_O(i_0 i_1 \ldots i_n) = o_n$. A strategy $\sigma_I$ of Player $I$ is consistent with $\rho$ iff $\sigma_I(o_0 o_1 \ldots o_{n-1}) = i_n$ for all $n \in \mathbb{N}$. The set of all plays consistent with a strategy $\sigma_P$ is denoted by $\mathrm{Plays}(\mathcal{A}, \sigma_P)$.

Regarding our example, both of the aforementioned plays $\rho_e^1$ and $\rho_e^2$ are consistent with the strategy $\sigma_O^e$.

To play a game in an arena, it remains to determine which of the players has won after the players played ad infinitum. To this end, we equip the arena with an additional winning condition, which fixes the set of infinite plays that are winning for Player $O$ and loosing for Player $I$. At the same time, the complement set fixes the plays that are winning for Player $I$ and loosing for Player $O$.

**Definition 5.** A *game* $\mathcal{G} = (\mathcal{A}, \mathrm{Win})$ is a tuple consisting of an arena $\mathcal{A}$ and a set of infinite winning plays $\mathrm{Win} \subseteq \mathrm{Plays}(\mathcal{A})$ through the arena. We call a play $\rho$ winning for Player $O$ iff $\rho \in \mathrm{Win}$ and winning for Player $I$ otherwise.

We consider the following standard winning conditions Win over infinite sequences of elements from $\Xi = V_I \times \Sigma_I \times V_O \times \Sigma_O$, *i.e.*, with $\mathrm{Win} \subseteq \Xi^\omega$:

- The *safety* winning condition fixes a special subset $S \subseteq V_I \cup V_O$ of *safe* vertices. A play is winning iff all vertices visited are elements of $S$.

$$\textsc{Safety}(S) = \{\, \alpha \in \Xi^\omega \mid \forall n \in \mathbb{N}.\ pr_0(\alpha_n) \in S \wedge pr_2(\alpha_n) \in S \,\}$$

- The *reachability* winning condition fixes a special subset $R \subseteq V_I \cup V_O$ of *goal* vertices. A play is winning iff it visits at least one goal vertex of $R$.

$$\textsc{Reach}(R) = \{\, \alpha \in \Xi^\omega \mid \exists n \in \mathbb{N}.\ pr_0(\alpha_n) \in R \vee pr_2(\alpha_n) \in R \,\}$$

- The *parity* winning condition is defined with respect to a coloring function $\Omega\colon V_I \cup V_O \to \mathbb{N}$ that assigns each vertex a natural number, the so called *color* of the vertex. A play is winning iff the maximal color seen infinitely often is even.

$$\textsc{Parity}(\Omega) = \{\, \alpha \in \Xi^\omega \mid \max(\mathrm{Inf}(\Omega(pr_0(\alpha))) \cup \mathrm{Inf}(\Omega(pr_2(\alpha)))) \textit{ is even} \,\}$$

Let $\mathcal{A} = (\Sigma_I, \Sigma_O, V_I, V_O, v_I, \delta_I, \delta_O)$, then $\mathcal{G}_\forall = (\mathcal{A}, \textsc{Safety}(S))$ with $S \subseteq V$ is a safety game, $\mathcal{G}_\exists = (\mathcal{A}, \textsc{Reach}(R))$ with $R \subseteq V$ is a reachability game, and $\mathcal{G}_\Omega = (\mathcal{A}, \textsc{Parity}(\Omega))$ with $\Omega\colon V \to \mathbb{N}$ is a parity game.

For example, $\mathcal{G}^e_\exists = (\mathcal{A}^e, \textsc{Reach}(\{v_5\}))$ is a reachability game that is played in the arena of Figure 10 with the goal of Player $O$ to reach vertex $v_5$.

**Definition 6.** Let $\mathcal{G} = (\mathcal{A}, \mathrm{Win})$ be a game and $\sigma_P$ be a strategy for Player $P$ on $\mathcal{A}$. The strategy $\sigma_P$ is a *winning strategy* iff every play $\rho \in \mathrm{Plays}(\mathcal{A}, \sigma_P)$ is winning for Player $P$.

Both of the aforementioned plays $\rho^0_e$ and $\rho^1_e$ are winning for Player $O$, but the strategy $\sigma^e_O$ is not, since Player $I$ can trap it's opponent to the set of vertices $v_0$, $v_1$, and $v_2$ by always moving back to $v_1$. Correspondingly, Player $I$ has a winning strategy in the game $\mathcal{G}^e_\exists$ instead. We call games where always one of the players has a winning strategy determined.

**Definition 7.** A game $\mathcal{G} = (\mathcal{A}, \mathrm{Win})$ is *determined* iff there is either a winning strategy $\sigma_O$ for Player $O$ or a winning strategy $\sigma_I$ for Player $I$.

Note that by the definition of a winning condition it is impossible that both players win a game. However, it may be possible that none of the players has a winning strategy, in which case the game is not determined. For all winning conditions considered so far (safety, reachability and parity), the corresponding games are all determined (cf. for example [56]).

Also note that safety, reachability and parity winning conditions only take the vertices into account, that have been visited during a play, but completely ignore the edge labeling. This is also the reason why games in the literature are often defined without an additional edge labeling. However, we will consider more advanced winning conditions later in this thesis that require to take the edge labeling into account.

# 5 Implementations

While infinite trees provide a sound and complete model to express the behavior of a reactive system, they are not amenable to be used in practice, since their representation is infinite by definition. Thus, we only use them as part of our mathematical toolbox. For practical considerations, we also require compact models that are designed to provide finite model representations.

One of the oldest and most popular models is given by Mealy machines, which are named after their inventor: George H. Mealy [104]. Intuitively, the model compresses a tree into a finite transition graph, where the states of the graph identify the repetitive behavior of the tree. Similarly, a Mealy machine can be considered as a complete infinite tree, whose labeling is extended with collections of labels indicating the states and where all sub-trees starting in a position labeled with the same state are consistent among the whole tree.

**Definition 8.** *Mealy machines* $\mathbb{M} = (\Sigma_I, \Sigma_O, M, m_I, \delta_{\mathbb{M}}, \ell)$ consist of

- a set of inputs $\Sigma_I$,
- a set of outputs $\Sigma_O$,
- a set of states $M$,
- an initial state $m_I \in M$
- a transition function $\delta_{\mathbb{M}} \colon M \times \Sigma_I \to M$, and
- an output function $\ell \colon M \times \Sigma_I \to \Sigma_O$.

The outputs produced by a Mealy machine only depend on the current state and the last inputs. The size of $\mathbb{M}$, denoted by $|\mathbb{M}|$, is defined to be $|M|$. The transition function $\delta_{\mathbb{M}}$ can be lifted to words over $\Sigma_I$ using the function $\delta^*_{\mathbb{M}} \colon (\Sigma_I)^* \to M$, recursively defined via $\delta^*_{\mathbb{M}}(\varepsilon) = m_I$ at first and $\delta^*_{\mathbb{M}}(wv) = \delta_{\mathbb{M}}(\delta^*_{\mathbb{M}}(w), v)$ for the remaining steps.

**Construction 1.** Every Mealy machine $\mathbb{M}$ induces an input-complete infinite tree $t_{\mathbb{M}} \colon (\Sigma_I)^* \to \Sigma_O$ via the infinite application of $\delta_{\mathbb{M}}$ and $\ell$, *i.e.*,

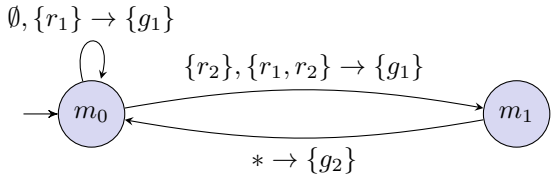



Figure 11: Graphical representation of the Mealy machine $\mathbb{M}^e$.

for all $n \in \mathbb{N}$ it holds that

$$t_{\mathbb{M}}(v_0 v_1 \dots v_n) = \ell(\delta^*_{\mathbb{M}}(v_0 v_1 \dots v_{n-1}), v_n).$$

We define the semantics of $\mathbb{M}$ by the induced tree $t_{\mathbb{M}}$, leading to the word language $\mathcal{L}(\mathbb{M})$ of $\mathbb{M}$ to be defined as $\mathcal{L}(t_{\mathbb{M}})$. Mealy machines are depicted as directed graphs, where vertices represent the states of the machine and edges the transitions. Furthermore, each transition is labeled by the input triggering the transition and the corresponding output produced by $\ell$. If multiple inputs produce the same output, we correspondingly group them together, where we use $*$ to denote arbitrary possible inputs. The initial state is marked by an incoming edge without a source.

Figure 11 depicts a Mealy machine $\mathbb{M}^e = (2^{\{r_1,r_2\}}, 2^{\{g_1,g_2\}}, \{m_0, m_1\}, m_0, \delta_{\mathbb{M}}, \ell)$ with

$$\delta_{\mathbb{M}}(m, v) := \begin{cases} m_1 & \text{if } m = m_0 \wedge r_2 \in v \\ m_0 & \text{otherwise} \end{cases} \quad \text{and}$$

$$\ell(m, v) := \begin{cases} \{g_1\} & \text{if } m = m_0 \\ \{g_2\} & \text{otherwise} \end{cases}$$

# 6 Linear Temporal Logic

So far, we only considered definitions of reactive system implementations. They are, however, not expressive enough to capture temporal behavior properties in general. Reactive system implementations are infinite trees that branch according to the inputs from the environment and are labeled with outputs that are produced by the system. Reactive system properties, on the

other hand, are sets of infinite trees covering all implementations that satisfy the property to be described. For practical applications, however, describing such sets explicitly is infeasible due to the infinity of time. Therefore, a more compact description is required. Natural ones are given by temporal logics, which bundle the infinite behavior of time into temporal operators, similar to the behavior, as it would be described using natural human language.

The most popular temporal logic is given by *Linear-time Temporal Logic* (LTL), as introduced by Pnueli in 1977 [116]. In general, LTL describes the temporal behavior of infinite words over sets of atomic propositions $\Sigma$. Nevertheless, LTL can also be used to describe properties over infinite trees. To this end, the alphabet is divided into input propositions $\mathcal{I}$ and output propositions $\mathcal{O}$. The properties then describe sets of infinite words over $\Sigma = 2^{\mathcal{I} \cup \mathcal{O}}$, which are lifted to infinite trees via branching among the input propositions $\mathcal{I}$. Remember that the input propositions are under the control of the environment. Therefore, the word languages induced by LTL only cover infinite trees that branch according to all inputs, as provided by the environment.

Syntactically, every LTL specification $\varphi$ adheres to the following grammar:

$$\varphi \quad := \quad \mathit{true} \mid a \in \mathcal{I} \cup \mathcal{O} \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc\varphi \mid \varphi \,\mathcal{U}\, \varphi$$

The size of a specification $\varphi$ is denoted by $|\varphi|$ and defines the number of subformulas of $\varphi$. The semantics of LTL are defined over infinite words $\alpha \in \Sigma^\omega$. We define the satisfaction of a word $\alpha$ at a position $n \in \mathbb{N}$ and a specification $\varphi$, denoted by $\alpha, n \vDash \varphi$, for the different choices of $\varphi$ as follows:

- $\alpha, n \vDash \mathit{true}$
- $\alpha, n \vDash a$ iff $a \in \alpha_n$
- $\alpha, n \vDash \neg\varphi$ iff $\alpha, n \nvDash \varphi$
- $\alpha, n \vDash \varphi \vee \vartheta$ iff $\alpha, n \vDash \varphi$ or $\alpha, i \vDash \vartheta$
- $\alpha, n \vDash \bigcirc\varphi$ iff $\alpha, n+1 \vDash \varphi$
- $\alpha, n \vDash \varphi \,\mathcal{U}\, \vartheta$ iff $\exists m \geq n.\ \alpha, m \vDash \vartheta$ and $\forall n \leq i < m.\ \alpha, i \vDash \varphi$

An infinite word $\alpha$ satisfies $\varphi$, denoted by $\alpha \vDash \varphi$, iff $\alpha, 0 \vDash \varphi$. The word language $\mathcal{L}(\varphi)$ is the set of all words that satisfy $\varphi$, i.e.,

$$\mathcal{L}(\varphi) = \{\alpha \in \Sigma^\omega \mid \alpha \vDash \varphi\}.$$

The tree language of $\varphi$ is the set of all trees $t$ such that $\mathcal{L}(t) \subseteq \mathcal{L}(\varphi)$. The $\bigcirc$ operator is called "next" and the $\mathcal{U}$ operator is called "until". Beside the defined

operators, we have the standard derivatives of the Boolean connectives, as well as eventually: $\Diamond\varphi \;:=\; \mathit{true}\,\mathcal{U}\,\varphi$, globally: $\Box\varphi \;:=\; \neg\Diamond\neg\varphi$, and the weak version of until: $\varphi\,\mathcal{W}\,\vartheta \;:=\; \varphi\,\mathcal{U}\,\vartheta \;\vee\; \Box\varphi$.
The following are some example temporal properties that can be defined using LTL:

1. *Order of Change.* The output $o \in \mathcal{O}$ is not enabled before output $o' \in \mathcal{O}$, which again is not enabled before another output $o'' \in \mathcal{O}$.

$$\varphi_0^e \;:=\; \neg o\,\mathcal{W}\,o' \;\wedge\; \neg o'\,\mathcal{W}\,o''$$

2. *Reactivity.* If at least one of the inputs $i \in \mathcal{I}$ changes infinitely often, then also at least one of the outputs $o \in \mathcal{O}$ must change infinitely often.

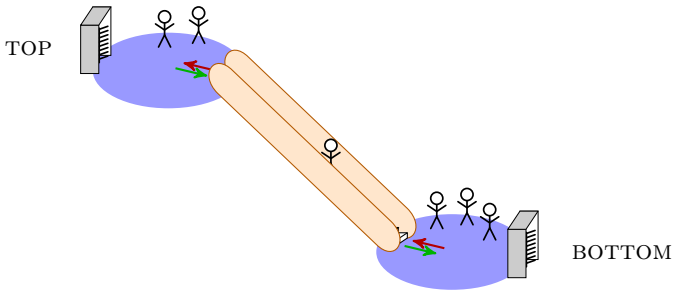
$$\varphi_1^e \;:=\; \bigvee_{i\in\mathcal{I}} \big(\Box\Diamond i \wedge \Box\Diamond\neg i\big) \rightarrow \bigvee_{o\in\mathcal{O}} \big(\Box\Diamond o \wedge \Box\Diamond\neg o\big)$$

3. *Simple Arbiter.* An arbiter manages the access to a shared resource among $n$ clients. Clients place requests $r_i \in \mathcal{I}$, which must be granted eventually via grants $g_i \in \mathcal{O}$. Furthermore, only a single grant can be given at every point in time, *i.e.*, grants are mutually exclusive.

$$\varphi_2^e \;:=\; \bigwedge_{j=0}^{n-1} \big(\Box(r_j \rightarrow \Diamond g_j)\big) \;\wedge\; \bigvee_{j=0}^{n-1} (g_j \rightarrow \bigwedge_{\substack{k=0\\k\neq j}}^{n-1} \neg g_k)$$

The *verification* problem asks for a given implementation $\mathbb{X}$, which induces a language $\mathcal{L}(\mathbb{X})$, and a given specification $\varphi$, both defined over the same sets of inputs $\mathcal{I}$ and outputs $\mathcal{O}$, whether the implementation satisfies the specification $\mathbb{X} \vDash \varphi$, or stated in terms of the languages, whether $\mathcal{L}(\mathbb{X}) \subseteq \mathcal{L}(\varphi)$. If only the specification $\varphi$ is given, then the *realizability* problem asks for the existence of an implementation that satisfies $\varphi$. If, however, the specification $\varphi$ is given and an implementation model is fixed, then the *synthesis* problem asks for a concrete implementation $\mathbb{X}$ that adheres to the model and satisfies $\varphi$. If the specification is unrealizable, then the synthesis problem is only required to return the unrealizability result, *e.g.*, "*the specification is unrealizable.*".

To demonstrate the differences and major advantages of synthesis against manual programming, we consider the design process of an escalator control specification, which is responsible for the reliable transport of passengers between two floors.




The escalator consists of a single transport belt, which can either move up or down. To observe the behavior of the passengers and to deduce their intentions, there are two sensors, at the top and at the bottom, that reliably detect whenever a passenger enters or exists the escalator. The escalator is always moving in one direction only. Hence, passengers can only use it when it is moving towards their target floor. This is why there are two additional waiting platforms behind the sensors for the passengers to wait, in case the escalator currently is moving into the wrong direction. Our goal is to create a controller that reliably delivers every passenger to his or her desired target floor. To describe the behavior of the controller in LTL, we use the Boolean input signals $\mathcal{I} = \{\texttt{enter}_{\langle\texttt{top}\rangle}, \texttt{enter}_{\langle\texttt{bottom}\rangle}, \texttt{exit}_{\langle\texttt{top}\rangle}, \texttt{exit}_{\langle\texttt{bottom}\rangle}\}$ to denote the feedback of the sensors at the bottom and top floors and the single Boolean output signal $\mathcal{O} = \{\texttt{move}_{\langle\texttt{up}\rangle}\}$ to denote that the escalator is moving upwards, if set, and moving downwards otherwise. The property of delivering every passenger then can be formalized for both directions with the following guarantees:

$$\square\big(\texttt{enter}_{\langle\texttt{bottom}\rangle} \rightarrow \diamondsuit\big(\texttt{move}_{\langle\texttt{up}\rangle}\,\mathcal{U}\,\texttt{exit}_{\langle\texttt{top}\rangle}\big)\big) \tag{G1}$$

$$\square\big(\texttt{enter}_{\langle\texttt{top}\rangle} \rightarrow \diamondsuit\big(\neg\,\texttt{move}_{\langle\texttt{up}\rangle}\,\mathcal{U}\,\texttt{exit}_{\langle\texttt{bottom}\rangle}\big)\big) \tag{G2}$$

Unfortunately, these two guarantees are not sufficient yet, because the specification that results from the conjunction of the guarantees is unrealizable. The reason is that there is no way to ensure that the top sensor will produce an exit signal after the bottom sensor recognized a passenger entering.

Indeed, in theory it is possible that a passenger enters at the bottom and waits in the waiting area forever. In this case, the given setup of the two sensors is too limited to detect the behavior of the passengers. The controller just cannot determine the exact position of every passenger in the system. However, in practice we can assume that such a behavior will not occur (no passenger has the patience to wait forever, only to break the system).

In order to fix the specification, we need to add this additional assumption. The question that still remains open is: how long does it take until a passenger leaves at the top after entering at the bottom? Note that a passenger only can reach the top floor, if the escalator moves up for long enough such that it is ensured the passenger had a successful trip that covered the distance. However, the behavior of the escalator has not been determined yet, since we still have not fixed the behavior of the system. Hence, there is no guaranteed upper bound on the time of travel that we can use.

However, thinking about the problem for another moment reveals that we also do not need to. The solution is that we have the guarantee that, if we move up the escalator for some sufficiently large amount of time continuously, than each passenger will be delivered at the top eventually, independently of the length and speed the escalator is moving with, and no matter of when the passenger decides to start the trip. Eventually, the passenger will be delivered to the top. This assumption is sufficient enough to make the specification realizable. Formally, we add the following two assumptions to the specification, one for each direction:

$$\square\big(\texttt{enter}_{\langle\texttt{bottom}\rangle} \wedge \diamondsuit(\texttt{move}_{\langle\texttt{up}\rangle}\,\mathcal{W}\,\texttt{exit}_{\langle\texttt{top}\rangle}) \rightarrow \diamondsuit\,\texttt{exit}_{\langle\texttt{top}\rangle}\big) \tag{A1}$$

$$\square\big(\texttt{enter}_{\langle\texttt{top}\rangle} \wedge \diamondsuit(\neg\,\texttt{move}_{\langle\texttt{up}\rangle}\,\mathcal{W}\,\texttt{exit}_{\langle\texttt{bottom}\rangle}) \rightarrow \diamondsuit\,\texttt{exit}_{\langle\texttt{bottom}\rangle}\big) \tag{A2}$$

Finished! The resulting specification is realizable and the synthesis produces a controller that works correctly. Just ask yourself: how much time would you have spend with constructing the respective controller by hand and completely on your own? Another exercise: What do you think is the minimal number of states required by a Mealy machine implementing this controller? Finally: What is easier? Creating the specification or the manual implementation?

As you see, synthesis offers some strong advantages against the manual creation of solutions, but it also comes with some new problems and challenges. For example, how does the synthesized controller work? How exactly does it archive the goal of delivering ever passenger. The answer may be given after inspection of the synthesized system. However, is there again the guarantee that this will always be an easy task?

# 7 Universal co-Büchi Automata

We conclude our preliminary definitions with one further specification model for infinite languages $L \subseteq \Sigma^\omega$ with $\Sigma = 2^{\mathcal{I}\cup\mathcal{O}}$: the model of universal co-Büchi word/tree automata. In general, there are various automata models that are

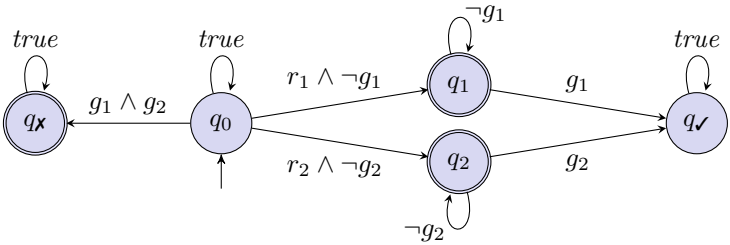



Figure 12: The universal co-Büchi automaton $\mathfrak{A}^e$, which expresses the language $\mathcal{L}(\varphi_2^e) = \mathcal{L}(\square(r_1 \rightarrow \diamondsuit g_1) \wedge \square(r_2 \rightarrow \diamondsuit g_2) \wedge \square\neg(g_1 \wedge g_2))$ for $n = 2$.

capable of expressing languages over infinite words or trees. Universal co-Büchi automata, however, enjoy the special property that their word and tree models are so close to each other that it is just a matter of the formal definition to distinguish them. The reasons are twofold. On the one hand, universal co-Büchi automata are free of nondeterministic choices. Thus, the acceptance of infinite trees only depends on the infinite behavior of all branches and not on how these branches have been chosen nondeterministically during the execution of time. On the other hand, the environment induces a universal branching over input propositions $\mathcal{I}$, similar to the interpretation of LTL properties over trees. Therefore, we only consider the word model of universal co-Büchi automata, since they already cover all relevant tree properties.

**Definition 9.** A *universal co-Büchi word automaton* $\mathfrak{A}$ is a tuple $\mathfrak{A} = (\Sigma, Q, q_I, \Delta_\mathfrak{A}, \textsc{coBüchi}(R))$, where

- $\Sigma$ is a finite alphabet,
- $Q$ is a finite set of states,
- $q_I \in Q$ is the initial state,
- $\Delta_\mathfrak{A} \subseteq Q \times \Sigma \times Q$ is the transition relation,
- $R \subseteq Q$ is the set of rejecting states,

where the set $\textsc{coBüchi}(X) \subseteq (Q \times \Sigma)^\omega$ fixes the accepting sequences.

$$\textsc{coBüchi}(X) = \{(q_0, \sigma_0)(q_1, \sigma_1) \ldots \in (Q \times \Sigma)^\omega \mid \exists n \in \mathbb{N}.\ \forall j > n.\ q_j \notin X\}$$

An example for a universal co-Büchi word automaton is depicted in Figure 12.

Its language is equivalent to $\mathcal{L}(\varphi_2^e)$ expressing a simple arbiter with two clients ($n = 2$). The automaton is illustrated as a labeled directed graph, where states are nodes and the transitions are edges. The initial state is marked by an arrow without a source. Rejecting states are marked through doubly framed nodes. For the representation of the transition relation, we utilize an alternative representation of $\Delta_{\mathfrak{A}}$ as a function $\Delta_{\mathfrak{A}} \colon Q^2 \to \mathbb{B}^*(\mathcal{I} \cup \mathcal{O})$, which is only applicable if $\Sigma = 2^{\mathcal{I} \cup \mathcal{O}}$. The representation assigns each pair of states a Boolean formula over the input and output propositions such that for all $q, q' \in Q$ and $\sigma \in 2^{\mathcal{I} \cup \mathcal{O}}$ we have that $\sigma \vDash \Delta_{\mathfrak{A}}(q, q')$ if and only if $(q, \sigma, q') \in \Delta$. For the illustration of universal co-Büchi word automata, we label the edges with $\Delta_{\mathfrak{A}}(q, q')$, as long as the formula is satisfiable. Otherwise, there is not edge depicted between two states. For the sake of readability, we use $\Delta_{\mathfrak{A}}$ for both: the function and relational representation, since both notions can always be clearly distinguished with respect to their application context.

The size of universal co-Büchi automaton $\mathfrak{A}$ is denoted by $|\mathfrak{A}|$ and defined to be $|Q|$. A run $r = (q_0, \sigma_0)(q_1, \sigma_1) \ldots \in (Q \times \Sigma)^\omega$ of $\mathfrak{A}$ is an infinite sequence with $q_0 = q_I$ and $(q_n, \sigma_n, q_{n+1}) \in \Delta_{\mathfrak{A}}$ for all $n \in \mathbb{N}$. The word language $\mathcal{L}(\mathfrak{A})$ contains all $\sigma_0 \sigma_1 \ldots \in \Sigma^\omega$, for which all runs $r = (q_0, \sigma_0)(q_1, \sigma_1) \ldots \in (Q \times \Sigma)^\omega$ with $q_0 q_1 \ldots \in Q^\omega$ are accepting, *i.e.*, for which $r \in \textsc{coBüchi}(R)$.

Every LTL formula $\varphi$ can be converted into an equivalent universal co-Büchi automaton expressing the same language as $\varphi$.

**Theorem 1** ([90])**.** *For every LTL formula $\varphi$ there is a universal co-Büchi word automaton $\mathfrak{A}_\varphi$ with $\mathcal{L}(\varphi) = \mathcal{L}(\mathfrak{A}_\varphi)$ and $|\mathfrak{A}_\varphi| \in O(2^{|\varphi|})$.*

Given a machine $\mathbb{M}$, it is easy to check whether $\mathbb{M}$ satisfies an infinite behavior property given as universal co-Büchi automaton $\mathfrak{A}$. To this end, we have to create the run graph consisting of the cross-product of $\mathbb{M}$ and $\mathfrak{A}$ to check that no execution of $\mathbb{M}$ visits infinitely many rejecting states of $\mathfrak{A}$.

**Definition 10.** The *run graph* $G_{\mathfrak{A},\mathbb{M}} = (M \times Q, E)$ of a universal co-Büchi automaton $\mathfrak{A} = (2^{\mathcal{I} \cup \mathcal{O}}, Q, q_I, \Delta_{\mathfrak{A}}, \textsc{coBüchi}(R))$ and a Mealy machine $\mathbb{M} = (2^{\mathcal{I}}, 2^{\mathcal{O}}, M, m_I, \delta_{\mathbb{M}}, \ell)$ is a directed graph, where

$$E = \{((m, q), (m', q')) \mid \exists \sigma \in 2^{\mathcal{I} \cup \mathcal{O}}.\ \delta_{\mathbb{M}}(m, \mathcal{I} \cap \sigma) = m' \wedge \ell(m, \mathcal{I} \cap \sigma) = \mathcal{O} \cap \sigma \wedge (q, \sigma, q') \in \Delta_{\mathfrak{A}}\}.$$

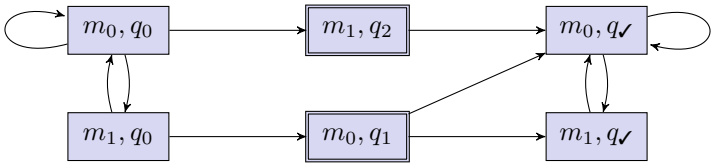



Figure 13: Run graph of the Mealy machine $\mathbb{M}^e$ of Figure 11 and the universal co-Büchi automaton $\mathfrak{A}^e$ of Figure 12.

A vertex $(m, q)$ of $G_{\mathfrak{A},\mathbb{M}}$ is rejecting iff $q \in R$. A run graph is considered to be accepting iff there is no cycle of $G_{\mathfrak{A},\mathbb{M}}$, which contains a rejecting vertex.

An example for a run graph for the Mealy machine $\mathbb{M}^e$ of Figure 11 and the universal co-Büchi automaton $\mathfrak{A}^e$ of Figure 12 is depicted in Figure 13. The rejecting vertices are marked by doubly framed nodes. Only the fraction of the run graph that is reachable from the initial vertex $(m_0, q_0)$ is illustrated.

In general, the non-existence of a rejecting vertex is witnessed by a ranking function $\lambda\colon M \times Q \to \mathbb{N} \cup \{\_\}$ with $\_ \notin \mathbb{N}$. If the run graph contains no rejecting vertex, then there is a ranking that labels all vertices that are reachable from $(m_I, q_I)$ with a natural number, which never increases among the transition relation and strictly decreases after each rejecting vertex.

**Definition 11.** A ranking $\lambda\colon M \times Q \to \mathbb{N} \cup \{\_\}$ *validates* a run graph $G_{\mathfrak{A},\mathbb{M}}$ with $\mathfrak{A} = (2^{\mathcal{I}\cup\mathcal{O}}, Q, q_I, \Delta_{\mathfrak{A}}, \textsc{coBüchi}(R))$ and $\mathbb{M} = (2^{\mathcal{I}}, 2^{\mathcal{O}}, M, m_I, \delta_{\mathbb{M}}, \ell)$ if and only if:

- $\lambda(m_I, q_I) \in \mathbb{N}$
- $\forall v, v' \in M \times Q.\ \lambda(v) \in \mathbb{N} \wedge (v, v') \in E \to \lambda(v') \in \mathbb{N} \wedge \lambda(v') \leq \lambda(v)$
- $\forall v, v' \in M \times R.\ \lambda(v) \in \mathbb{N} \wedge (v, v') \in E \to \lambda(v') \in \mathbb{N} \wedge \lambda(v') < \lambda(v)$

The existence of a valid ranking proves that the run graph is accepting. If the run graph is accepting, we also say that $\mathbb{M}$ is accepted by $\mathfrak{A}$.

**Theorem 2** ([45])**.** *Let $\mathfrak{A}$ and $\mathbb{M}$ be a universal co-Büchi automaton and a Mealy machine, respectively, with compatible inputs and outputs. Then $\mathcal{L}(\mathbb{M}) \subseteq \mathcal{L}(\mathfrak{A})$ if and only if there is a ranking $\lambda$ that validates $G_{\mathfrak{A},\mathbb{M}}$.*

A ranking that validates the run graph of Figure 13 is for example $\lambda^e$ with:

$$\lambda^e(m,q) = \begin{cases} 1 & \text{if } q \in \{q_0, q_1\} \\ 0 & \text{if } q \in \{q_{\checkmark}\} \\ \_ & \text{otherwise} \end{cases}$$

Thus, the ranking $\lambda^e$ proves that $\mathbb{M}^e$ satisfies $\varphi_2^e$ for $n = 2$ clients.

# Chapter III

# Temporal Stream Logic

Analyzing the design requirements of reactive applications in the real world reveals that Boolean and temporal reasoning alone is not enough to construct scalable reactive systems. Instead, it turns out that real-world applications not only need to coordinate the transformation of a few data bits, but require complex data structures that must be scalable according to the users' desires. The problem that we encounter with classical specification logics like LTL is that they are limited to data instances that only can be expressed by the composition of a few explicit data bits. Thus, they miss required abstractions for keeping data entities scalable.

Our solution to the problem is the clean separation between the reactive program's control, affecting its execution behavior, and the data that is processed continuously. To this end, we abstract from the actual data representation and instead focus on the behavior description of the system's high-level control. Immediate advantages are *scalability*, because data representations no longer need to be concretized, and *modularization*, since the data can be instantiated independently of the control. Furthermore, a strict separation improves the quality of the systems under design, since developers are forced to keep a clean separation between the behavior description and data properties, instead of mixing them in the design specification implicitly.

## 1 The Logic

In this chapter, we lay the foundations for describing reactive system behavior with a specification logic that explicitly separates data from control. We present: Temporal Stream Logic (TSL), a specification language that is especially designed for reactive system synthesis. TSL is the first temporal logic that formalizes reactive behavior over infinite data streams of arbitrary, non-enumerative, and higher order type. To this end, the logic utilizes a straightforward notation for specifying how outputs are computed from inputs and uses Boolean and temporal connectives for an intuitive interface to standard logical reasoning and for accessing time. Therefore, TSL focuses on describing temporal control flow, while universally abstracting from possible

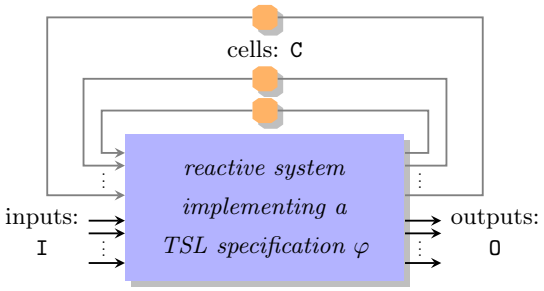



Figure 14: The reactive system architecture that is implicitly utilized by TSL.

data representations. As a consequence, the logic is not only intuitive and simple, but also allows developers to identify issues with the control flow, even without a concrete implementation of data instances and their possible transformations at hand. Accordingly, TSL scales up to any required data abstraction, as well as to complex functional transformations or API calls.

## 1.1 Architecture

Every TSL formula $\varphi$ describes a reactive system component processing a finite set of input streams `I` and producing a finite set of output streams `O`. Additionally, cells `C` can be used to store values computed at time $t$ for reusing them at the next time step $t + 1$. An overview of the resulting architecture model is presented in Figure 14. According to this model, the environment produces infinite streams of unconstrainted input data, while the system uses pure and side-effect free functional transformations to manipulate the values of the input streams at every point in time. After traversing a sequence of function transformations, data values are either output or stored in cells, where cells are used to provide the stored values again as inputs at the next moment in time. The final component's behaviour then results from its infinite execution over time.

## 1.2 Updates, Function, and Predicate Terms

In order to process infinite data streams of arbitrary type, the logic differentiates between two basic building blocks: purely functional transformations, reflected by functions $f \in \mathcal{F}$ and their compositions, as well as predicates $p \in \mathcal{P}$, used to access data properties of the inputs and cells, which then are projected to Boolean decisions that control the temporally evolving data flow inside the

system. Furthermore, functions and predicates are composable to new joint transformations. To this end, we leverage a term based notion that separates between function terms $\tau_F$ and predicate terms $\tau_P$, respectively. Function terms

$$\tau_F \;:=\; \mathtt{s_i} \;\mid\; \mathtt{f}\ \tau_F\ \tau_F\ \cdots\ \tau_F$$

either utilize inputs or cells $\mathtt{s_i} \in \mathtt{I} \cup \mathtt{C}$, without any prior conversion, or first apply pure function transformations $\mathtt{f}$ in composition with other function terms, *i.e.*, pre-process input streams before they are passed as arguments to the joining function. Function terms are of an arbitrary type. Predicate terms

$$\tau_P \;:=\; \mathtt{s_i} \;\mid\; \mathtt{p}\ \tau_F\ \tau_F\ \cdots\ \tau_F$$

are constructed similarly, but instead are of Boolean type. Note that the term syntax uses curried argument notation similar to functional programming languages. Finally, an update $[\,\mathtt{s_o} \leftarrowtail \tau_F\,]$ takes the result of a composed function transformation $\tau_F$ and passes it either to an output or a cell $\mathtt{s_o} \in \mathtt{O} \cup \mathtt{C}$.

We denote sets of function terms, predicate terms, or updates by $\mathcal{T}_F$, $\mathcal{T}_P$ and $\mathcal{T}_\lhd$, respectively, where $\mathcal{T}_P \subseteq \mathcal{T}_F$. Furthermore, we use $\mathtt{F}$ to denote the set of function literals and $\mathtt{P} \subseteq \mathtt{F}$ to denote the set of predicate literals, where the literals $\mathtt{s_i}$, $\mathtt{s_o}$, $\mathtt{f}$ and $\mathtt{p}$ are symbolic representations of inputs and cells, outputs and cells, functions, or predicates, respectively. The arity of function and predicate literals $\mathtt{f}$ is accessed by $\sharp(\mathtt{f})$, similar to the functions themselves. Note that, as a consequence of the utilized symbolic representation, functions and predicates are not tied to a specific implementation. However, we still classify them according to their arity, i.e., the number of function terms they are applied to, as well as by their type: input, output, cell, function, or predicate. To give functions and predicates some semantics, an assignment function $\langle\cdot\rangle\colon \mathtt{F} \to \mathcal{F}$ is used that preserves the arity, *i.e.*, $\forall \mathtt{f} \in \mathtt{F}.\ \sharp(f) = \sharp(\langle\mathtt{f}\rangle)$. For comparing two function or predicate terms syntactically, *i.e.*, according to the abstract syntax tree that is induced through the used function and predicate literals, we use the syntactic equivalence relation $\equiv$.

## 1.3 Inputs, Outputs, and Computations

We consider momentary inputs of the environment as assignments $i \in \mathcal{V}^{[\mathtt{I}]}$ that map inputs $\mathtt{i} \in \mathtt{I}$ to values $v \in \mathcal{V}$. For the sake of readability, we use $\mathcal{I} \mathrel{\hat{=}} \mathcal{V}^{[\mathtt{I}]}$. Input streams then are infinite sequences $\iota \in \mathcal{I}^\omega$ consisting of infinitely many momentary inputs. Similarly, momentary outputs of the system are assignments $o \in \mathcal{V}^{[\mathtt{O}]}$ of outputs $\mathtt{o} \in \mathtt{O}$ to values $v \in \mathcal{V}$, where we also use $\mathcal{O} \mathrel{\hat{=}} \mathcal{V}^{[\mathtt{O}]}$. Output streams are infinite sequences $\varrho \in \mathcal{O}^\omega$.

To capture the behavior of cells, we introduce the notion of computations $\varsigma$. Computations fix the function terms used to compute outputs and cell updates, without fixing semantics of function literals. In other words, computations determine the function terms that are executed at a time instance to compute output values and the values that are stored in cells. The momentary element of a computation is a computation step $c \in \mathcal{T}_F^{[\mathtt{O} \cup \mathtt{C}]}$ assigning outputs and cells $\mathtt{s_o} \in \mathtt{O} \cup \mathtt{C}$ to function terms $\tau_F \in \mathcal{T}_F$. For the sake of readability let $\mathcal{C} \mathrel{\hat{=}} \mathcal{T}_F^{[\mathtt{O} \cup \mathtt{C}]}$. A computation step fixes the control flow behaviour at a time instance, while a computation $\varsigma \in \mathcal{C}^\omega$ is an infinite sequence of computation steps. As soon as input streams, and function and predicate implementations are known, computations can be turned into concrete output streams. To this end, let $\langle\cdot\rangle \colon \mathtt{F} \to \mathcal{F}$ be some function assignment, where we assume that there are constant literals $\mathtt{init_c} \in \mathtt{F}$ for every cell $\mathtt{c} \in \mathtt{C}$ that provide initial values at the start of time and do not appear in any formula $\varphi$. To evaluate the values of output streams from computations $\varsigma \in \mathcal{C}^\omega$ under the input $\iota$ and implementations $\langle\cdot\rangle$, we use the evaluation function $\eta_{\langle\cdot\rangle}$.

**Definition 12.** The *evaluation function* $\eta_{\langle\cdot\rangle} \colon \mathcal{C}^\omega \times \mathcal{I}^\omega \times \mathbb{N} \times \mathcal{T}_F \to \mathcal{V}$ determines the value of a function term under a computation $\varsigma$ and input $\iota$ at time $t \in \mathbb{N}$:

$$\eta_{\langle\cdot\rangle}(\varsigma, \iota, t, \mathtt{s_i}) \quad := \quad \begin{cases} \iota(t)(\mathtt{s_i}) & \text{if } \mathtt{s_i} \in \mathtt{I} \\ \langle \mathtt{init_{s_i}} \rangle & \text{if } \mathtt{s_i} \in \mathtt{C} \ \wedge \ t = 0 \\ \eta_{\langle\cdot\rangle}(\varsigma, \iota, t-1, \varsigma(t-1)(\mathtt{s_i})) & \text{if } \mathtt{s_i} \in \mathtt{C} \ \wedge \ t > 0 \end{cases}$$

$$\eta_{\langle\cdot\rangle}(\varsigma, \iota, t, \mathtt{f}\ \tau_0\ \cdots\ \tau_{m-1}) \quad := \quad \langle \mathtt{f} \rangle\ \eta_{\langle\cdot\rangle}(\varsigma, \iota, t, \tau_0)\ \cdots\ \eta_{\langle\cdot\rangle}(\varsigma, \iota, t, \tau_{m-1})$$

Consequently, output streams then result from evaluating $\eta_{\langle\cdot\rangle}$ on inputs and the accumulated cell contents at every point in time, *i.e.*, $\varrho_{\langle\cdot\rangle,\varsigma,\iota} \in \mathcal{O}^\omega$ is defined via $\varrho_{\langle\cdot\rangle,\varsigma,\iota}(t)(\mathtt{o}) = \eta_{\langle\cdot\rangle}(\varsigma, \iota, t, \varsigma(t)(\mathtt{o}))$ for all $t \in \mathbb{N}$ and $\mathtt{o} \in \mathtt{O}$.

## 1.4 Syntax

**Definition 13.** Every TSL formula $\varphi$ is build according to the grammar:

$$\varphi \quad := \quad \mathit{true} \ \mid \ \tau_P \ \mid \ [\mathtt{s_o} \leftarrowtail \tau_F] \ \mid \ \neg\varphi \ \mid \ \varphi \wedge \varphi \ \mid \ \bigcirc\varphi \ \mid \ \varphi\, \mathcal{U}\, \varphi$$

The basic propositions are either predicate terms $\tau_P$, serving as the Boolean interface to the inputs, or updates $[\,\mathtt{s}_\circ \leftarrowtail \tau_F\,]$, enforcing the expressed flow at the current point in time. The remaining operators are standard for a temporal logic. We have the Boolean operations via negation and conjunction, which allow to express arbitrary Boolean combinations of predicate evaluations and updates. Furthermore, we have the temporal operator $\bigcirc\psi$ (*next*) stating that the behavior expressed by the sub-formula $\psi$ must be realized at the next point in time and the temporal operator $\vartheta\,\mathcal{U}\,\psi$ (*until*), which enforces a property $\vartheta$ to hold until the property $\psi$ holds, where $\psi$ must hold at some future point in time eventually.

## 1.5 Semantics

TSL formulas are semantically evaluated under the scope of a given function implementation $\langle\cdot\rangle\colon \mathtt{F} \to \mathcal{F}$, an input stream $\iota \in \mathcal{I}^\omega$, and a computation $\varsigma \in \mathcal{C}^\omega$ that has been selected by the system.

**Definition 14.** Let $\langle\cdot\rangle\colon \mathtt{F} \to \mathcal{F}$, $\iota \in \mathcal{I}^\omega$, and $\varsigma \in \mathcal{C}^\omega$ be given, then the validity of a TSL formula $\varphi$ with respect to $\varsigma$ and $\iota$ is defined inductively over $t \in \mathbb{N}$ via:

$$
\begin{aligned}
\varsigma, \iota, t \;&\vDash_{\langle\rangle}\; \mathit{true} && :\Leftrightarrow \quad \mathit{true}\\
\varsigma, \iota, t \;&\vDash_{\langle\rangle}\; \mathtt{p}\;\tau_0\;\ldots\;\tau_{m-1} && :\Leftrightarrow \quad \eta_{\langle\rangle}(\varsigma, \iota, t, \mathtt{p}\;\tau_0\;\ldots\;\tau_{m-1})\\
\varsigma, \iota, t \;&\vDash_{\langle\rangle}\; [\,\mathtt{s} \leftarrowtail \tau\,] && :\Leftrightarrow \quad \varsigma(t)(\mathtt{s}) \equiv \tau\\
\varsigma, \iota, t \;&\vDash_{\langle\rangle}\; \neg\psi && :\Leftrightarrow \quad \varsigma, \iota, t \;\nvDash_{\langle\rangle}\; \psi\\
\varsigma, \iota, t \;&\vDash_{\langle\rangle}\; \vartheta \wedge \psi && :\Leftrightarrow \quad \varsigma, \iota, t \;\vDash_{\langle\rangle}\; \vartheta \;\wedge\; \varsigma, \iota, t \;\vDash_{\langle\rangle}\; \psi\\
\varsigma, \iota, t \;&\vDash_{\langle\rangle}\; \bigcirc\psi && :\Leftrightarrow \quad \varsigma, \iota, t+1 \;\vDash_{\langle\rangle}\; \psi\\
\varsigma, \iota, t \;&\vDash_{\langle\rangle}\; \vartheta\,\mathcal{U}\,\psi && :\Leftrightarrow \quad \exists t'' \geq t.\; \forall t \leq t' < t''.\\
& && \qquad \varsigma, \iota, t' \;\vDash_{\langle\rangle}\; \vartheta \;\wedge\; \varsigma, \iota, t'' \;\vDash_{\langle\rangle}\; \psi
\end{aligned}
$$

Consider that the satisfaction of predicates depends on the current computation step and the steps of the past, while the satisfaction of updates only depends on the current computation step. Furthermore, updates are only checked syntactically, while the satisfaction of predicates depends on the given assignment $\langle\cdot\rangle$ and the input stream $\iota$. We say that $\varsigma$ and $\iota$ satisfy $\varphi$, denoted by $\varsigma, \iota \;\vDash_{\langle\rangle}\; \varphi$, if and only if $\varsigma, \iota, 0 \;\vDash_{\langle\rangle}\; \varphi$.

Beside the aforementioned default operators we have the standard derived Boolean operators, such as *false*, disjunction $\vee$, implication $\rightarrow$, and equivalence $\leftrightarrow$, and the derived temporal operators *release* $\varphi\,\mathcal{R}\,\psi \mathrel{\hat{=}} \neg((\neg\psi)\,\mathcal{U}(\neg\varphi))$,

*finally* $\Diamond\varphi \mathrel{\hat{=}} \mathit{true}\,\mathcal{U}\,\varphi$, *always* $\Box\varphi \mathrel{\hat{=}} \mathit{false}\,\mathcal{R}\,\varphi$, the *weak* version of *until* $\varphi\,\mathcal{W}\,\psi \mathrel{\hat{=}} (\varphi\,\mathcal{U}\,\psi) \vee (\Box\varphi)$, and *as soon as* $\varphi\,\mathcal{A}\,\psi \mathrel{\hat{=}} \neg\psi\,\mathcal{W}(\psi \wedge \varphi)$.

### 1.6 Realizability

With syntax and semantics of TSL at hand, we are ready to precisely formalize the realizability problem of the logic.

**Problem 1.** Given a TSL formula $\varphi$, is there a strategy $\sigma\colon (2^{\mathcal{T}_P})^+ \to \mathcal{C}$ such that for every input $\iota \in \mathcal{I}^\omega$ and function implementation $\langle\cdot\rangle\colon \mathtt{F} \to \mathcal{F}$, the output that results according to $\sigma$ satisfies $\varphi$? Or formally:

$$\exists\sigma\colon (2^{\mathcal{T}_P})^+ \to \mathcal{C}.\ \forall\iota \in \mathcal{I}^\omega.\ \forall\langle\cdot\rangle\colon \mathtt{F} \to \mathcal{F}.\ \sigma \between_{\langle\cdot\rangle} \iota, \iota \vDash_{\langle\cdot\rangle} \varphi$$

where $\sigma \between_{\langle\cdot\rangle} \iota = \varsigma$ is the unique computation $\varsigma \in \mathcal{C}^\omega$ such that:

$$\forall t \in \mathbb{N}.\ \varsigma_t = \sigma(\{\tau \in \mathcal{T}_P \mid \eta_{\langle\cdot\rangle}(\varsigma, \iota, 0, \tau)\} \ldots \{\tau \in \mathcal{T}_P \mid \eta_{\langle\cdot\rangle}(\varsigma, \iota, t, \tau)\})$$

If a strategy $\sigma$ exists, we say that $\sigma$ realizes $\varphi$. If we additionally ask for a concrete finitely representable instantiation of $\sigma$, we consider the synthesis problem of TSL.

## 2 Specification Examples

Before we continue, we demonstrate the advantages of TSL as a specification language for the development of reactive systems on two real-world example applications. Our first example reconsiders the kitchen timer application of the introduction, for which specifying the reactive behavior revealed to be a tedious task, if tried to be done with LTL. Moreover, we had identified that such an LTL specification would be out of scope for every currently available synthesis tool. We show that TSL easily outperforms LTL in such a case, by utilizing the now available abstractions that we identified as missing for LTL.

Our second application targets a music player app that needs to be implemented in the environment of the Android operating system, a system that is deployed worldwide on a large scale of different mobile phones. Regarding the requirements of the app, we compare the design challenge of creating a TSL specification and using synthesis with the task of manually creating the application using classical iterative development techniques instead.

## 2.1 Specifying a Kitchen Timer

With the formal semantics of TSL at hand, we reconsider the kitchen timer application of Chapter I. Remember: the timer utilizes three buttons, a screen for displaying the currently set time, and a buzzer for producing an alarm. For the sake of simplicity, we consider the button values to be provided as Boolean input streams, which deliver *true*, as long as a button is pressed, and *false* otherwise. Furthermore, we assume that there is another input stream providing the time since the last evaluated moment, used to synchronize the displayed time with the underlying clock of the application framework.

Similar to the button inputs, the system outputs a Boolean data stream for controlling the buzzer, which emits an alarm whenever the output is `true` and keeps silent otherwise. At the same time, the system outputs a time value in a format that is suitable to be displayed on the screen. Note that the concrete representation of this time value depends on the application framework, and, thus, is not concretized as part of the TSL specification.

The required properties of the specification can be easily formalized using TSL. To this end, we start by fixing possible operations on `time` – a cell that holds the time that has been configured by the user.

$$\begin{aligned}
\text{COUNTUP} &:= [\texttt{time} \leftarrowtail \texttt{countup time dt}] \\
\text{COUNTDOWN} &:= [\texttt{time} \leftarrowtail \texttt{countdown time dt}] \\
\text{INCMIN} &:= [\texttt{time} \leftarrowtail \texttt{incMinutes time}] \\
\text{INCSEC} &:= [\texttt{time} \leftarrowtail \texttt{incSeconds time}] \\
\text{IDLE} &:= [\texttt{time} \leftarrowtail \texttt{time}]
\end{aligned}$$

The updates on `time` use the literals `countup`, `countdown`, `incMinutes`, and `incSeconds`, which represent pure functions for updating the value of the cell with the correspondingly transformed input. For counting up or down, the input signal `dt` delivers the time difference since the last execution of the network. Note that the semantics of TSL ensure that these assignments to the same cell are mutually exclusive, i.e., it can never be the case that two of these operations are executed simultaneously at the same point time.

Next, we need to control the flow of the updates of `time`. To this end, we utilize a predicate that checks whether the time is currently set to zero or not

$$\text{ZERO} := \texttt{eq time zero}^{\triangledown}$$

where $\texttt{zero}^{\triangledown}$ is a constant of the same type as `time`[1]. Finally, we define some sub-properties, which are useful for expressing conditions that regularly used later. In our case these are

[1] We tag constant literals with $^{\triangledown}$ to make it easier to distinguish them from input literals.

$$
\begin{aligned}
\textsc{reset} &:= \texttt{Min} \wedge \texttt{Sec} \\
\textsc{counting} &:= \textsc{countup} \vee \textsc{countdown} \\
\textsc{anykey} &:= \mathit{press}\ \texttt{Min} \vee \mathit{press}\ \texttt{Sec} \vee \mathit{press}\ \texttt{StartStop} \\
\textsc{start} &:= \mathit{press}\ \texttt{StartStop} \wedge \neg \mathit{press}\ \texttt{Min} \wedge \neg \mathit{press}\ \texttt{Sec} \\
\textsc{start\&min} &:= \mathit{press}\ \texttt{StartStop} \wedge \mathit{press}\ \texttt{Min} \wedge \bigcirc \neg \texttt{Sec} \wedge \bigcirc\bigcirc \neg \texttt{Sec} \\
\textsc{start\&sec} &:= \mathit{press}\ \texttt{StartStop} \wedge \mathit{press}\ \texttt{Sec} \wedge \bigcirc \neg \texttt{Min} \wedge \bigcirc\bigcirc \neg \texttt{Min}
\end{aligned}
$$

where the literals `Min`, `Sec`, and `StartStop` represent the input signals for the three buttons, respectively. The function *press* is used for providing a more compact notion and for improved readability. It is defined as:

$$\mathit{press}\ x \quad := \quad \neg\, x \ \wedge \ \bigcirc x$$

The last two conditions START&MIN and START&SEC are required for realizing requirement 8.

Now we are ready to reformulate all properties of the kitchen timer, that we informally introduced during the introduction, but this time with TSL:

$$
\begin{aligned}
\psi_1 &:= \textsc{reset} \leftrightarrow [\,\texttt{time} \leftarrowtail \texttt{zero}^{\triangledown}\,] \\
\psi_2 &:= \neg\textsc{counting} \wedge \mathit{press}\ \texttt{Min} \wedge \bigcirc \neg \texttt{Sec} \leftrightarrow \bigcirc\, \textsc{incmin} \\
\psi_3 &:= \neg\textsc{counting} \wedge \mathit{press}\ \texttt{Sec} \wedge \bigcirc \neg \texttt{Min} \leftrightarrow \bigcirc\, \textsc{incsec} \\
\psi_4 &:= \textsc{zero} \rightarrow \\
&\quad \big( (\,\textsc{idle} \wedge \textsc{start} \\
&\qquad \rightarrow \bigcirc\, \mathit{tillAnyInput}\ \textsc{countup})\, \mathcal{W}\, (\textsc{incmin} \vee \textsc{incsec}) \big) \\
\psi_5 &:= \textsc{incmin} \vee \textsc{incsec} \rightarrow \\
&\quad \big( (\neg\textsc{counting} \wedge \textsc{start} \\
&\qquad \rightarrow \bigcirc\, \mathit{tillAnyInput}\ \textsc{countdown})\, \mathcal{W}\, \bigcirc\, \textsc{zero} \big) \\
\psi_6 &:= \bigcirc \neg\textsc{counting} \wedge \bigcirc\bigcirc \textsc{counting} \\
&\quad \rightarrow \bigcirc \textsc{start} \vee \textsc{start\&min} \vee \textsc{start\&sec} \\
\psi_7 &:= \textsc{counting} \wedge \textsc{anykey} \wedge \bigcirc \neg\textsc{reset} \\
&\quad \rightarrow \bigcirc\, \mathit{tillAnyInput}\ \textsc{idle} \\
\psi_8 &:= \neg\textsc{counting} \wedge (\textsc{start\&min} \vee \textsc{start\&sec}) \\
&\quad \rightarrow \bigcirc\bigcirc\, \mathit{tillAnyInput}\ \textsc{countdown} \\
\psi_9 &:= ([\,\texttt{beep} \leftarrowtail \texttt{true}^{\triangledown}\,] \vee [\,\texttt{beep} \leftarrowtail \texttt{false}^{\triangledown}\,]) \wedge \\
&\quad \big( \bigcirc\, (\textsc{countdown} \wedge \textsc{zero}) \vee \textsc{anykey} \leftrightarrow \bigcirc [\,\texttt{beep} \leftarrowtail \texttt{true}^{\triangledown}\,] \big) \\
\psi_{10} &:= [\,\texttt{screen} \leftarrowtail \texttt{display time}\,]
\end{aligned}
$$

We again use a helping function *tillAnyInput* for improved readability, which denotes that a condition is satisfied until either the system is reset or any button gets pressed. The property can be expressed in TSL as follows:

$$\mathit{tillAnyInput}\ x \quad := \quad (x \wedge \neg \textsc{anykey})\ \mathcal{W}\ (\textsc{reset} \vee x \wedge \textsc{anykey})$$

The final specification is then composed as $\varphi = \psi_{init} \wedge \square \bigwedge_{i=1}^{10} \psi_i$, where $\psi_{init}$ adds some remaining startup conditions, which ensure that the system is initialized correctly according to the corresponding temporal context.

$$\begin{aligned} \psi_{init} \quad := \quad & \neg \textsc{counting} \wedge (\bigcirc \textsc{counting} \rightarrow \textsc{start}) \wedge \\ & \neg \textsc{incsec} \wedge \neg \textsc{incmin} \wedge [\,\texttt{beep} \leftharpoondown \texttt{false}^{\triangledown}\,] \end{aligned}$$

Also note that the $\bigcirc$-operations that have been used in the $\psi_i$ formulas are necessary, since "being pressed" requires a change of the Boolean input and this change is only observable by comparing the currently provided value with the previous one.

The definition of $\varphi$ provides the behavioral specification of the kitchen timer, where the compactness of $\varphi$ indeed proves that TSL overcomes the Boolean limitations of LTL that we observed in the introduction. Moreover, TSL features to argue about arbitrary data streams by utilizing the separation of data and control. This separation is required, because any representation of the time value, that needs to be displayed on the screen, is too complex to be encoded efficiently by a Boolean combination of binary values.

## 2.2 Specifying a Music Player

While the previous example demonstrates the advantages of TSL against classical logics, which are based on atomic propositions, it still might not be obvious which advantages synthesis provides against manual code creation. To this end, we also demonstrate the advantage of having the separation of different behavioral properties into a logical conjunction against iterative code development in the sense of feature extensions. We illustrate this difference on the development task of a simple Android music player app. The most time consuming and error-prone considerations during the development of Android apps are caused by the temporal behavior of the app through the *Android lifecycle* [133]. This lifecycle specifies how apps are paused, when moved to the background, come back into focus, or are terminated. Specifically, *resume and restart errors* are commonplace and hard to debug and correct.

Our music player example highlights a development situation, in which a resume and restart error could be introduced unintentionally when programming by hand, but is avoided when using a TSL specification. The app utilizes

```
Sys.leaveApp() {
  if (MP.musicPlaying()) {
    Ctrl.pause();
  }
}
Sys.resumeApp() {
  pos = MP.trackPos();
  Ctrl.play(Tr,pos);
}
```

$$\square\big(\texttt{leaveApp Sys} \wedge \texttt{musicPlaying MP} \rightarrow [\,\texttt{Ctrl} \leftharpoondown \texttt{pause}^{\triangledown}\,]\big)$$

$$\square\big(\texttt{resumeApp Sys} \rightarrow [\,\texttt{Ctrl} \leftharpoondown \texttt{play Tr (trackPos MP)}\,]\big)$$

Figure 15: On the left, the code that is used to interact with the system for leaving and resuming the app. On the right, the equivalent formulation in TSL.

the Android music player library (`MP`) in combination with its control interface (`Ctrl`). It pauses music, if playing and being moved to the background (for example if a call is received), and continues playing the currently selected track (`Tr`) at the stopped position, if the app is resumed. In the Android operating system (`Sys`), the `leaveApp` method is called whenever an app moves to the background, while the `resumeApp` method is called to resume. When the user leaves the app and the music is currently playing, then the music must be paused. Similarly, when the user resumes the player, then the music must be started again at the position it has been stopped before. Some code that implements this behavior can be found on the left of Figure 15. An equivalent TSL specification is placed on the right.

Now consider an extension of the app's functionality, which requires that it only starts playing after being resumed, if music was playing on the previous leave of the app before. For the manually developed code, such a change requires a new shared program variable `wasPlaying` keeping track of the condition under which the app has been left in the past. Therefore, this new variable must be correctly initialized, read and updated at the right places in the code, as illustrated on the right of Figure 16. The respective changes include an additional conditional in the `resumeApp` method that branches into two different update conditions for `wasPlaying`, respectively.

The example highlights how a small update of a minor requirement may lead to some wide-ranging code changes. Furthermore, for the application of these changes a developer with decent knowledge of the code base is necessary, who knows all the places that must be updated. Moreover, the introduced change requires a new globally scoped variable, which then might be changed unwittingly elsewhere.

On the other hand, changing the TSL specification is straightforward. Here, only updating a single sub-formula is required: *if the app is left while*

```
bool wasPlaying = false;
Sys.leaveApp() {
  if (MP.musicPlaying()) {
    wasPlaying = true;
    Ctrl.pause();
  }
  else {
    wasPlaying = false;
  }
}
Sys.resumeApp() {
  if (wasPlaying) {
    pos = MP.trackPos();
    Ctrl.play(Tr,pos);
  }
}
```

$$\square\big(\texttt{leaveApp Sys} \wedge \texttt{musicPlaying MP} \rightarrow [\,\texttt{Ctrl} \leftharpoondown \texttt{pause}^{\triangledown}\,]\big)$$

$$\square\big(\texttt{leaveApp Sys} \wedge \texttt{musicPlaying MP} \rightarrow [\,\texttt{Ctrl} \leftharpoondown \texttt{play Tr (trackPos MP)}\,] \; \mathcal{A}\; \texttt{resumeApp Sys}\big)$$

Figure 16: A minor change in functionality may require multiple code changes, but can be reflected by a small change in the TSL specification.

*music was playing, then music must play again as soon as the app resumes.*

In general, the development of formal specifications brings many advantages against manually created code. On the one hand, one obtains a precise description of the required system features, which are logically organized through Boolean and temporal connectives. On the other hand, implementation details can be postponed to the synthesis engine and later function and predicate implementations. Finally, through synthesis the created systems are correct by construction. Hence, potential errors are already discovered during development and can be fixed at an early design stage.

# 3 Decidability

Before we consider the details of our solution to the TSL synthesis problem, we examine some properties of the logic that arise as a consequence of the utilized universal quantification over function and predicate terms. The most constrictive implication of this quantification is that it renders the synthesis problem undecidable, since, in a nutshell, universally quantified predicates can be used to construct a path with two equal, but differently constructed cell elements using a diagonalization argument. In combination with the temporally proceeding evaluation over time, such two equal elements can be utilized to encode undecidable problems, as for example the Post correspondence problem (PCB) [118].

**Theorem 3.** *The realizability problem of TSL is undecidable.*

*Proof.* We reduce an instance of the Post Correspondence Problem (PCP) [118], consisting of an alphabet $\Sigma$ and sequences $w_0, w_1, \ldots, w_n, v_0, v_1, \ldots, v_n \in \Sigma^+$, to the realizability of a TSL formula $\varphi$. The PCP asks, whether there is some finite sequence $i_0 i_1 \ldots i_k \in \mathbb{N}^+$ such that $w_{i_0} w_{i_1} \ldots w_{i_k} = v_{i_0} v_{i_1} \ldots v_{i_k}$. The problem is well known to be undecidable.

To this end, we translate an arbitrary instance of PCP to a TSL formula $\varphi$, which is realizable if and only if there is a solution to the PCP instance. Let $n \in \mathbb{N}$, $w_0, w_1, \ldots, w_n \in \Sigma^*$ and $v_0, v_1, \ldots, v_n \in \Sigma^*$ be given. We fix predicate literals $\mathtt{P} = \{\mathtt{p}\}$, utilizing some unary predicate literal $\mathtt{p}$, function literals $\mathtt{F} = \Sigma \cup \{\mathtt{X}^\nabla\}$, with every $\mathtt{f} \in \Sigma$ corresponding to some unary function, and $\mathtt{X}^\nabla$, denoting some constant literal, $\mathtt{I} = \emptyset$, $\mathtt{O} = \emptyset$, and $\mathtt{C} = \{\mathtt{A}, \mathtt{B}\}$. Let $\mu(x_0 x_1 \ldots x_m, s) = x_0(x_1(\ldots x_m(s)\ldots))$, then we define $\varphi$ via:

$$
\begin{aligned}
\varphi \quad := \quad & \Big( [\, \mathtt{A} \leftharpoonup \mathtt{X}^\nabla \,] \;\wedge\; [\, \mathtt{B} \leftharpoonup \mathtt{X}^\nabla \,] \Big) \\
& \wedge\; \bigcirc \square \Big( \bigvee_{j=0}^{n} \big( [\, \mathtt{A} \leftharpoonup \mu(w_j, \mathtt{A}) \,] \wedge [\, \mathtt{B} \leftharpoonup \mu(v_j, \mathtt{B}) \,] \big) \Big) \\
& \wedge\; \bigcirc \bigcirc \diamond \Big( \mathtt{p}(\mathtt{A}) \leftrightarrow \mathtt{p}(\mathtt{B}) \Big)
\end{aligned}
$$

Intuitively, we first assign the cells $\mathtt{A}$ and $\mathtt{B}$ a constant base value. Then, from the next time step on, we have to pick pairs $(w_j, v_j)$ in every time step. Our choice is stored in the cells $\mathtt{A}$ and $\mathtt{B}$, respectively. Finally, we check the constructed sequences of function applications to be equal at some future time, where we use the universally quantified predicate $\mathtt{p}$ to check for equality. The TSL formula $\varphi$ is realizable if and only if there is an index sequence $i_0 i_1 \ldots i_k$ such that $w_{i_0} w_{i_1} \ldots w_{i_k} = v_{i_0} v_{i_1} \ldots v_{i_k}$:

"$\Rightarrow$": Assume $\varphi$ is realizable, i.e., there is some strategy $\sigma \colon (2^{\mathcal{T}_P})^+ \to \mathcal{C}$ that satisfies $\varphi$ for $\iota = \emptyset^\omega$ and all possible choices of $\langle\cdot\rangle \colon \mathtt{F} \to \mathcal{F}$. We claim that:

$$
\exists \alpha \in (2^{\mathcal{T}_P})^\omega.\; \exists t > 1.\; \eta_{\langle\cdot\rangle}(\sigma \,\wr\, \alpha, \iota, t, \mathtt{A}) = \eta_{\langle\cdot\rangle}(\sigma \,\wr\, \alpha, \iota, t, \mathtt{B}) \tag{1}
$$

For the sake of contradiction assume the contrary, *i.e.*, that $\mathtt{A}$ and $\mathtt{B}$ are always different on every branch after two steps. We fix $\langle \mathtt{X}^\nabla \rangle = \varepsilon$ to be the empty word and $\langle \mathtt{f} \rangle(w) = w \cdot \mathtt{f}$ to be functions for all $\mathtt{f} \in \Sigma$ that concatenate the letters $\mathtt{f} \in \Sigma$ to words $w \in \Sigma^*$. Correspondingly, the strategy $\sigma$ operates on sequences, which are stored in $\mathtt{A}$ and $\mathtt{B}$ and get enlarged over time. Moreover, the content of $\mathtt{A}$ and $\mathtt{B}$ is strictly increasing with every update, since $|w_k| > 0$

and $|v_k| > 0$ for all $k \in [n]$. Therefore, a simple induction shows that at every time step $t > 1$ either $\mathtt{A}$ or $\mathtt{B}$ contains a value $w \in \Sigma^+$ which never has been stored in $\mathtt{A}$ or $\mathtt{B}$ at some earlier time step $t' < t$. Let $w_2^{\mathtt{A}}, w_3^{\mathtt{A}}, \ldots \in \Sigma^*$ be the sequences stored in $\mathtt{A}$ and $w_2^{\mathtt{B}}, w_3^{\mathtt{B}}, \ldots \in \Sigma^*$ be the sequences stored in $\mathtt{B}$ for all $t > 1$. We choose $\langle \mathtt{p} \rangle$ such that it satisfies

$$\langle \mathtt{p} \rangle(w_j^{\mathtt{X}}) = \begin{cases} \neg\langle \mathtt{p} \rangle(w_j^{\mathtt{A}}) & \text{if } \mathtt{X} = \mathtt{B} \text{ and } |w_j^{\mathtt{A}}| \leq |w_j^{\mathtt{B}}| \text{ and } w_j^{\mathtt{A}} \neq w_j^{\mathtt{B}} \\ \neg\langle \mathtt{p} \rangle(w_j^{\mathtt{B}}) & \text{if } \mathtt{X} = \mathtt{A} \text{ and } |w_j^{\mathtt{A}}| > |w_j^{\mathtt{B}}| \end{cases}$$

A corresponding choice is always possible due to the first time appearance of one of the elements of $\mathtt{A}$ or $\mathtt{B}$ as mentioned before. Using the corresponding semantics $\langle\cdot\rangle$, it is easy to observe that on the branch $\sigma \wr\!\!\wr_{\langle\rangle} \iota$ there is no time $t > 2$, where $\mathtt{p}(\mathtt{A}) \leftrightarrow \mathtt{p}(\mathtt{B})$ as long as the content of $\mathtt{A}$ and $\mathtt{B}$ differs at every $t > 2$. However, this contradicts that $\sigma \wr\!\!\wr_{\langle\rangle} \iota, \iota \models_{\langle\rangle} \varphi$ and therefore proves the claim of Equation (1). Hence, if there is some time $t > 2$, at which $w_t^{\mathtt{A}} = w_t^{\mathtt{B}}$, then the updates of $\mathtt{A}$ and $\mathtt{B}$ on the branch $\sigma \wr\!\!\wr_{\langle\rangle} \iota$ chosen at times $0 < t' \leq t$ determine the indices $i_0 i_1 \ldots i_{t-1}$ such that $w_{i_0} w_{i_1} \ldots w_{i_{t-1}} = v_{i_0} v_{i_1} \ldots v_{i_{t-1}}$.

"$\Leftarrow$": Now, assume that there is a solution $i_0 i_1 \ldots i_k$ to the PCP instance. As $\mathtt{I} = \emptyset$, it suffices to construct the computation $\varsigma = c_0 c_1 \ldots$ played on all branches independently of the evaluations of $\mathtt{p}$ with $c_0(\mathtt{A}) = c_0(\mathtt{B}) = \mathtt{X}^\nabla$ and for all $t > 0$: $c_t(\mathtt{A}) = \mu(w_{(t-1) \bmod (k+1)}, \mathtt{A})$ and $c_t(\mathtt{B}) = \mu(w_{(t-1) \bmod (k+1)}, \mathtt{B})$. It is straightforward to see that $\varsigma$ satisfies $[\,\mathtt{A} \leftharpoonup \mathtt{X}^\nabla\,]$, $[\,\mathtt{B} \leftharpoonup \mathtt{X}^\nabla\,]$ and

$$\bigcirc \Box \Big( \bigvee_{j=0}^{n} \big( [\,\mathtt{A} \leftharpoonup \mu(w_j, \mathtt{A})\,] \wedge [\,\mathtt{B} \leftharpoonup \mu(v_j, \mathtt{B})\,] \big) \Big).$$

Thus, it only remains to argue that $\varsigma$ satisfies $\bigcirc\bigcirc\Diamond(\mathtt{p\ A} \leftrightarrow \mathtt{p\ B})$. To this end, let $j_0 j_1 \ldots = (i_0 i_1 \ldots i_k)^\omega$. Then a simple induction shows that

$$\varrho_{\langle\rangle,\varsigma,\iota}(t)(\mathtt{A}) = \eta_{\langle\rangle}(\varsigma, \iota, t, \mu(w_{j_0} w_{j_1} \ldots w_{j_{t-2}}, \mathtt{X}^\nabla))$$

and $\varrho_{\langle\rangle,\varsigma,\iota}(t)(\mathtt{B}) = \eta_{\langle\rangle}(\varsigma, \iota, t, \mu(v_{j_0} v_{j_1} \ldots v_{j_{t-2}}, \mathtt{X}^\nabla))$ for all $t > 1$, $\iota = \emptyset^\omega$, and all choices of $\langle\cdot\rangle$. Now, consider that especially for $t = k + 2$ we have that

$$w_{j_0} w_{j_1} \ldots w_{j_{t-2}} \;=\; w_{i_0} w_{i_1} \ldots w_{i_k} \;=\; v_{i_0} v_{i_1} \ldots v_{i_k} \;=\; v_{j_0} v_{j_1} \ldots v_{j_{t-2}}$$

and, thus, $\varrho_{\langle\rangle,\varsigma,\iota}(k+2)(\mathtt{A}) = \varrho_{\langle\rangle,\varsigma,\iota}(k+2)(\mathtt{B})$, independent of the choice of $\langle\cdot\rangle$. As this implies that $p(\varrho_{\langle\rangle,\varsigma,\iota}(k+2)(\mathtt{A})) = p(\varrho_{\langle\rangle,\varsigma,\iota}(k+2)(\mathtt{B}))$ for any unary predicate $p \in \mathcal{P}$, it proves that $\varsigma$ satisfies $\mathtt{p\ A} \leftrightarrow \mathtt{p\ B}$ at position $k+2$. Hence, the computation $\varsigma$ also satisfies $\bigcirc\bigcirc\Diamond(\mathtt{p\ A} \leftrightarrow \mathtt{p\ B})$, which finally concludes the proof. □

Undecidability, thus, results from the combination of multiple features of TSL: the ability to store and reuse values through cells, the possibility to construct two equal values through universally quantified predicates, and the features of temporal behavior allowing for the manipulation of data streams over time.

# 4 Fragments

The realizability problem of TSL being undecidable is an unpleasant result and leads to the natural desire of being avoided in the first place. A common technique for recovering decidability is the restriction of the logic to a decidable fragment, which is still expressive enough to cover all of the required considerations. Classically, such limitations are introduced on the winning conditions of the underlying synthesis games, but in case of TSL, it is also possible to limit the term expressions that are used as building blocks for constructing the specification. We consider both variants. However, for simplifications of the winning conditions, we do not instantiate them as term simplifications on the logic level, but instead consider restrictions of the winning conditions for the underlying games directly. The results reveal that restrictions of the term syntax are of little help, but restrictions on the winning conditions are a step forward towards solving the undecidability problem.

We start with restrictions on the term expressions, where we show that most of the natural limitations do not affect expressivity, and, thus, keep the undecidability result. Hence, while being of little help for improving the situation in terms of practical considerations, the results still help us to achieve a better understanding of the core capabilities of TSL. We consider a chain of realizability preserving transformations that convert a given TSL formula to a simplified fragment of the logic. This fragment, however, still is expressive enough to cover all of the original specification's intentions. The simplifications that we consider are (1) the restriction to a single unary predicate literal $\mathtt{p}$, and (2) the restriction to a single binary function literal $\mathtt{f}$. We call the resulting fragment $\mathrm{TSL}^{\mathtt{p}}_{\mathtt{f}}$ (pronounced TSL-f-p) and show that it is equally expressive to full TSL in terms of realizability.

**Definition 15.** The syntax of $\mathrm{TSL}^{\mathtt{p}}_{\mathtt{f}}$ is formally defined as follows:

$$\varphi \;:=\; \mathit{true} \mid \mathtt{p}\,\hat{\tau_F} \mid [\mathtt{s_o} \leftharpoondown \hat{\tau_F}] \mid \neg\varphi \mid \varphi \wedge \varphi \mid \bigcirc\varphi \mid \varphi\,\mathcal{U}\,\varphi$$

$$\text{where } \hat{\tau_F} \;:=\; \mathtt{s}_i \mid \mathtt{f}\;\hat{\tau_F}\;\hat{\tau_F}\;.$$

Semantically, $\mathrm{TSL}^{\mathtt{p}}_{\mathtt{f}}$ formulas are evaluated the same as full TSL.

We show that both logics are equally expressive through a step-wise reduction from TSL to $\text{TSL}_{\mathtt{f}}^{\mathtt{p}}$ (the other direction is trivial). To this end, we first restrict different predicate literals, appearing in the formula, to a single predicate literal $\mathtt{p}$. Afterwards, we eliminate function literals with strictly more or less than two arguments (except constant literals). Then, we eliminate different function literals and reduce them to the single literal $\mathtt{f}$. Finally, we eliminate constant literals. For every step, realizability is preserved. Nevertheless, the underlying system architecture may change according to the number of used cells, as well as the utilized function and predicate literals.

**Theorem 4.** *For every TSL formula $\varphi$ there exists an equi-realizable $\text{TSL}_{f}^{\mathtt{p}}$ formula $\varphi'$ that can be constructed from $\varphi$ in linear time in $|\varphi|$.*

*Proof.* For the proof, we use $\mathtt{P}_\varphi$, $\mathtt{F}_\varphi$, and $\mathtt{C}_\varphi$ to denote the sets of predicates, functions, and cells that are used in a formula $\varphi$, respectively.

We convert $\varphi$ to $\varphi'$ via a five-step transformation. In the first step, we reduce different predicate literals to a single predicate literal $\mathtt{p}$. The resulting fragment is denoted by $\text{TSL}^{\mathtt{p}}$.

**TSL $\rightarrow$ $\text{TSL}^{\mathtt{p}}$:** We apply the transformation $\kappa_{\mathtt{p}}$ to a TSL formula $\varphi$, where

$$\kappa_{\mathtt{p}}(\varphi) := \begin{cases} \mathtt{p}\ \mathtt{s_i} & \text{if } \varphi \equiv \mathtt{s_i} \in \mathtt{I} \cup \mathtt{C} \\ \mathtt{p}\ (\mathtt{p}_\varphi\ \tau_F\ \cdots\ \tau_F) & \text{if } \varphi \equiv \mathtt{p}_\varphi\ \tau_F\ \cdots\ \tau_F \text{ with } \mathtt{p}_\varphi \in \mathtt{P}_\varphi \\ \odot\ \kappa_{\mathtt{p}}(\vartheta) & \text{if } \varphi \equiv \odot\ \vartheta \text{ with } \odot \in \{\neg, \bigcirc\} \\ \kappa_{\mathtt{p}}(\vartheta) \odot \kappa_{\mathtt{p}}(\psi) & \text{if } \varphi \equiv \vartheta \odot \psi \text{ with } \odot \in \{\wedge, \mathcal{U}\} \\ \varphi & \text{otherwise} \end{cases}$$

turning every predicate literal $\mathtt{p}_\varphi \in \mathtt{P}_\varphi$, which appears in the formula $\varphi$, into a function literal $\mathtt{p}_\varphi \in \mathtt{F}_{\kappa_{\mathtt{p}}(\varphi)}$ of the same arity. Thus, with $\mathtt{P}_{\kappa_{\mathtt{p}}(\varphi)} = \{\mathtt{p}\}$ the only predicate literal that is left is $\mathtt{p}$, which is freshly introduced to the formula. Then, every TSL formula $\varphi$ is realizable if and only if $\kappa_{\mathtt{p}}(\varphi)$ is realizable.

"$\Rightarrow$": Assume $\varphi$ is realizable, then there exists a strategy $\sigma\colon (2^{\mathcal{T}_P})^+ \rightarrow \mathcal{C}$ that satisfies $\varphi$ for all $\iota \in \mathcal{I}^\omega$ and all possible choices of $\langle\cdot\rangle\colon \mathtt{F}_\varphi \rightarrow \mathcal{F}$. We claim that $\sigma$ also is a winning strategy with respect to $\kappa_{\mathtt{p}}(\varphi)$. Hence, for the sake of contradiction assume the claim does not hold, *i.e.*, there exists an input $\iota' \in \mathcal{I}^\omega$ and some assignment $\langle\cdot\rangle'\colon \mathtt{F}_{\kappa_{\mathtt{p}}(\varphi)} \rightarrow \mathcal{F}$ such that $\sigma \,\text{⫽}_{\langle\cdot\rangle'}\, \iota', \iota' \not\models_{\langle\cdot\rangle'} \kappa_{\mathtt{p}}(\varphi)$. According to $\langle\cdot\rangle'$, we then can choose a modified assignment $\langle\cdot\rangle$, which assigns every predicate literal $\mathtt{p}_\varphi$, that has been reinterpreted as a function literal after the transformation $\kappa_{\mathtt{p}}$, the composition of $\mathtt{p}$ and $\mathtt{p}_\varphi$ according to $\langle\cdot\rangle'$, *i.e.*, $\langle \mathtt{p}_\varphi \rangle := \langle \mathtt{p} \rangle' \circ \langle \mathtt{p}_\varphi \rangle'$ for all $\mathtt{p}_\varphi \in \mathtt{P}_\varphi$. Next, let $\mathtt{I}' \subseteq \mathtt{I}$ and $\mathtt{C}' \subseteq \mathtt{C}$ be the sets

of input literals and cells, respectively, for which every $\mathtt{s_i} \in \mathtt{I}' \cup \mathtt{C}'$ has been replaced by $\mathtt{p}\ \mathtt{s_i}$ through the transformation $\kappa_\mathtt{p}$. Furthermore, let $\mathtt{I}' \subseteq \mathtt{I}'' \subseteq \mathtt{I}$ and $\mathtt{C}' \subseteq \mathtt{C}'' \subseteq \mathtt{C}$ be the sets of input literals and cells, respectively, that are covered through the transitive closure of direct updates of the form $[\,\mathtt{s_i} \leftarrowtail \mathtt{s}'_\mathtt{i}\,]$ appearing in $\varphi$ and starting from the initial sets $\mathtt{I}'$ and $\mathtt{C}'$. Finally, let $\mathtt{F}' \subseteq \mathtt{F}_\varphi$ be the set of function literals that appear as the last transformation in updates to cells of $\mathtt{C}''$, *i.e.*, those $\mathtt{f} \in \mathtt{F}_\varphi$ that are used in updates of the form $[\,\mathtt{c} \leftarrowtail \mathtt{f}\ \tau_F\ \cdots\ \tau_F\,]$ appearing in $\varphi$ with $\mathtt{c} \in \mathtt{C}''$. Then, we fix $\langle\mathtt{f}\rangle := \langle\mathtt{p}\rangle' \circ \langle\mathtt{f}\rangle'$ for all $\mathtt{f} \in \mathtt{F}'$ and $\langle\mathtt{f}\rangle := \langle\mathtt{f}\rangle'$ for all remaining function literals $\mathtt{f}$. Moreover, we change the input $\iota'$ to $\iota$ with $\iota(\mathtt{i}) := \langle\mathtt{p}\rangle' \circ \iota'(\mathtt{i})$ for all $\mathtt{i} \in \mathcal{I}''$ and $\iota(\mathtt{i}) := \iota'(\mathtt{i})$ otherwise. A simple induction shows, that the computation $\sigma\,\mathbb{1}_{\langle\cdot\rangle}\,\iota$ evaluates exactly the same under the assignment $\langle\cdot\rangle$ as $\sigma\,\mathbb{1}_{\langle\cdot\rangle'}\,\iota'$ under the assignment $\langle\cdot\rangle'$. As a consequence, it follows that $\sigma\,\mathbb{1}_{\langle\cdot\rangle}\,\iota, \iota \not\models_{\langle\cdot\rangle} \varphi$, which proves that $\varphi$ is unrealizable as well. This is, however, a contradiction against our initial assumption, concluding the proof of this direction.

"$\Leftarrow$": Now assume that $\kappa_\mathtt{p}(\varphi)$ is realizable and let $\sigma\colon (2^{\mathcal{T}_\mathcal{P}})^+ \to \mathcal{C}$ be the corresponding realizing strategy. We again claim that $\sigma$ also realizes $\varphi$, which we, nevertheless, endeavor to contradict with the existence of some input $\iota$ and some assignment $\langle\cdot\rangle$ such that $\sigma\,\mathbb{1}_{\langle\cdot\rangle}\,\iota, \iota \not\models_{\langle\cdot\rangle} \varphi$. To this end, we simply choose the extended assignment $\langle\cdot\rangle'$, which assigns the fresh literal $\mathtt{p} \in \mathtt{P}_{\kappa_\mathtt{p}(\varphi)}$, only appearing in $\kappa_\mathtt{p}(\varphi)$, to the identity function *id*, *i.e.*, $\langle\mathtt{p}\rangle' := id$, and keeps all remaining literals unchanged according to $\langle\cdot\rangle$, *i.e.*, $\langle\mathtt{f}\rangle' := \langle\mathtt{f}\rangle$ for all $\mathtt{f} \in \mathtt{F}_\varphi$. Again, a simple induction shows that $\sigma\,\mathbb{1}_{\langle\cdot\rangle}\,\iota, \iota \models_{\langle\cdot\rangle} \varphi$ if and only if $\sigma\,\mathbb{1}_{\langle\cdot\rangle'}\,\iota, \iota \models_{\langle\cdot\rangle'} \kappa_\mathtt{p}(\varphi)$, finally leading to the desired contradiction, since then also $\sigma\,\mathbb{1}_{\langle\cdot\rangle'}\,\iota, \iota \not\models_{\langle\cdot\rangle'} \kappa_\mathtt{p}(\varphi)$.

**$\mathbf{TSL}^\mathtt{p} \to \mathbf{TSL}^\mathtt{p}_{\neg 1}$:** Next, we eliminate unary function literals by replacing them with binary ones. The resulting fragment of the logic is denoted by $\mathrm{TSL}^\mathtt{p}_{\neg 1}$. To this end, the applied transformation $\kappa_{\neg 1}$ recursively traverses over the formula structure via

$$\kappa_{\neg 1}(\varphi) := \begin{cases} \mathit{true} & \text{if } \varphi \equiv \mathit{true} \\ \mathtt{p}\ \kappa'_{\neg 1}(\tau_F) & \text{if } \varphi \equiv \mathtt{p}\ \tau_F \\ [\,\mathtt{s_i} \leftarrowtail \kappa'_{\neg 1}(\tau_F)\,] & \text{if } \varphi \equiv [\,\mathtt{s_i} \leftarrowtail \tau_F\,] \\ \odot\, \kappa_{\neg 1}(\vartheta) & \text{if } \varphi \equiv \odot\, \vartheta \text{ with } \odot \in \{\neg, \bigcirc\} \\ \kappa_{\neg 1}(\vartheta) \odot \kappa_{\neg 1}(\psi) & \text{if } \varphi \equiv \vartheta \odot \psi \text{ with } \odot \in \{\wedge, \mathcal{U}\} \end{cases}$$

and function terms via

$$\kappa'_{\neg 1}(\tau_F) := \begin{cases} \mathtt{f_1}\ \kappa'_{\neg 1}(\tau'_F)\ \kappa'_{\neg 1}(\tau'_F) & \text{if } \tau_F \equiv \mathtt{f_1}\ \tau'_F \\ \mathtt{f}\ \kappa'_{\neg 1}(\tau^1_F)\ \cdots\ \kappa'_{\neg 1}(\tau^m_F) & \text{if } \tau_F \equiv \mathtt{f}\ \tau^1_F\ \cdots\ \tau^m_F \text{ and } m > 1 \\ \tau_F & \text{otherwise} \end{cases}$$

Let $\mathtt{F}^1_\varphi$ be the set of all unary function literals appearing in $\varphi$. Intuitively, all elements of $\mathtt{F}^1_\varphi$ are eliminated by the transformation $\kappa_{\neg 1}$ via replacing them by equally named binary literals that are applied twice to the same argument. We prove that every TSL$^{\mathtt{p}}$ formula $\varphi$ is equi-realizable to $\kappa_{\neg 1}(\varphi)$.

"$\Rightarrow$": Assume that $\varphi$ is realizable. Hence, there exists a realizing strategy $\sigma\colon (2^{\mathcal{T}_P})^+ \to \mathcal{C}$. Consider the strategy $\sigma'$ with $\sigma'(w)(\mathtt{s_i}) := \kappa'_{\neg 1}(\sigma(w)(\mathtt{s_i}))$ for all $w \in (2^{\mathcal{T}_P})^+$ and $\mathtt{s_i} \in \mathtt{0} \cup \mathtt{C}$. We claim that $\sigma'$ is a realizing strategy for $\kappa_{\neg 1}(\varphi)$. However, for the sake of contradiction, assume the opposite, *i.e.*, that there exists an input $\iota'$ and an assignment $\langle\cdot\rangle$, for which $\sigma' \,\mathbb{R}_{\langle\cdot\rangle'}\, \iota', \iota' \not\models_{\langle\cdot\rangle'} \kappa_{\neg 1}(\varphi)$. To this end, we choose $\langle\cdot\rangle\colon \mathtt{F}_\varphi \to \mathcal{F}$ to map every unary function literal $\mathtt{f}_1 \in \mathtt{F}^1_\varphi$ to a function $\langle \mathtt{f}_1 \rangle$ such that $\langle \mathtt{f}_1 \rangle\ v := \langle \mathtt{f}_1 \rangle'\ v\ v$ for all $v \in \mathcal{V}$, while for every other function literal, the assignment $\langle\cdot\rangle$ returns the same as $\langle\cdot\rangle'$. Then a simple induction over the structure of $\mathrm{TSL}^{\mathtt{p}}_{>0}$ formulas $\psi$ proves that $\sigma \,\mathbb{R}_{\langle\cdot\rangle}\, \iota', \iota' \models_{\langle\cdot\rangle} \psi$ if and only if $\sigma' \,\mathbb{R}_{\langle\cdot\rangle'}\, \iota', \iota' \models_{\langle\cdot\rangle'} \kappa_{\neg 1}(\psi)$. Therefore, $\sigma \,\mathbb{R}_{\langle\cdot\rangle'}\, \iota', \iota' \not\models_{\langle\cdot\rangle'} \kappa_{\neg 1}(\varphi)$ also implies that $\sigma \,\mathbb{R}_{\langle\cdot\rangle}\, \iota', \iota' \not\models_{\langle\cdot\rangle} \varphi$ leading to the desired contradiction.

"$\Leftarrow$": Assume that $\kappa_{\neg 1}(\varphi)$ is realizable and let $\sigma'$ be the realizing strategy. W.l.o.g. we can assume that $\sigma'$ only maps to function terms that appear in $\kappa_{\neg 1}(\varphi)$. We construct a strategy $\sigma$ from $\sigma'$ such that for all $w \in (2^{\mathcal{T}_P})^+$ and $\mathtt{s_i} \in \mathtt{0} \cup \mathtt{C}$ we have that $\sigma(w)(\mathtt{s_i}) := (\kappa'_{\neg 1})^{-1}(\sigma'(w)(\mathtt{s_i}))$, where $(\kappa'_{\neg 1})^{-1}$ is the inverse transformation of $\kappa'_{\neg 1}$. This inverse is well-defined due to restriction to function terms appearing in $\kappa_{\neg 1}(\varphi)$, for which the set $\mathtt{F}^1_\varphi$ is known. We claim that $\sigma$ realizes $\varphi$, where for the sake of contradiction we assume the opposite. Then there is an input $\iota$ and an assignment $\langle\cdot\rangle$, under which $\sigma$ does not satisfy $\varphi$. We fix some assignment $\langle\cdot\rangle'$ as follows: for every $\mathtt{f}_1 \in \mathtt{F}^1_\varphi$ and $v, v' \in \mathcal{V}$ let $\langle \mathtt{f}_1 \rangle'\ v\ v' := \langle \mathtt{f}_1 \rangle\ v$ and for all remaining function literals $\mathtt{f}$ let $\langle \mathtt{f} \rangle' := \langle \mathtt{f} \rangle$. A simple induction over the structure of $\varphi$ shows that $\sigma \,\mathbb{R}_{\langle\cdot\rangle}\, \iota, \iota \models_{\langle\cdot\rangle} \varphi$ if and only if $\sigma' \,\mathbb{R}_{\langle\cdot\rangle'}\, \iota, \iota \models_{\langle\cdot\rangle'} \kappa_{\neg 1}(\varphi)$. Hence, according to the contraposition, we derive that $\sigma' \,\mathbb{R}_{\langle\cdot\rangle'}\, \iota, \iota \not\models_{\langle\cdot\rangle'} \kappa_{\neg 1}(\varphi)$, which leads to the desired contradiction.

**$\mathrm{TSL}^{\mathtt{p}}_{\neg 1} \to \mathrm{TSL}^{\mathtt{p}}_{0,2}$:** Next, we eliminate function literals that are applied to more than two arguments and replace them by chains of binary function applications. The resulting TSL fragment is denoted by $\mathrm{TSL}^{\mathtt{p}}_{0,2}$. The utilized transformation $\kappa_2$ recursively traverses over the structure of a given formula $\varphi$:

$$\kappa_2(\varphi) := \begin{cases} \mathit{true} & \text{if } \varphi \equiv \mathit{true} \\ \mathtt{p}\ \kappa'_2(\tau_F) & \text{if } \varphi \equiv \mathtt{p}\ \tau_F \\ [\,\mathtt{s_i} \leftharpoondown \kappa'_2(\tau_F)\,] & \text{if } \varphi \equiv [\,\mathtt{s_i} \leftharpoondown \tau_F\,] \\ \odot\ \kappa_2(\vartheta) & \text{if } \varphi \equiv \odot\ \vartheta \text{ with } \odot \in \{\neg, \bigcirc\} \\ \kappa_2(\vartheta) \odot \kappa_2(\psi) & \text{if } \varphi \equiv \vartheta \odot \psi \text{ with } \odot \in \{\wedge, \mathcal{U}\} \end{cases}$$

and contained function terms with $\kappa_2'$, where $\kappa_2'(\mathtt{s_i}) := \mathtt{s_i}$ for all $\mathtt{s_i} \in \mathtt{I} \cup \mathtt{C}$ and

$$\kappa_2'(\mathtt{f}\ \tau_F^1\ \cdots\ \tau_F^m) := \mathtt{f}\ \kappa_2'(\tau_F^1)\ (\mathtt{t}\ \kappa_2'(\tau_F^2) \cdots (\mathtt{t}\ \kappa_2'(\tau_F^{m-1})\ \kappa_2'(\tau_F^m)) \cdots)$$

The literal $\mathtt{t}$ is a fresh binary function literal that does not appear in the initial formula $\varphi$. Note that binary function literals keep unchanged by $\kappa_2'$. We proof that $\varphi$ and $\kappa_2(\varphi)$ are equi-realizable.

"$\Rightarrow$": Assume $\varphi$ is realizable, *i.e.*, there exists some realizing strategy $\sigma$. We construct the strategy $\sigma'$ via $\sigma'(w)(\mathtt{s_i}) := \kappa_2'(\sigma(w)(\mathtt{s_i}))$ for all $w \in (2^{\mathcal{T}_P})^+$ and $\mathtt{s_i} \in \mathtt{O} \cup \mathtt{C}$ and claim that $\sigma'$ realizes $\kappa_2(\varphi)$. For the sake of contradiction assume the opposite, *i.e.*, that there exists some input $\iota'$ and assignment $\langle\cdot\rangle'$ such that $\sigma' \between_{\langle\cdot\rangle'} \iota', \iota' \not\models_{\langle\cdot\rangle'} \kappa_2(\varphi)$. Then we choose the assignment $\langle\cdot\rangle$ such that for all $\mathtt{f} \in \mathtt{F}_\varphi$ and arguments $v_\mathtt{f}^1, v_\mathtt{f}^2, \ldots, v_\mathtt{f}^m \in \mathcal{V}$ we have that

$$\langle \mathtt{f} \rangle\ v_\mathtt{f}^1\ v_\mathtt{f}^2\ \cdots\ v_\mathtt{f}^m\ :=\ \langle \mathtt{f} \rangle'\ v_\mathtt{f}^1\ (\langle \mathtt{t} \rangle'\ v_\mathtt{f}^2 \cdots (\langle \mathtt{t} \rangle'\ v_\mathtt{f}^{m-1}\ v_\mathtt{f}^m) \cdots)$$

An induction over $\varphi$ shows that $\sigma \between_{\langle\cdot\rangle} \iota', \iota' \models_{\langle\cdot\rangle} \varphi$ iff $\sigma' \between_{\langle\cdot\rangle'} \iota', \iota' \models_{\langle\cdot\rangle'} \kappa_2(\varphi)$. Hence, also $\sigma \between_{\langle\cdot\rangle} \iota', \iota' \not\models_{\langle\cdot\rangle} \varphi$ leading to the desired contradiction.

"$\Leftarrow$": Now assume that $\kappa_2(\varphi)$ is realizable and let $\sigma'$ be the realizing strategy. W.l.o.g. we can assume that $\sigma'$ only maps to function terms that appear in $\kappa_2(\varphi)$. We construct the strategy $\sigma$ via $\sigma(w)(\mathtt{s_i}) := (\kappa_2')^{-1}(\sigma'(w)(\mathtt{s_i}))$ for all $w \in (2^{\mathcal{T}_P})^+$ and $\mathtt{s_i} \in \mathtt{O} \cup \mathtt{C}$, where $(\kappa_2')^{-1}$ is the inverse transformation of $\kappa_2'$, which is well defined on all function terms appearing in $\kappa_2(\varphi)$. We claim that $\sigma$ is a realizing strategy for $\varphi$, but assume the opposite for the sake of contradiction. Hence, let $\iota$ be an input and $\langle\cdot\rangle$ be some assignment, which witness the non-satisfaction. We select the assignment $\langle\cdot\rangle'$ such that $\langle \mathtt{t} \rangle'\ v\ v' := (v, v')$ implements the tuple constructor for all possible values $v, v' \in \mathcal{V}$ and $\langle \mathtt{f} \rangle'$ implements the $m$-times nested un-curried version of $\langle \mathtt{f} \rangle$. A simple induction over the formula structure of $\varphi$ shows that $\sigma \between_{\langle\cdot\rangle} \iota, \iota \models_{\langle\cdot\rangle} \varphi$ if and only if $\sigma' \between_{\langle\cdot\rangle'} \iota, \iota \models_{\langle\cdot\rangle'} \kappa_2(\varphi)$ such that it immediately follows that $\sigma' \between_{\langle\cdot\rangle'} \iota, \iota \not\models_{\langle\cdot\rangle'} \varphi$. This leads to the desired contradiction proving that $\varphi$ is indeed realizable as well.

**$\mathbf{TSL}^{\mathtt{p}}_{0,2} \rightarrow \mathbf{TSL}^{\mathtt{p}}_{\mathtt{c},\mathtt{f}}$:** In the next step, we eliminate binary function applications of different function literals and unify them according to a single function literal $\mathtt{f}$. The resulting fragment is denoted by $\mathrm{TSL}^{\mathtt{p}}_{\mathtt{c},\mathtt{f}}$ and the utilized transformation by $\kappa_\mathtt{f}$. It is defined via

$$\kappa_\mathtt{f}(\varphi) := \begin{cases} \mathit{true} & \text{if } \varphi \equiv \mathit{true} \\ \mathtt{p}\ \kappa_\mathtt{f}'(\tau_F) & \text{if } \varphi \equiv \mathtt{p}\ \tau_F \\ [\,\mathtt{s_i} \leftarrowtail \kappa_\mathtt{f}'(\tau_F)\,] & \text{if } \varphi \equiv [\,\mathtt{s_i} \leftarrowtail \tau_F\,] \\ \odot\ \kappa_\mathtt{f}(\vartheta) & \text{if } \varphi \equiv \odot\ \vartheta \text{ with } \odot \in \{\neg, \bigcirc\} \\ \kappa_\mathtt{f}(\vartheta) \odot \kappa_\mathtt{f}(\psi) & \text{if } \varphi \equiv \vartheta \odot \psi \text{ with } \odot \in \{\wedge, \mathcal{U}\} \end{cases}$$

and the utilized term transformation $\kappa'_{\mathtt{f}}$ is defined via $\kappa'_{\mathtt{f}}(\mathtt{s_i}) := \mathtt{s_i}$ for all $\mathtt{s_i} \in \mathtt{I} \cup \mathtt{C}$ and

$$\kappa'_{\mathtt{f}}(\mathtt{g}\ \tau_F\ \tau'_F) := \mathtt{f}\ \mathtt{c}^{\triangledown}_{\mathtt{g}}\ \Big(\mathtt{f}\ \big(\mathtt{f}\ \mathtt{c}^{\triangledown}_0\ \kappa'_{\mathtt{f}}(\tau_F)\big)\ \big(\mathtt{f}\ \mathtt{c}^{\triangledown}_0\ \kappa'_{\mathtt{f}}(\tau'_F)\big)\Big)$$

where $\mathtt{f}$ is a fresh binary function literal that does not appear in $\varphi$ and $\mathtt{c}^{\triangledown}_{\mathtt{g}}$ and $\mathtt{c}^{\triangledown}_0$ are fresh constant literals for all $\mathtt{g} \in \mathtt{F}_\varphi$ that do not appear in $\varphi$. We prove that $\varphi$ and $\kappa_{\mathtt{f}}(\varphi)$ are equi-realizable.

"$\Rightarrow$": Assume that $\varphi$ is realizable and let $\sigma$ be the strategy realizing $\varphi$. We construct the strategy $\sigma'$ via $\sigma'(w)(\mathtt{s_i}) := \kappa'_{\mathtt{f}}(\sigma(w)(\mathtt{s_i}))$ for all $w \in (2^{\mathcal{T}_P})^+$ and $\mathtt{s_i} \in \mathtt{O} \cup \mathtt{C}$ and claim that $\sigma'$ realizes $\kappa_{\mathtt{f}}(\varphi)$. For the sake of contradiction assume the opposite, *i.e.*, that there exists an input $\iota'$ and an assignment $\langle\cdot\rangle' \colon \{\mathtt{f}, \mathtt{p}\} \to \mathcal{F}$ such that $\sigma' \between_{\langle\cdot\rangle'} \iota', \iota \not\models_{\langle\cdot\rangle'} \kappa_{\mathtt{f}}(\varphi)$. We select the assignment $\langle\cdot\rangle \colon \mathtt{F}_\varphi \to \mathcal{F}$ such that

$$\langle\mathtt{g}\rangle\ v\ v' = \langle\mathtt{f}\rangle'\ \langle\mathtt{c}^{\triangledown}_{\mathtt{g}}\rangle'\ \Big(\langle\mathtt{f}\rangle'\ \big(\langle\mathtt{f}\rangle'\ \langle\mathtt{c}^{\triangledown}_0\rangle'\ v\big)\ \big(\langle\mathtt{f}\rangle'\ \langle\mathtt{c}^{\triangledown}_0\rangle'\ v'\big)\Big)$$

for all $v, v' \in \mathcal{V}$ and all binary function literals $\mathtt{g}$. For all remaining function literals $\mathtt{g}$ we fix $\langle\mathtt{g}\rangle := \langle\mathtt{g}\rangle'$. A simple induction over the structure of the formula $\varphi$ reveals that $\sigma \between_{\langle\rangle} \iota', \iota' \models_{\langle\rangle} \varphi$ if and only if $\sigma' \between_{\langle\rangle'} \iota', \iota' \models_{\langle\rangle'} \kappa_{\mathtt{f}}(\varphi)$. Thus, by the contraposition it follows that $\sigma \between_{\langle\rangle} \iota', \iota' \not\models_{\langle\rangle} \varphi$ leading to the desired contradiction.

"$\Leftarrow$": Now assume that $\kappa_{\mathtt{f}}(\varphi)$ is realizable and let $\sigma'$ be the corresponding realizing strategy. W.l.o.g. we can assume that $\sigma'$ only maps to function terms that appear in $\kappa_{\mathtt{f}}(\varphi)$. We construct a strategy $\sigma$ from $\sigma'$ such that for all $w \in (2^{\mathcal{T}_P})^+$ and $\mathtt{s_i} \in \mathtt{O} \cup \mathtt{C}$ we have that $\sigma(w)(\mathtt{s_i}) := (\kappa'_{\mathtt{f}})^{-1}(\sigma'(w)(\mathtt{s_i}))$, where $(\kappa'_{\mathtt{f}})^{-1}$ is the inverse transformation of $\kappa'_{\mathtt{f}}$. Note that this inverse is well-defined for the fixed formula $\kappa_{\mathtt{f}}(\varphi)$. We claim that $\sigma$ realizes $\varphi$, but assume the opposite for the sake of contradiction. Hence, let $\iota$ and $\langle\cdot\rangle$ be some input stream and literal assignment, respectively, which together witness the non-satisfaction of $\varphi$. We choose the assignment $\langle\cdot\rangle'$ such that $\langle\mathtt{c}^{\triangledown}_0\rangle' := 0$, $\langle\mathtt{c}^{\triangledown}_{\mathtt{g}}\rangle' := \langle\mathtt{g}\rangle$ for all $\mathtt{g} \in \mathtt{F}_\varphi$, $\langle\mathtt{init_c}\rangle' := \langle\mathtt{init_c}\rangle$ for all cells $\mathtt{c} \in \mathtt{C}$, and

$$\langle\mathtt{f}\rangle'\ v\ v' := \begin{cases} (0, v') & \text{if } v \equiv 0 \\ (v, v') & \text{if } v \equiv (x, y) \\ v\ (pr_1(pr_0(v')))\ (pr_1(pr_1(v'))) & \text{otherwise} \end{cases}$$

for all $v, v' \in \mathcal{V}$. For the evaluation of $\langle\mathtt{f}\rangle'$ consider that the value 0, tuples $(x, y)$, and functions $g$ are always pairwise unequal. Furthermore, note that the application of $\langle\mathtt{f}\rangle'$ is always well-defined for the evaluation of function terms that appear in a formula $\kappa_{\mathtt{f}}(\varphi)$. Then an induction over the formula

structure of $\varphi$ shows that $\sigma \,\mathbb{R}_{\langle\cdot\rangle}\, \iota, \iota \;\models_{\langle\cdot\rangle}\; \varphi$ if and only if $\sigma' \,\mathbb{R}_{\langle\cdot\rangle'}\, \iota, \iota \;\models_{\langle\cdot\rangle'}\; \kappa_{\mathtt{f}}(\varphi)$. Hence, according to the contraposition it follows that $\sigma' \,\mathbb{R}_{\langle\cdot\rangle'}\, \iota, \iota \;\not\models_{\langle\cdot\rangle'}\; \kappa_{\mathtt{f}}(\varphi)$. This finally leads to the desired contradiction.

**$\mathbf{TSL}^{\mathtt{p}}_{\mathtt{c},\mathtt{f}} \rightarrow \mathbf{TSL}^{\mathtt{p}}_{\mathtt{f}}$:** With the final step, we remove constant literals from $\mathrm{TSL}^{\mathtt{p}}_{\mathtt{c},\mathtt{f}}$ formulas $\varphi$ eventually leading to the fragment $\mathrm{TSL}^{\mathtt{p}}_{\mathtt{f}}$. Therefore, let $\mathtt{F}^0_\varphi := \mathtt{F}_\varphi \setminus \{\mathtt{p}, \mathtt{f}\}$ be the set of all remaining constant function literals appearing in $\varphi$. We define the transformation $\kappa_{\neg 0}$ via

$$\kappa_{\neg 0}(\varphi) \;:=\; \square\Big( \bigwedge_{\mathtt{c} \in \mathtt{F}^0_\varphi} [\, \mathtt{c} \leftarrowtail \mathtt{c} \,] \Big) \;\wedge\; \varphi$$

where constant function literals $\mathtt{c} \in \mathtt{F}^0_\varphi$ are reinterpreted as cells $\mathtt{c} \in \mathtt{C}_{\kappa_{\neg 0}(\varphi)}$. We prove that for every $\mathrm{TSL}^{\mathtt{p}}_{\mathtt{c},\mathtt{f}}$ formula $\varphi$: $\varphi$ is equi-realizable to $\kappa_{\neg 0}(\varphi)$.

"$\Rightarrow$": To this end, we first assume that $\varphi$ is realizable, *i.e.*, there exists a realizing strategy $\sigma\colon (2^{\mathcal{T}_P})^+ \to \mathcal{C}$. Now, consider the modified strategy $\sigma'$, which returns the same computations as $\sigma$, but is extended with identity updates for all cells $\mathtt{f}_0 \in F^0_\varphi \subseteq \mathtt{C}_{\kappa_{\neg 0}(\varphi)}$, *i.e.*, for every input prefix $w \in (2^{\mathcal{T}_P})^+$ we have that $\sigma'(w) = \sigma(w) \cup \bigcup_{\mathtt{f}_0 \in F_\varphi} \{\mathtt{f}_0 \mapsto \mathtt{f}_0\}$. We claim that $\sigma'$ satisfies $\kappa_{\neg 0}(\varphi)$ for every chosen input $\iota'$ and assignment $\langle\cdot\rangle'$, but for the sake of contradiction assume there would be $\iota' \in \mathcal{I}^\omega$ and $\langle\cdot\rangle'\colon \mathtt{F}_{\kappa_{\neg 0}} \to \mathcal{F}$ such that $\sigma' \,\mathbb{R}_{\langle\cdot\rangle'}\, \iota', \iota' \;\not\models_{\langle\cdot\rangle'}\; \kappa_{\neg 0}(\varphi)$. Therefore, according to the aforementioned choice of $\sigma'$ it immediately follows that $\sigma' \,\mathbb{R}_{\langle\cdot\rangle'}\, \iota', \iota' \;\not\models_{\langle\cdot\rangle'}\; \varphi$. With this at hand, we choose the assignment $\langle\cdot\rangle\colon \mathtt{F}_\varphi \to \mathcal{F}$ with $\langle \mathtt{f}_0 \rangle := \langle \mathtt{init}_{\mathtt{f}_0} \rangle'$ for all $\mathtt{f}_0 \in \mathtt{F}^0_\varphi$ and $\langle \mathtt{f} \rangle := \langle \mathtt{f} \rangle'$ otherwise. With respect to Definition 12 of the evaluation function $\eta_{\langle\cdot\rangle}$ it is straightforward to see that $\sigma \,\mathbb{R}_{\langle\cdot\rangle}\, \iota'$ and $\sigma' \,\mathbb{R}_{\langle\cdot\rangle'}\, \iota'$ evaluate the same. As a consequence, $\sigma' \,\mathbb{R}_{\langle\cdot\rangle'}\, \iota', \iota' \;\not\models_{\langle\cdot\rangle'}\; \varphi$ immediately implies that also $\sigma \,\mathbb{R}_{\langle\cdot\rangle}\, \iota', \iota' \;\not\models_{\langle\cdot\rangle}\; \varphi$, which finally leads to the targeted contradiction.

"$\Leftarrow$": Next, we assume that $\kappa_{\neg 0}(\varphi)$ is realizable, as witnessed by some realizing strategy $\sigma'$. This time, let $\sigma$ be the strategy derived from $\sigma'$, where the domain of every returned computation step has been reduced to $\mathtt{0} \cup \mathtt{C}_\varphi$. We claim that $\sigma$ realizes $\varphi$, against which we strike for the sake of contradiction with the hypothesis of an existing input $\iota \in \mathcal{I}^\omega$ and an assignment $\langle\cdot\rangle\colon \mathtt{F}_\varphi \to \mathcal{F}$ proving that $\sigma \,\mathbb{R}_{\langle\cdot\rangle}\, \iota, \iota \;\not\models_{\langle\cdot\rangle}\; \varphi$. We choose $\langle\cdot\rangle'\colon \mathtt{F}_{\kappa_{\neg 0}(\varphi)} \to \mathcal{F}$ such that $\langle \mathtt{init}_{\mathtt{f}_0} \rangle' := \langle \mathtt{f}_0 \rangle$ for all $\mathtt{f}_0 \in \mathtt{F}^0_\varphi$ and $\langle \mathtt{f} \rangle' = \langle \mathtt{f} \rangle$ otherwise. Again, it is straightforward to see that $\sigma \,\mathbb{R}_{\langle\cdot\rangle}\, \iota$ and $\sigma' \,\mathbb{R}_{\langle\cdot\rangle'}\, \iota$ evaluate the same. Since $\sigma \,\mathbb{R}_{\langle\cdot\rangle}\, \iota, \iota \;\not\models_{\langle\cdot\rangle}\; \varphi$ clearly implies that $\sigma' \,\mathbb{R}_{\langle\cdot\rangle}\, \iota, \iota \;\not\models_{\langle\cdot\rangle}\; \varphi$, also $\sigma' \,\mathbb{R}_{\langle\cdot\rangle}\, \iota, \iota \;\not\models_{\langle\cdot\rangle}\; \kappa_{\neg 0}(\varphi)$. This contracts the initial assumption and, concludes this part of the proof.

Finally, with all five transformations at hand, the claimed formula $\varphi'$ can be constructed via $\varphi' := \kappa_{\neg 0}(\kappa_{\mathtt{f}}(\kappa_2(\kappa_{\neg 1}(\kappa_{\mathtt{p}}(\varphi)))))$. Furthermore note, that every transformations is linear in the size of it's corresponding input formula, and, thus, $\varphi'$ can be constructed in linear time in $|\varphi|$. □

The results of Theorem 4 show that only a single unary predicate `p` and a single binary function literal `f` are needed to achieve the full expressiveness of TSL. More reductions of the term syntax, however, come with semantic restrictions that imply strong practical limitations on the application side. Nevertheless, the transformations that we utilized for the proof of Theorem 4 show that there is a linear increase in the number of cells required to express the replaced function and predicate literals. Hence, with respect to Theorem 3 and $\text{TSL}_{\mathtt{f}}^{\mathtt{p}}$, we observe that the number of required cells indeed depends on the number of sequences $w_0 w_1 \dots w_n, v_0 v_1 \dots v_n$ that are specific to the PCP instance. Thus, whether an equivalent proof of Theorem 3 on the basis of $\text{TSL}_{\mathtt{f}}^{\mathtt{p}}$ is possible, which only requires a constant number of cells, is an open question.

# 5 Temporal Stream Games

Our previous considerations show that the realizability problem being undecidable cannot be avoided with term restrictions in general, even when limiting ourselves to only a single unary predicate `p` and a single binary function term `f`. As a consequence, we also must consider other possible directions in order to tackle the undecidability property of the realizability problem.

One particular of these directions is the conversion of formulas to infinite two-player games, which switches the perspective from the logic world to the equivalent game world alternative. The translation to infinite games comes at the advantage that Boolean connectives have been resolved and that the temporal operators are unfolded, such that their semantics can be reflected by the winning condition of the two-player games. To this end, at least the parity winning condition is required to cover all the behavior of the temporal operators that are utilized by TSL. Hence, in order to identify the cause for the undecidability of TSL, the reduction to weaker winning conditions, such as Safety, Reachability, or Büchi, may bring us forward in terms of avoiding the repelling ingredients of the realizability problem.

Unfortunately, it turns out that translating TSL to infinite two-player games, which provision exactly the same semantics as Problem 1, is not as gentle as their equivalent counterparts for standard temporal logics like LTL. To this end, we discuss the corresponding peculiarities at first. Nevertheless, in order to get around these, we will impose some restrictions and assumptions that may be avoided at the price of further investigations. These are, however, out of scope of this thesis, since they require a much more involved analysis in general. As a consequence, we will leave some open questions behind.

## 5.1 Determinacy

The first difference, we need to consider, relates to the determinacy property of infinite two-player games. An infinite two-player game is said to be determined, if either always the system player or the environment player has a winning strategy. Or in other words: it cannot be the case that none of the players is able to win the game. Almost every class of infinite two-player games that are considered in practice is known to be determined, like for example the class of games with $\omega$-regular winning conditions. Determinacy is a desirable property, since it allows to prove the non-existence of a winning strategy for one player through the existence of a winning strategy for the other player, and, thus, gives a constructive argument for both: realizable and unrealizable specifications.

On the logic level, both players are represented by the quantifies $\exists$ and $\forall$ such that according to this representation, the determinacy property allows us to swap their order without affecting the validity of the corresponding realizability query. For example, the realizability problem of LTL relies on an $\exists$-$\forall$-quantifier alternation

$$\exists \sigma.\ \forall \pi.\ \rho_{\sigma,\pi} \vDash \varphi$$

that queries the existence of a winning strategy $\sigma$ for the system player satisfying the specification against all possible input player strategies $\pi$. The realizability problem of LTL can be reduced to infinite two-player games with parity winning condition, since LTL is an $\omega$-regular language. Thus, we can apply a quantifier swap in order to prove LTL unrealizability, because infinite two-player games with parity winning conditions are determined [152]. More concrete, if an LTL formula $\varphi$ is unrealizable, *i.e.*,

$$\forall \sigma.\ \exists \pi.\ \rho_{\sigma,\pi} \vDash \neg\varphi$$

then due to the determinacy of games resulting from $\varphi$, the statement is equivalent to

$$\exists \pi.\ \forall \sigma.\ \rho_{\sigma,\pi} \nvDash \varphi$$

In this sense, determinacy allows us to move the universal quantifier before the existential one. Note that the other direction is always possible, independently of the game being determined or not.

Our goal is to translate TSL formulas to infinite two-player games such that solving the infinite game at the same time solves the realizability problem. However, this immediately raises the question: Are the resulting games even determined? To this end, reconsider the quantifier alternation of the realizability query of Problem 1:

$$\exists \sigma.\ \forall \iota.\ \forall \langle\cdot\rangle.\ \sigma \between_{\langle\rangle} \iota, \iota \ \vDash_{\langle\rangle} \varphi$$

The query asks for a system player strategy $\sigma$ that satisfies the formula against any input choice $\iota$ and any assignment $\langle\cdot\rangle$. Hence, according to the conceptual view of the quantifier alternation being a two-player game, the system player is in charge of choosing the updates, while the environment player selects the input data and the function implementations. However, formally, this idea is not immediately reflected by the above formulation, since the choices of the environment are not determined according to a strategy responding to the system, but instead are expanded into the $\forall$-quantifiers over all inputs and assignments. Thus, is a similar consideration even applicable to TSL?

We can answer with "yes", since conceptually both views are equivalent. The reason is that every decision, which is made along the different branches of the corresponding strategy tree, only depends on the past, but not on the decisions made on other branches of the tree. As a consequence, every choice that is covered by the "$\forall\iota.\ \forall\langle\cdot\rangle.$" prefix can be compressed into a strategy

$$\pi\colon \mathcal{C}^{\omega} \to \mathcal{I} \cup \{\langle\cdot\rangle\colon \mathtt{F} \rightharpoonup \mathcal{F}\}$$

that selects input values and assignments according to a step-wise semantics. Therefore, even the assignment $\langle\cdot\rangle$ can be determined in an iterative fashion through a refinement of partial functions $\langle\cdot\rangle\colon \mathtt{F} \rightharpoonup \mathcal{F}$, which the environment player only must fix, whenever they are needed for the evaluation of a predicate at the current point in time. We only have to take care, that once an implementation of a function literal has been fixed, it also must stay at this implementation for the remaining duration of the game in order to ensure purity of the function implementations. Furthermore, note that the on-demand refinement does not weaken the system player at this point, since the system must provide a strategy that wins independently of the environment anyway.

According to these considerations, the game based formulation of TSL realizability again reduces to the familiar query of the form of $\exists\sigma.\ \forall\pi.\ \rho_{\sigma,\pi} \vDash \varphi$. Likewise, determinacy of TSL reduces to proving that $\exists\pi.\ \forall\sigma.\ \rho_{\sigma,\pi} \nvDash \varphi$ implies that the formula $\varphi$ is unrealizable. Consider, however, that the statement is not equivalent to $\exists\iota.\ \exists\langle\cdot\rangle.\ \forall\sigma.\ \sigma\,\mathbb{1}_{\langle\rangle}\,\iota, \iota \not\vDash_{\langle\rangle} \varphi$, which again is too strong on the other hand, since a reaction of the environment against all potential system choices is not possible. The determinacy definition of TSL, thus, differs against the equivalent definition of LTL, because the TSL realizability property additionally includes the universal quantification over function and predicate implementations. This in contrast to the function implementations conceptually being implemented after synthesis and, thus, not being resolved according to a step-wise semantics. Nevertheless, with the aim of having a well-defined quantifier swap along with a corresponding determinacy definition, we give the control over the implementations to the

environment, which fixes them with every of his moves accordingly. We leave it up to the reader, whether our considerations indeed are valid in the sense of the original semantics of TSL.

The other question, which we leave open as well, is whether the resulting games indeed are determined according to our previous considerations. An answer would require a more involved analysis of the underlying game semantics, which, however, is out of scope of this thesis. Thus, for the sake of simplicity, we just consider $\exists\pi.\ \forall\sigma.\ \rho_{\sigma,\pi} \not\models \varphi$ to be the required condition for proving unrealizability, even if we cannot guarantee that such a strategy $\pi$ always exists. It is still open, whether we can always find a winning strategy for both players in general.

## 5.2 Branching Restrictions

Another peculiarity, that we need to discuss, is the freedom of the synthesizer in choosing update terms according to the formulation of the realizability problem and the semantics of TSL. A small example that demonstrates the corresponding problem is given by the formula $\square\neg[\,\mathtt{x} \leftarrowtail \mathtt{x}\,]$. The formula basically states that any satisfying solution never does nothing on the output stream $\mathtt{x}$ at any point in time. Is $\square\neg[\,\mathtt{x} \leftarrowtail \mathtt{x}\,]$ realizable? According to our formulation of Problem 1, the answer depends. Remember that the realizability query universally quantifies over all possible implementations of function literals $\mathtt{F}$. Hence, if there is no function literal, nor another cell or input, *i.e.*, if $\mathtt{F} = \mathtt{C} = \mathtt{I} = \emptyset$, then the formula indeed is not realizable, because the only possible function term that remains for the update is $\mathtt{x}$. Otherwise, if there is at least one function $\mathtt{f} \in \mathtt{F}$ or some $\mathtt{s_i} \in \mathtt{I} \cup \mathtt{C}$, then the formula is realizable. For example, a possible solution would be to always update $\mathtt{x}$ with $\mathtt{f\ x}\ \cdots\ \mathtt{x}$, such that the number of arguments of $\mathtt{f}$ match, or we could use the existing cell or input.

Just for clarification: one could argue at this point that this is not correct in general, since the universal quantification also covers the implementation of a unary function literal $\mathtt{f}$ with the identity function. This implementation then leads to a solution that is semantically equivalent to the one always updating $\mathtt{x}$ with $\mathtt{x}$. This argument, however, is refuted by the fact that the semantics of TSL build on the syntactic equivalence of updates instead of considering their semantic evaluation. Such a design is desirable for practical reasons, since there are updates that usually correspond to some code snippets, which need to be selected at the right points in time by the synthesizer, accordingly, to be composed to a final program in the end. Therefore, the synthesis engine does not know how the execution of the snippet behaves, but only takes care

that they are executed at the right points in time. Accordingly, the program that always updates `x` with `x` and the program that always updates `x` with `f x` are different programs, even if `f` is the identity function.

That being clarified, let us assume that there is some unary function $\mathtt{f} \in \mathtt{F}$ such that the formula $\square\neg[\,\mathtt{x} \leftarrowtail \mathtt{x}\,]$ is realizable, *e.g.*, if we always update `x` with `f x`. However, then it is also realizable, if we update `x` with `f (f x)`, or `f (f (f x))`, or basically any chain of function applications of `f`. Thus, even if the specification only considers a finite number of update choices, the synthesizer can choose from an infinite number of possible alternatives leading to infinite branching of corresponding strategy trees.

This is a displeasing observation with respect to our targeted translation to infinite games, since in the game arena resulting from the TSL formula, every outgoing edge of the system player must correspond to a possible update. Similarly, every possible predicate term leads to an outgoing edge for the environment player, even if not used by the original specification. Hence, we potentially would need an arena with an edge relation of infinite size, which we, however, like to avoid for practical reasons. To this end, we separate the synthesis question into two variants, which we call *creational TSL* and *non-creational TSL*. The first variant allows the synthesizer to be creative, *i.e.*, to introduce new updates and to check new predicates, even if they do not appear in the specification. The variant, thus, covers the full semantics of the realizability problem, as stated in Problem 1. Non-creational TSL, on the other hand, restricts the usage of updates and predicates to those that indeed appear in the formula, with one exception: for cells $\mathtt{c} \in \mathtt{C}$, the self-update $[\,\mathtt{c} \leftarrowtail \mathtt{c}\,]$ can still be chosen, even if it does not appear in the formula. This exception is added for practical reasons, since a cell being unchanged by default is a well-agreed assumption also used in sequential programming languages, where, according to the semantics of standard sequential programs, variables do not change if they are not assigned a new value explicitly.

Clearly, non-creational TSL considers an easier realizability question than it's creational counterpart and, thus, should be the preferred choice in practice. There are even more advantages to prefer non-creational TSL. One is that non-creational TSL requires the designer to explicitly state the set of allowed updates and predicate checks as part of the specification. Therefore, the choice of updates, from which the synthesis engine chooses, is under the designers control. Furthermore, non-creational TSL limits the number of updates to a finite set of elements, and thus solves the aforementioned problem of infinite sized arenas. Hence, for the construction of games from TSL formulas, we only consider non-creational TSL. Nevertheless, all lower bounds, we derive for non-creational TSL in a sequel, lift to creational TSL as well.

## 5.3 Purity

We finally are ready to convert TSL formulas $\varphi$ into two player games, which we also denote as *Temporal Stream Games* in the sequel. To this end, we utilize the same $\omega$-automata over infinite words and their corresponding language preserving transformations, as used for classical temporal logics that are based on atomic propositions. The only difference is that we need to take care of the utilized terms, which appear as part of the updates and predicates. To this end, we restrict ourselves to non-creational TSL such that the set of possible updates and predicates is limited to be finite. Therefore, it suffices to limit the set of predicate terms to those that appear in $\varphi$, as they are the only ones whose control behavior has an influence on the satisfaction of $\varphi$. Note that for the resolution of temporal and Boolean operators, we do not have to introduce new predicate checks or updates that are not present in the original formula in general, even in the case of creational TSL. Everything that can be derived from the formula $\varphi$, still can be derived from the later game structure as well. Nevertheless, the limitation to non-creational TSL is of importance for a clean game semantics, because even though we still can derive non-explicit updates from the game structure, the corresponding implicit updates must be reflected by the game semantics. With the restriction to non-creational TSL, every appearing predicate literal can be considered as an input proposition and every appearing update as output proposition instead, which is what we require for the intermediate $\omega$-automata translations.

In this way, every TSL formula $\varphi$ can be translated into a corresponding parity game. In the automata world, there are multiple possibilities for this purpose. A classical path is to first translate the formula $\varphi$ into a very weak alternating Büchi word automaton [123] and then to remove conjunctive transitions according to Miyano&Hayashi [106]. Afterwards, the resulting non-deterministic Büchi word automaton is turned into a deterministic parity word automaton with Safra's construction [124], which then is expanded to a deterministic parity tree automaton along the input propositions. Finally, the automaton is turned into an infinite game according to Rabin's theorem [120].

In the resulting two-player game, the system player determines the updates, while the environment player is in charge of providing inputs and fixing function and predicate evaluations. Therefore, he must respect their pure evaluation semantics, *i.e.*, he cannot choose predicates to evaluate arbitrarily at every point in time, which is in contrast to predicates terms being just re-interpreted as input propositions. However, in order to stay as close as possible to the standard setting, we express purity as an extension to the classical winning conditions, which lets the input player loose as soon as it is

violated. Note that the system player never can violate purity, since she only determines the control flow of the system. This is why our extension only affects the environment player. He determines the evaluation of predicates, which depends on true environmental inputs, passed either directly or via arguments, that are not known till the final execution of the system, but also on the selected function and predicate implementations. These implementations, however, are only fixed once after the synthesis, since they are required for the creation of the final program in the end.

We model this concept through a winning condition that enforces the environment to select predicate evaluation results, such that they can always be witnessed with corresponding function and predicate evaluations. In other words, the environment cannot evaluate a predicate differently at two different points in time, if the passed arguments are provably the same independently of the chosen function and predicate evaluations. Formally, it thus remains to choose the input alphabet $\Sigma_I = 2^{\mathcal{T}_P}$ such that it consists of all combinations of evaluations of predicates $\tau_P \in \mathcal{T}_P$ that appear in the original formula $\varphi$. Similarly, the output alphabet $\Sigma_O = \mathcal{C}$ consists of all assignments of function terms to outputs and cells, as determined by the updates of the original formula $\varphi$. Temporal stream games then are played in arenas over $\Sigma_I$ and $\Sigma_O$, extended by a set of impure plays to be avoided by the environment.

**Definition 16.** Let $\Xi = V_I \times \Sigma_I \times V_O \times \Sigma_O$. The set of plays that violate purity is formally defined as:

$$\begin{aligned}\textsc{Impure} \;:=\; \{\, \rho \in \Xi^\omega \mid \forall \iota \in \mathcal{I}^\omega.\, \forall \langle\cdot\rangle \colon \mathtt{F} \to \mathcal{F}.\, \exists t, t' \in \mathbb{N}.\, \exists \tau_P, \tau'_P \in \mathcal{T}_P.\\ \tau_P \in pr_1(\rho_t) \;\wedge\; \tau'_P \notin pr_1(\rho_{t'}) \;\wedge\\ \eta_{\langle\cdot\rangle}(pr_3(\rho), \iota, t, \tau_P) \;\leftrightarrow\; \eta_{\langle\cdot\rangle}(pr_3(\rho), \iota, t', \tau'_P)\,\}\end{aligned}$$

The set contains every play, for which, independently of the input $\iota$ and the function assignment $\langle\cdot\rangle$, there are two points in time $t$ and $t'$ such that there are predicates $\tau_P$ and $\tau'_P$ that are evaluated differently at $t$ and $t'$, according to the input player, but evaluate the same according to the semantic evaluation $\eta_{\langle\cdot\rangle}$. Thus, the different evaluation of the predicate cannot be witnessed through a corresponding input and a respective assignment of literals to function and predicate implementations.

In temporal stream games, the system player then either wins, if she satisfies the original winning condition, or if the environment player violates the purity condition. As result, we obtain the following extended classes of winning conditions for temporal stream games:

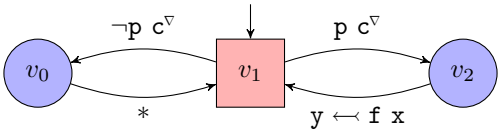



Figure 17: The safety stream game for the TSL formula $\square(\mathtt{p}\ \mathtt{c}^{\triangledown} \rightarrow [\mathtt{y} \leftharpoondown \mathtt{f}\ \mathtt{x}])$. Plays of this game only leave the safe region, if Player $O$ assigns something different to `y` than `f x`. For the sake of readability all edges that lead to an unsafe vertex, as well as the unsafe vertices themselves, are never depicted.

- $\textsc{StreamSafety}(S) := \textsc{Safety}(S) \cup \textsc{Impure}$
- $\textsc{StreamReach}(R) := \textsc{Reach}(R) \cup \textsc{Impure}$
- $\textsc{StreamParity}(\Omega) := \textsc{Parity}(\Omega) \cup \textsc{Impure}$

An example is given by the safety stream game that results from the TSL formula $\square(\mathtt{p}\ \mathtt{c}^{\triangledown} \rightarrow [\mathtt{y} \leftharpoondown \mathtt{f}\ \mathtt{x}])$, which is depicted in Figure 17. It is easy to observe that the system player controls the updates, while the environment player controls the predicate evaluations. Furthermore, both players strictly alternate between their positions and pick updates and predicates according to the outgoing edges. However, not all of these choices appear on the edge labels of the outgoing edges due to our representation. As we consider safety games, we utilize a compact representation, where we assume that every vertex that is part of the arena is safe. Hence, every move of a player, that does not appear as an outgoing edge of a vertex automatically leads to an unsafe vertex, because if such a move is taken, then Player $I$ automatically wins the game. Hence, in the arena of Figure 17 an unsafe is reached, if Player $O$ plays something different than $[\mathtt{y} \leftharpoondown \mathtt{f}\ \mathtt{x}]$ at vertex $v_2$.

For a compact representation of edges that leave the vertices owned by Player $I$, we use Boolean formulas over the respective predicate terms to describe corresponding subsets of $\Sigma_I$. Similarly, if Player $O$ can play arbitrary updates form a position, then we consolidate the corresponding outgoing edges with a $*$-symbol. Otherwise, the edges are labeled with a table representing the corresponding assignment function of $\Sigma_O$. We only consider games where either $\Sigma_I$ and $\Sigma_O$ are clear from the context or are given explicitly otherwise.

The game of Figure 17 is safety stream game. Thus, Player $I$ not only must reach an unsafe vertex in order to win, but also always must satisfy the purity condition. Accordingly, since `p` is evaluated on a constant $\mathtt{c}^{\triangledown}$ the corresponding Boolean result cannot change over time. Thus, if Player $I$

chooses for example to move from $v_1$ to $v_2$ in his first move, then he must also move to $v_2$ in every later move. Otherwise, purity is violated and he immediately looses. Consequently, the environment player also must take the semantic evaluation of the edge labels into account in order to avoid a possible purity violation.

## 5.4 Memory Requirements

The example of Figure 17 indicates that the additional purity condition potentially requires that winning strategies for temporal stream games need to access the past. Up to the parity winning condition, this is in contrast to classical infinite games, which only need positional strategies, *i.e.*, both players can always determine their next move only from the vertex the game token is currently placed at. For temporal stream games, however, the players may need to choose different successors depending on the past of the play, which is why they need memory to satisfying the classical winning conditions, while preserving purity at the same time. In this section, we have a closer look on the corresponding implications. Accordingly, we show that indeed, already for reachability and safety games, both player need memoryful stratgies in order to win. With respect to these observations, we then establish lower and upper bounds on the memory requirements for both players.

We start with reachability stream games, where we first consider the memory requirements of the environment player.

**Theorem 5.** *There exists a reachability stream game that is won by the environment player, but every winning strategy requires infinite memory.*

*Proof.* Consider the reachability game depicted in Figure 18. In this game, the goal of Player $O$ is to reach the vertex $v_{\boldsymbol{\chi}}$, which, thus, must be avoided by Player $I$ in contrary. To this end, Player $O$ must manipulate the cells $\{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$, which can be set to the constant $\mathtt{c}^{\triangledown}$ or be transformed via the unary function $\mathtt{f}$.

First, the system player initializes the cell $\mathtt{x}$ to $\mathtt{c}^{\triangledown}$ at vertex $v_I'$. Afterwards, at vertex $v_c$, she can update $\mathtt{x}$ with $\mathtt{f}\ \mathtt{x}$ over an open number of rounds, producing an unbounded chain of values that are stored in $\mathtt{x}$. At the same time, she either copies the content of $\mathtt{x}$ to $\mathtt{y}$ or to $\mathtt{z}$, such that according to these choices, Player $I$ must either claim that the predicate $\mathtt{p}$ evaluates to *true* or to *false*, since he would loose the game immediately otherwise. Note that Player $I$ always can choose a predicate $\mathtt{p}$ that satisfies the requirements imposed by Player $O$. Furthermore, it also is under control of Player $O$, whether

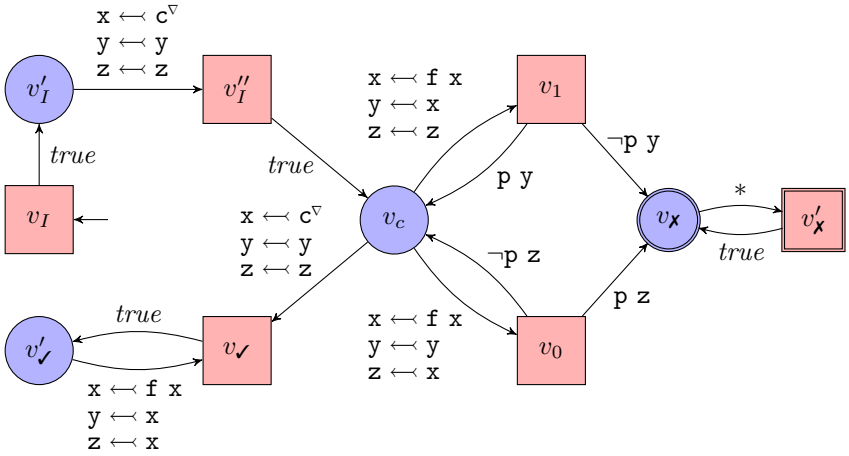



Figure 18: A reachability stream game that is won by Player $I$, where every winning strategy requires at least infinite memory.

she wants to repeat this cycle forever, since she can always stop it by moving to $v_✓$ instead. In this case, the value of $\mathtt{x}$ is reset to $\mathtt{c}^▽$ and the production of the chain of values stored in $\mathtt{x}$ is restarted from scratch. Once this second cycle is reached, the game continues there forever. Nevertheless, the environment still must choose predicate evaluations at $v_✓$ such that they are in line with respect to purity.

The environment player has a winning strategy in this game, which avoids $v_✗$ and satisfies purity. The strategy only needs to remember the choices of the system at $v_c$ as long as Player $O$ decides to cycle through $v_c$, $v_0$ and $v_1$, where Player $I$ can always move back to $v_c$ without violating purity, because there always are implementations for $\mathtt{c}^▽$, $\mathtt{f}$, $\mathtt{p}$ that are in line with these choices. A possible implementation could for example implement $\langle \mathtt{c}^▽ \rangle := 0$, $\langle \mathtt{f} \rangle\ x := x+1$, and choose $\langle \mathtt{p} \rangle \colon \mathbb{N} \to \mathcal{B}$ such that it repeats the choices at $v_c$. If the system then decides to move to $v_✓$, eventually, Player $I$ only must repeat these choices by the respective evaluations of $\mathtt{p}$ at $v_✓$. This is always possible, since all required information can be accessed via the history of the play.

At the same time, Player $I$ only can win if he is able to remember an unbounded number of choices. Due to the system player being able to decide on the number of cycles through $v_c$, she can easily exceed any memory limitations of Player $I$ by cycling more often than any given bound. However, if not remembered correctly, the environment cannot repeat all the choices at $v_✓$ and, thus, has no guaranteed winning strategy. □

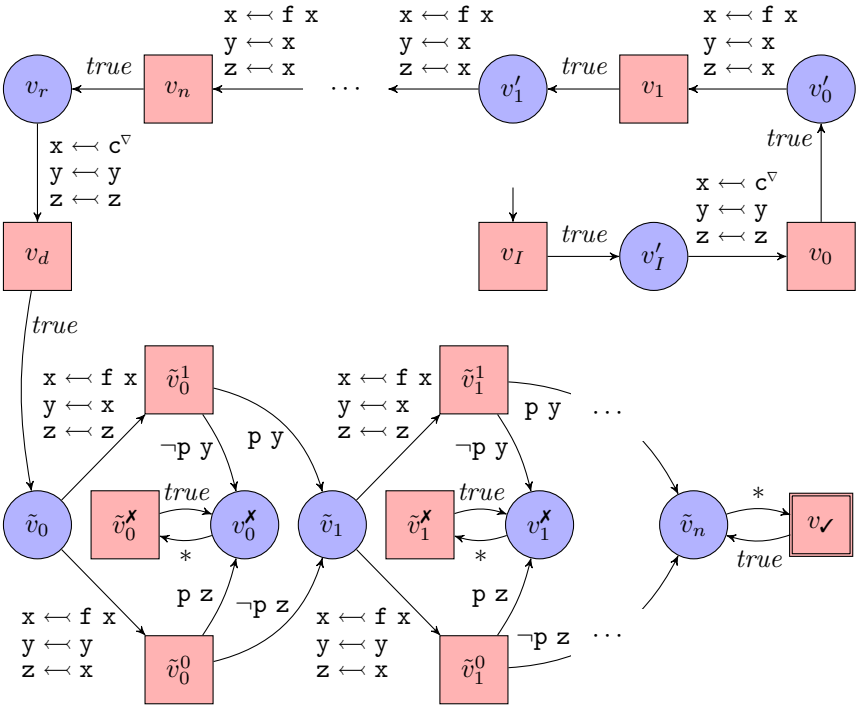



Figure 19: A reachability stream game that is won by Player $O$, where all winning strategies require at least exponential memory in the size of the arena.

Note that even if Player $I$ is limited to finite memory, the system still does not have a winning strategy. Although, Player $I$ cannot access the whole history of a play for choosing correct evaluations of $\mathtt{p}$ at $v_{\checkmark}$ for sure, accordingly, she still can correctly guess them instead. Thus, the environment can win with finite memory. Player $I$ just does not have a winning strategy.

Next, we consider the memory requirements of the system player.

**Theorem 6.** *There exists a family of reachability stream games that are won by the system player, but every winning strategy requires memory that grows at least exponentially with the size of the game arena.*

*Proof.* Consider the family of temporal reachability games of Figure 19, which are parameterized in $n \in \mathbb{N}$. There, the system player must reach $v_{\checkmark}$, for which she must correctly update $\mathtt{x}$, $\mathtt{y}$, and $\mathtt{z}$ using $\mathtt{c}^{\triangledown}$ and the unary function $\mathtt{f}$.

At the beginning of every play, the cell $\mathtt{x}$ is first initialized to $\mathtt{c}^{\triangledown}$ while $\mathtt{y}$ and $\mathtt{z}$ stay unchanged. Afterwards, $\mathtt{x}$ is updated with $\mathtt{f}\ \mathtt{x}$ for $n$ iterations. Furthermore, at every iteration the cells $\mathtt{y}$ and $\mathtt{z}$ receive a copy of $\mathtt{x}$. Meanwhile, the environment leaks information about the predicate $\mathtt{p}$, since it determines the truth values of the evaluations of $\mathtt{p}\ \mathtt{y}$ and $\mathtt{p}\ \mathtt{z}$, which covers the repeated applications of $\mathtt{f}$. Eventually, the vertex $v_n$ is reached, where $\mathtt{x}$ again is reset to $\mathtt{c}^{\triangledown}$. Afterwards, the construction of the chain of $\mathtt{f}$-applications on $\mathtt{x}$ is repeated, only that now the system player can choose between moving the content of $\mathtt{x}$ either to $\mathtt{y}$ or to $\mathtt{z}$. If the goal is to reach $v_{\checkmark}$, then the correct choices depend on the evaluation of $\mathtt{p}$. If $\mathtt{p}$ evaluates positively, then the content of $\mathtt{x}$ should be copied to $\mathtt{y}$. Otherwise, the content of $\mathtt{x}$ should be copied to $\mathtt{z}$. If done correctly, eventually $v_{\checkmark}$ is reached. Otherwise, the play gets stuck in on of the intermediate non-accepting sinks $v_i^{\text{✗}}$ and $\tilde{v}_i^{\text{✗}}$.

We show that Player $O$ indeed has winning strategies $\sigma_n$ in these games for every $n \in \mathbb{N}$. The strategies first play the fix sequence of updates that are required for reaching $\tilde{v}_0$. Then they either copy $\mathtt{x}$ to $\mathtt{y}$ or to $\mathtt{z}$ for the next $n$ iterations, where in every vertex $\tilde{v}_i$, they choose successors such that the resulting play stays in line with $\tilde{v}_0\tilde{v}_1 \dots \tilde{v}_n$ according to the evaluation of $\mathtt{p}$. To this end, Player $O$ can assume that the environment evaluates $\mathtt{p}$ the same way, as earlier in the game at $v_0v_1 \dots v_n$, which Player $O$ can observe through the play's history. Under this assumption, every play that is consistent with these strategies eventually reaches $v_{\checkmark}$. Otherwise, if the assumption is not satisfied by Player $I$, then purity is violated and the system wins as well.

It remains to prove that every winning strategy for Player $O$ requires some memory, which grows at least exponential with the size of the arena. To this end, we claim that every winning strategy needs to distinguish between at least $2^n$ many histories, while telling apart $2^n$ histories is also sufficient. The latter is already witnessed by the winning strategies $\sigma_m$, where the system only must tell apart two choices at every position $v_0, v_1, \dots v_{n-1}$: whether the environment chooses $\langle \mathtt{p}\ \mathtt{y} \rangle \mathrel{\hat{=}} \mathit{true}$ or $\langle \mathtt{p}\ \mathtt{y} \rangle \mathrel{\hat{=}} \mathit{false}$. Note that according to purity $\mathtt{p}\ \mathtt{x}$ and $\mathtt{p}\ \mathtt{y}$ must always evaluate the same at every $v_0, v_1, \dots v_{n-1}$. For proving the former, assume that there is a strategy $\sigma$ that wins by distinguishing less than $2^n$ many histories. Then there is an $0 \leq i < n$ such that $\sigma$ behaves the same, no matter of whether Player $I$ chooses $\langle \mathtt{p}\ \mathtt{y} \rangle \mathrel{\hat{=}} \mathit{true}$ or $\langle \mathtt{p}\ \mathtt{y} \rangle \mathrel{\hat{=}} \mathit{false}$ at $v_i$. Then, let Player $I$ choose $\langle \mathtt{p}\ \mathtt{y} \rangle \mathrel{\hat{=}} \mathit{false}$, if the strategy moves to $\tilde{v}_i^1$ at $\tilde{v}_i$, and $\langle \mathtt{p}\ \mathtt{y} \rangle \mathrel{\hat{=}} \mathit{true}$ otherwise. The resulting play $\rho$ always ends up in $v_i^{\text{✗}}$ and $\tilde{v}_i^{\text{✗}}$. Furthermore, Player $I$ never violates purity. Thus, $\rho$ is not

winning, which disproves our assumption of $\sigma$ being a winning strategy. □

The proof of Theorem 5 shows that the environment player may need infinite memory in order to win a reachability stream game, while Theorem 6 only establishes a lower bound on the memory requirements of winning strategies for Player $O$ depending on the size of the arena. Our next result broadens these observations even further by proving that the memory requirements for both players indeed are not symmetric. It turns out that winning strategies for Player $O$ never need infinite memory in order to win.

**Theorem 7.** *For every reachability stream game that is won by the system player, there exits a winning strategy of finite size for Player $O$.*

*Proof.* Let $\mathcal{G} = (\mathcal{A}, \textsc{StreamReach}(R))$ be a reachability stream game that is won by Player $O$, *i.e.*, there exists a winning strategy $\sigma$. Then according to the winning condition $\textsc{StreamReach}(R) \hat{=} \textsc{Reach}(R) \cup \textsc{Impure}$ every branch $\nu$ of the strategy tree of $\sigma$ eventually reaches a position $v_\nu$ such that either $v_\nu \in R$ or the environment player violates purity at $v_\nu$. Now, let $\sigma'$ be the strategy tree that is cut after $v_\nu$ at every branch $\nu$. We claim $\sigma'$ to be finite. However, for the sake of contradiction we assume the opposite. Then, as $\sigma$ is finitely branching, due to our previous restriction to non-creational TSL, we can apply König's Lemma, which proves the existence of a branch $\nu_{\not\times}$ of $\sigma'$ of infinite length. However, $\nu_{\not\times}$ then also must be a branch of $\sigma$ and cannot contain a position $v$ with either $v \in R$ or the environment player violating purity at $v$. Hence, the branch $\nu_{\not\times}$ instead indicates a winning outcome for Player $I$, which contradicts that $\sigma$ is a winning strategy in the first place. This leads to the desired contradiction proving that $\sigma'$ indeed must be finite. However, note that $\sigma'$ also is a winning strategy, because every branch either reaches $R$ or leads the environment to violate purity, after which the system immediately has won. Thus, the existence of the strategy $\sigma'$ finally concludes the proof. □

While Theorem 6 establishes lower bounds on the memory requirements for strategies of Player $O$ and Theorem 7 establishes the corresponding upper bounds, there still remains a gap for the exact memory requirement for Player $O$. Unfortunately, we leave this gap open for the future at this point, since closing it turned out to be out of scope of this thesis.

Instead, we move on to safety stream games for fixing more lower bounds.

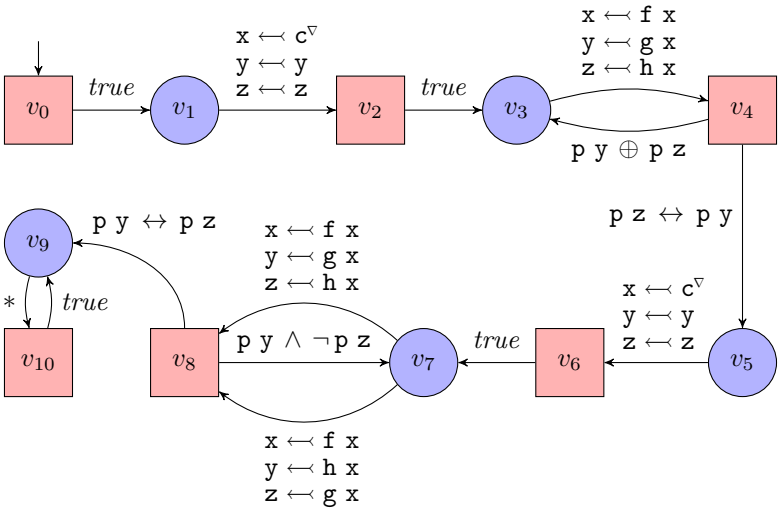



Figure 20: A safety stream game that is won by Player $O$, where every winning strategy requires at least infinite memory. The $\oplus$-operator denotes the Boolean operator for exclusive or.

**Theorem 8.** *There exists a safety stream game that is won by the system player, but every winning strategy requires infinite memory.*

*Proof.* Consider the safety stream game depicted in Figure 20, the goal of the system player is to never leave the vertices $v_0, v_1, \ldots, v_{10}$. To his end, she must manipulate the cells $\mathtt{x}$, $\mathtt{y}$, and $\mathtt{z}$ with the correct application of the constant $\mathtt{c}^{\triangledown}$ and the unary functions $\mathtt{f}$, $\mathtt{g}$, and $\mathtt{h}$.

Initially, the cell $\mathtt{x}$ is reset to $\mathtt{c}^{\triangledown}$ and then updated by $\mathtt{f}\ \mathtt{x}$ at $v_3$ as long as the environment decides to evaluate $\mathtt{p}$ differently on $\mathtt{y}$ and $\mathtt{z}$ at $v_4$, where the values stored in $\mathtt{y}$ and $\mathtt{z}$ are obtained through an additional application of $\mathtt{g}$ and $\mathtt{h}$ to $\mathtt{x}$, respectively. Once the environment player decides to evaluate $\mathtt{p}$ equivalently on both cells, $\mathtt{x}$ is reset to $\mathtt{c}$ and the chain of applications of $\mathtt{f}$ to $\mathtt{x}$ is repeated at $v_7$. Nevertheless, this time the system player can choose between moving $\mathtt{g}\ \mathtt{x}$ to $\mathtt{y}$ and $\mathtt{f}\ \mathtt{x}$ to $\mathtt{z}$ or vice versa. The choice must be taken, however, in such a way such that as long as $\mathtt{p}\ \mathtt{y}$ and $\mathtt{p}\ \mathtt{z}$ are not equivalent $\langle \mathtt{p}\ \mathtt{y} \rangle \mathrel{\hat{=}} \mathit{true}$ and $\langle \mathtt{p}\ \mathtt{y} \rangle \mathrel{\hat{=}} \mathit{false}$ afterwards. Otherwise an unsafe vertex is reached. If $\mathtt{p}\ \mathtt{y}$ and $\mathtt{p}\ \mathtt{z}$ are equivalent eventually, then the game idles at $v_9$ and $v_{10}$ forever.

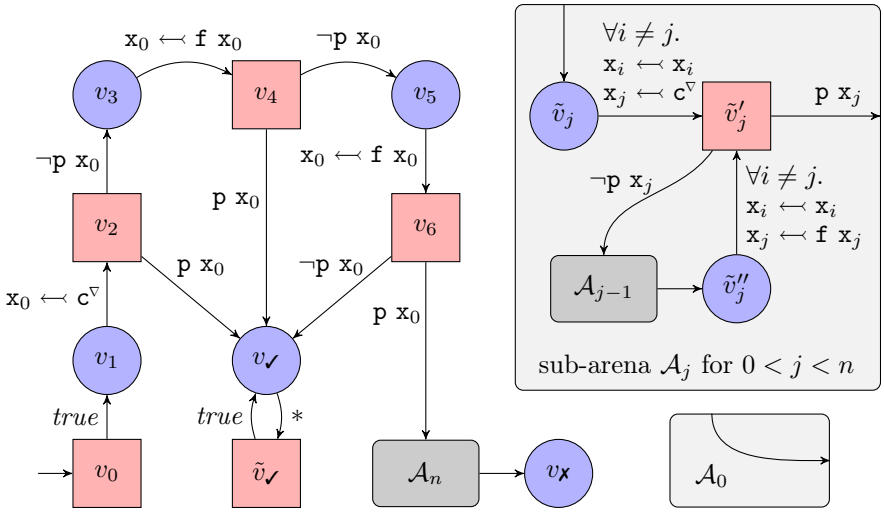



Figure 21: A safety stream game that is won by Player $I$, where all winning strategies require at least exponential memory in the size of the arena.

The system player wins this game. She only must choose the correct successor at $v_7$ such that never $\mathtt{p}\ \mathtt{z} \wedge \neg\mathtt{p}\ \mathtt{y}$ at $v_8$. This choice can always be correctly determined according to the previous choices of Player $I$ at $v_4$, which are accessible via the play's history.

At the same time, however, every winning strategy of the system player needs infinite memory in order to win. Any possible finite memory limitations can be exceeded by the environment, just by heading back to $v_3$ from $v_4$ for long enough until any given limit is exceeded. In this case, the system cannot have enough knowledge of the past to reconstruct the correct choices at $v_7$. Thus, even if the environment satisfies purity, always reaching a safe successor from $v_8$, not always can be guaranteed. Note that $v_8$ reaches an unsafe vertex if $\neg\mathtt{p}\ \mathtt{z}$ and $\mathtt{p}\ \mathtt{y}$, even if the environment satisfies purity. □

Next, we switch players for showing that Player $I$ may need strategies that require memory that is at least exponential in the size of the arena.

**Theorem 9.** *There exists a family of safety stream games that are won by the environment player, but every winning strategy requires memory that grows at least exponentially with the size of the game arena.*

*Proof.* Consider the family of recursively constructed games that is depicted in Figure 21. The boxed sub-arenas describe widgets that are used for the recursive construction of the full game. They are organized according to levels $0 < j \leq n$ with respect to some parameter $n \in \mathbb{N}^+$. On every level $j$, the system player only manipulates the cells $\mathtt{x}_j$ by using the unary function $\mathtt{f}$ and the constant $\mathtt{c}^\triangledown$, while all other cells always remain unchanged. The goal of the environment player is to traverse the whole recursive structure to eventually reach vertex $v_{\text{✗}}$. The size of the overall game arena is linear in $n$.

Before entering the recursively defined structure of $\mathcal{A}_n$, the system initializes $\mathtt{x}_0$ with $\mathtt{c}^\triangledown$, which forces the environment to select $\mathtt{p}$, $\mathtt{f}$, and $\mathtt{c}^\triangledown$ such that it satisfies $\neg\mathtt{p}\ \mathtt{c}^\triangledown$, $\neg\mathtt{p}\ (\mathtt{f}\ \mathtt{c}^\triangledown)$, and $\mathtt{p}\ (\mathtt{f}\ (\mathtt{f}\ \mathtt{c}^\triangledown))$. If Player $I$ does not select $\mathtt{p}$ accordingly, then he looses the game immediately by getting trapped at $v_{\text{✓}}$ and $\tilde{v}_{\text{✓}}$. Afterwards, the recursively defined structure of the arena is entered, where for every instance $j$ the cell $\mathtt{x}_j$ is first reset to $\mathtt{c}^\triangledown_j$. Then, every play must continue in the smaller sub-arena $\mathcal{A}_{j-1}$ as long as $\mathtt{p}\ \mathtt{x}_j$ is not satisfied. If the sub-arena $\mathcal{A}_{j-1}$ has been passed successfully, then the cell $\mathtt{x}_j$ is updated to $\mathtt{f}\ \mathtt{x}_j$ and the procedure is repeated until $\mathtt{p}\ \mathtt{x}_j$ is satisfied.

We show that the environment player has winning strategies $\pi_n$ in the game of Figure 21 for every $n \in \mathbb{N}^+$, which are for example witnessed by the implementations: $\langle \mathtt{c}^\triangledown \rangle := 0$, $\langle \mathtt{f} \rangle\ x := x + 1$, and $\langle \mathtt{p} \rangle\ x := x > 1$. It is easy to see that every strategy, that evaluates predicates according these implementations and updates according to the choices of the system, (1) always avoids $v_{\text{✓}}$, (2) enters every sub-arena $\mathcal{A}_j$ for $0 \leq j < n$ at most twice, and, thus, (3) finally reaches $v_{\text{✗}}$, whose successor is unsafe.

On the other hand, every winning strategy for Player $I$ requires memory that is at least exponential in $n$. For the sake of contradiction assume that there is a winning strategy $\pi$ that only requires to distinguish $m < 2^n$ many histories. First note that for $\pi$ being a winning strategy every outcome $\rho$ of $\pi$ must satisfy purity at every time, which also implies that Player $I$ must satisfy $\neg\mathtt{p}\ \mathtt{c}^\triangledown$, $\neg\mathtt{p}\ (\mathtt{f}\ \mathtt{c}^\triangledown)$, and $\mathtt{p}\ (\mathtt{f}\ (\mathtt{f}\ \mathtt{c}^\triangledown))$. A simple induction on $n \in \mathbb{N}^+$ shows that every vertex $\tilde{v}'_j$ is visited exactly $2^{n-j}$-times for every $0 \leq j \leq n$ by $\rho$, due to every traversal of sub-arenas $\mathcal{A}_j$ always entering the next smaller sub-arena $\mathcal{A}_{j-1}$ twice before leaving $\mathcal{A}_j$. It especially follows that $\tilde{v}'_0$ is visited exactly $2^n$ times during the whole traversal. Then by the pidgin hole principle, there must be at least two visits to $\tilde{v}'_0$, where $\pi$ cannot distinguish the histories of $\rho$ and, thus, proceeds to play equivalently from both positions onwards. Now let $t \in \mathbb{N}$ be the first corresponding visit to $\tilde{v}'_0$ and $t' > t$ be the second one. Then $\rho$ must be of the form $\rho_0\rho_1 \dots (\rho_t\rho_{t+1} \dots \rho_{t'-1})^\omega$ due to the equivalent observations of the history at $\rho_t$ and $\rho_{t'}$. Accordingly, $\rho$ never can reach $v_{\text{✗}}$ leading to the desired contradiction, since $\pi$ cannot be a winning strategy. □

**Theorem 10.** *For every safety stream game that is won by the environment player, there exits a winning strategy of finite size for Player $I$.*

*Proof.* Let $\mathcal{G} = (\mathcal{A}, \textsc{StreamSafety}(S))$ be a safety stream game that is won by the environment player, *i.e.*, there exists a winning strategy $\pi$ for Player $I$. Then according to the winning condition

$$\textsc{StreamSafety}(S) \mathrel{\hat{=}} \textsc{Safety}(S) \cup \textsc{Impure}$$

every outcome of the strategy $\pi$ must satisfy purity and violate safety, *i.e.*, every branch of the strategy tree eventually reaches an unsafe vertex $v_{\text{✗}}$. Let $\pi'$ be the strategy that results from $\pi$ by cutting every branch after it reaches $v_{\text{✗}}$. Due to König's Lemma $\pi'$ must be finite. Furthermore, it still is a winning strategy. Thus, $\pi'$ is a witness for a finite winning strategy for Player $I$. □

Our results show that the memory requirements of safety stream games are exactly dual to the ones of reachability stream games with respect to the two players. We again summarize the resulting upper and lower bounds on memory requirements for safety and reachability stream games in the following table, where $n \in O(|\mathcal{A}|)$ depends on the size of the underlying game arena $\mathcal{A}$.

| | Reachability Stream Games | Safety Stream Games |
|---|---|---|
| Player $I$ | $\lvert\pi\rvert \leq \infty$ | $2^n \leq \lvert\pi\rvert < \infty$ |
| Player $O$ | $2^n \leq \lvert\sigma\rvert < \infty$ | $\lvert\sigma\rvert \leq \infty$ |

Beside reachability and safety, these bounds also impose immediate consequences for the memory requirements of games with more expressive winning conditions, like Büchi, co-Büchi, or parity. Every reachability and safety stream game can be transformed into an equivalent Büchi, co-Büchi, or parity game by adding sink vertices and choosing the acceptance sets and coloring functions accordingly. The conversions work in the same fashion as for regular infinite (non-stream) games. Therefore, it immediately follows that both players may require infinite memory in order to win these games.

**Corollary 1.** *Player $I$ and Player $O$ may need infinite memory in order to win Büchi, co-Büchi and parity stream games.*

# 6 Synthesis

Our previous analysis of TSL game representations reveals that even simple game classes may require infinite memory and that it can be a requirement for both: the winning strategies of environment and the system player. Thus, temporal stream games do not offer an immediate advantage in terms of solving the realizability problem in contrast to the original logic representation of TSL. Consequently, we need other approaches for tackling the problem efficiently. The new challenge that TSL imposes is the added purity requirement of Definition 16, whose full satisfaction leads to undecidability in general, because the pure evaluation of functions and predicates allows the encoding of undecidable problems like Post's correspondence problem (PCP). Thus, in order of being able to handle purity in a satisfactory manner, we need to relax the purity requirements at least to some extend.

Our consideration towards such a relaxation is an approximation of TSL specifications in terms of LTL. This solution has the charm that it works exclusively on the logic level and, thus, completely avoids all of the problems that we encountered with the representation of temporal stream games in Section 5. Another advantage is that, we immediately can hark back on the advanced tools that already have been developed for synthesizing LTL. Nevertheless, we have to consider that such a reduction can never reflect all of the purity properties of TSL. However, as long as we choose the respective encoding carefully enough, it is possible to create a reduction to LTL that is at least sound with respect to a successful synthesis result. In other words, we choose the reduction in such a way, that if the approximate LTL specification is realizable, then the realizing strategy also represents a valid realization of the original TSL specification. On the other hand, if the weaker LTL specification is not realizable, then it can be an artifact of the approximation. In this situation, we need to investigate further to distinguish whether the original specification is indeed unrealizable, or whether the result is a consequence of the approximation. In the latter case, we call the unrealizability result of the approximated LTL specification *spurious*. Unfortunately, determination of the result being spurious inherits the same undecidability properties as realizing TSL, which is why we need another relaxation here as well.

Our solution is a bounded strategy search that limits synthesis to strategies, which additionally satisfy some size restrictions. Checking spuriousness only against strategies up to the imposed bound then becomes realizable. The procedure already has been summarized in Figure 4 of Chapter I, where the inputs to the presented synthesis approach are a TSL specification $\varphi_{TSL}$ and an upper bound on the strategy size $n \in \mathbb{N}$. The TSL specification $\varphi_{TSL}$

then is approximated to a weaker LTL specification $\varphi_{LTL}$, which is passed to a bounded synthesis solver together with the bound $n$. We show that if the LTL solver returns a realizing strategy, then this strategy also realizes the original TSL formula $\varphi_{TSL}$. Otherwise, the counter strategy witnessing unrealizability of the formula $\varphi_{LTL}$ is checked to be spurious. If it is not spurious, the original TSL formula is indeed unrealizable. Otherwise, we obtain another witness for the counter-strategy being spurious, which then is utilized to refine the LTL approximation such that the procedure can be restarted in a CEGAR like fashion. The procedure runs until either a realizing strategy is found or the strategy is proven to be unrealizable. Termination, however, cannot not guaranteed, as the system may win only with an infinite sized strategy, that is never discovered by the bounded synthesis approach. It remains however an open question, whether infinite strategies are indeed necessary in practice and thus would limit our approach.

## 6.1 Initial Purity Approximation

For the initial approximation we turn the syntactic elements $\mathcal{T}_P$ and $\mathcal{T}_\triangleleft$ into atomic propositions in LTL, which removes the semantic meaning of function applications and assignments according to TSL. Afterwards, we reconstruct this meaning lazily by adding assumptions during the refinement.

**Construction 2** (Initial Approximation)**.** Let $\mathcal{T}_P$ and $\mathcal{T}_\triangleleft$ be the finite sets of predicate terms and updates that appear in $\varphi_{TSL}$, respectively. For every assigned output or cell $\mathtt{s_o}$, we partition $\mathcal{T}_\triangleleft$ into $\biguplus_{\mathtt{s_o} \in \mathtt{O} \cup \mathtt{C}} \mathcal{T}_\triangleleft^{\mathtt{s_o}}$. For every $\mathtt{c} \in \mathtt{C}$ let $\mathcal{T}_{\triangleleft/\mathtt{id}}^{\mathtt{c}} = \mathcal{T}_\triangleleft^{\mathtt{c}} \cup \{[\mathtt{c} \leftarrow\!\!\prec \mathtt{c}]\}$ and for every $\mathtt{o} \in \mathtt{O}$ let $\mathcal{T}_{\triangleleft/\mathtt{id}}^{\mathtt{o}} = \mathcal{T}_\triangleleft^{\mathtt{o}}$. We introduce atomic output propositions $\mathcal{T}_{\triangleleft/\mathtt{id}}^{\mathtt{AP}}$ for every update of $\mathcal{T}_{\triangleleft/\mathtt{id}}^{\mathtt{s_o}}$ with $\mathtt{s_o} \in \mathtt{O} \cup \mathtt{C}$, *i.e.*, $\mathcal{T}_{\triangleleft/\mathtt{id}}^{\mathtt{AP}} := \bigcup_{\mathtt{s_o} \in \mathtt{O} \cup \mathtt{C}} \bigcup_{\tau \in \mathcal{T}_{\triangleleft/\mathtt{id}}^{\mathtt{s_o}}} a_\tau$. Similarly, we create atomic input propositions $\mathcal{T}_P^{\mathtt{AP}}$ for every predicate term of $\mathcal{T}_P$, *i.e.*, $\mathcal{T}_P^{\mathtt{AP}} := \bigcup_{\tau_P \in \mathcal{T}_P} a_{\tau_P}$. The LTL formula $\varphi_{LTL}$ then is constructed over the input propositions $\mathcal{T}_P^{\mathtt{AP}}$ and output propositions $\mathcal{T}_{\triangleleft/\mathtt{id}}^{\mathtt{AP}}$, where we utilize a rewrite function $rw$ that replaces every predicate term $\tau \in \mathcal{T}_P$ and update term $\tau \in \mathcal{T}_\triangleleft$ of the TSL formula $\varphi_{TSL}$ with an atomic proposition $a_\tau \in \mathcal{T}_P^{\mathtt{AP}} \cup \mathcal{T}_{\triangleleft/\mathtt{id}}^{\mathtt{AP}}$, respectively. Formally:

$$\varphi_{LTL} = \square\Big( \bigwedge_{\mathtt{s_o} \in \mathtt{O} \cup \mathtt{C}} \bigvee_{\tau \in \mathcal{T}_{\triangleleft/\mathtt{id}}^{\mathtt{s_o}}} \big(a_\tau \wedge \bigwedge_{\tau' \in \mathcal{T}_{\triangleleft/\mathtt{id}}^{\mathtt{s_o}} \setminus \{\tau\}} \neg a_{\tau'}\big)\Big) \wedge rw(\varphi_{TSL})$$

The first part of the construction of $\varphi_{LTL}$ partially reconstructs the semantic meaning of updates by ensuring that a signal is not updated with multiple values at a time. The second part extracts the reactive constraints of the TSL formula without the semantic meaning of functions and updates.

**Theorem 11.** *If $\varphi_{LTL}$ is realizable, then $\varphi_{TSL}$ is realizable.*

*Proof.* Assume $\varphi_{LTL}$ is realizable. Then there exists a winning strategy $\sigma\colon (2^{\mathcal{T}_P^{\mathtt{AP}}})^+ \to 2^{\mathcal{T}_{\lhd/\mathtt{id}}^{\mathtt{AP}}}$ for the system player in the underlying LTL realizability game. Furthermore, for the sake of contradiction assume that $\varphi_{TSL}$ is not realizable. Then for all $\kappa\colon (2^{\mathcal{T}_P})^+ \to \mathcal{C}$ there exists in input $\iota \in \mathcal{I}^\omega$ and a function assignment $\langle\cdot\rangle\colon \mathcal{T}_F \to \mathcal{F}$ such that $\kappa \between_{\langle\rangle} \iota, \iota \not\models_{\langle\rangle} \varphi_{TSL}$. We inductively construct the input sequence $\nu \in (2^{\mathcal{T}_P^{\mathtt{AP}}})^\omega$ and the computation $\varsigma \in \mathcal{C}^\omega$ over $t \in \mathbb{N}$ as follows:

$$\begin{aligned} \nu(t) &= \{a_{\tau_P} \in \mathcal{T}_P^{\mathtt{AP}} \mid \eta_{\langle\rangle}(\varsigma, \iota, t, \tau_P)\} \\ \varsigma(t)(\mathtt{s}) &= \tau_F, \quad \text{where } \tau_F \text{ is the unique element such that } a_{[\mathtt{s} \leftarrow \tau_F]} \in \sigma(\nu(0)\nu(1)\dots\nu(t)) \end{aligned}$$

Note that $\tau_F$ must be unique, due to the additional constraint:

$$\square\Big(\bigwedge_{\mathtt{s}\in\mathtt{O}} \bigvee_{\tau\in\mathcal{T}^{\mathtt{s}}_{\lhd/\mathtt{id}}} \big(a_\tau \wedge \bigwedge_{\tau'\in\mathcal{T}^{\mathtt{s}}_{\lhd/\mathtt{id}}\setminus\{\tau\}} \neg\, a_{\tau'}\big)\Big)$$

Furthermore, note that $\nu$ and $\varsigma$ are well-defined, since $\eta_{\langle\rangle}(\varsigma, \iota, t, \tau_P)$ only considers values of $\varsigma$ at previous times $t' < t$. Since $\varphi_{LTL}$ is realizable, we have that $\sigma \between \nu, \nu \vDash \varphi_{LTL}$, but at the same time $\varsigma, \iota \not\models_{\langle\rangle} \varphi_{TSL}$ due to unrealizability of $\varphi_{TSL}$. We show that this is contradictory via a structural induction over the structure of $\varphi_{TSL}$ for all $t \in \mathbb{N}$:

**Case:** $\varphi_{TSL} = \tau_P$

$$\begin{aligned} & \varsigma, \iota, t \models_{\langle\rangle} \tau_P \\ \Leftrightarrow\ & \eta_{\langle\rangle}(\varsigma, \iota, t, \tau_P) \\ \Leftrightarrow\ & a_{\tau_P} \in \nu(t) \\ \Leftrightarrow\ & \sigma \between \nu, \nu, t \vDash a_{\tau_P} \\ \Leftrightarrow\ & \sigma \between \nu, \nu, t \vDash rw(\tau_P) \end{aligned}$$

**Case:** $\varphi_{TSL} = [\,\mathtt{s} \leftarrowtail \tau_F\,]$

$$
\begin{array}{ll}
& \varsigma, \iota, t \;\vDash_{\langle\rangle}\; [\,\mathtt{s} \leftarrowtail \tau_F\,] \\
\Leftrightarrow & \varsigma(t)(\mathtt{s}) \equiv \tau_F \\
\Leftrightarrow & a_{[\,\mathtt{s}\leftarrowtail\tau_F\,]} \in \sigma(\nu(0)\nu(1)\ldots\nu(t)) \\
\Leftrightarrow & \sigma \wr \nu, \nu, t \vDash a_{[\mathtt{s}\leftarrowtail\tau_F]} \\
\Leftrightarrow & \sigma \wr \nu, \nu, t \vDash rw([\,\mathtt{s} \leftarrowtail \tau_F\,])
\end{array}
$$

**Case:** $\varphi_{TSL} = \neg\psi$

$$
\begin{array}{ll}
& \varsigma, \iota, t \;\vDash_{\langle\rangle}\; \neg\psi \\
\Leftrightarrow & \varsigma, \iota, t \;\nvDash_{\langle\rangle}\; \psi \\
\overset{IH}{\Leftrightarrow} & \sigma \wr \nu, \nu, t \nvDash rw(\psi) \\
\Leftrightarrow & \sigma \wr \nu, \nu, t \vDash \neg\, rw(\psi) \\
\Leftrightarrow & \sigma \wr \nu, \nu, t \vDash rw(\neg\psi)
\end{array}
$$

**Case:** $\varphi_{TSL} = \vartheta \wedge \psi$

$$
\begin{array}{ll}
& \varsigma, \iota, t \;\vDash_{\langle\rangle}\; \vartheta \wedge \psi \\
\Leftrightarrow & \varsigma, \iota, t \;\vDash_{\langle\rangle}\; \vartheta \;\wedge\; \varsigma, \iota, t \vDash \psi \\
\overset{IH}{\Leftrightarrow} & \sigma \wr \nu, \nu, t \vDash rw(\vartheta) \;\wedge\; \sigma \wr \nu, \nu, t \vDash rw(\psi) \\
\Leftrightarrow & \sigma \wr \nu, \nu, t \vDash rw(\vartheta) \wedge rw(\psi) \\
\Leftrightarrow & \sigma \wr \nu, \nu, t \vDash rw(\vartheta \wedge \psi)
\end{array}
$$

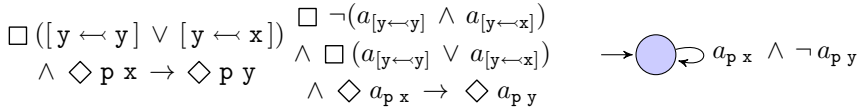
**Case:** $\varphi_{TSL} = \bigcirc\psi$

$$
\begin{array}{ll}
& \varsigma, \iota, t \;\vDash_{\langle\rangle}\; \bigcirc\psi \\
\Leftrightarrow & \varsigma, \iota, t+1 \;\vDash_{\langle\rangle}\; \psi \\
\overset{IH}{\Leftrightarrow} & \sigma \wr \nu, \nu, t+1 \vDash rw(\psi) \\
\Leftrightarrow & \sigma \wr \nu, \nu, t \vDash \bigcirc rw(\psi) \\
\Leftrightarrow & \sigma \wr \nu, \nu, t \vDash rw(\bigcirc\psi)
\end{array}
$$

**Case:** $\varphi_{TSL} = \vartheta\,\mathcal{U}\,\psi$

$$
\begin{array}{ll}
& \varsigma, \iota, t \;\vDash_{\langle\rangle}\; \vartheta\,\mathcal{U}\,\psi \\
\Leftrightarrow & \exists t'' \geq t.\;\; \forall t \leq t' < t''.\;\; \varsigma, \iota, t' \;\vDash_{\langle\rangle}\; \vartheta \;\;\wedge\;\; \varsigma, \iota, t'' \;\vDash_{\langle\rangle}\; \psi \\
\overset{IH}{\Leftrightarrow} & \exists t'' \geq t.\;\; \forall t \leq t' < t''.\;\; \sigma \wr \nu, \nu, t' \vDash rw(\vartheta) \;\;\wedge\;\; \sigma \wr \nu, \nu, t'' \vDash rw(\psi) \\
\Leftrightarrow & \sigma \wr \nu, \nu, t \vDash rw(\vartheta)\,\mathcal{U}\,rw(\psi) \\
\Leftrightarrow & \sigma \wr \nu, \nu, t \vDash rw(\vartheta\,\mathcal{U}\,\psi)
\end{array}
$$

□




(a) TSL specification (b) initial approximation (c) spurious counter-strategy

Figure 22: A realizable TSL specification (a) with input `x` and cell `y`. A winning strategy is given by saving `x` to `y` as soon as `p x` is satisfied. However, the initial approximation (b) is unrealizable, as proven by the counter-strategy (c).

Note that unrealizability of $\varphi_{LTL}$ does not imply that $\varphi_{TSL}$ is unrealizable. It may be that we have not added sufficiently many environment assumptions to the approximation in order for the system to produce a realizing strategy.

Some more concrete insights into the implications of the approximation are given through the example of Figure 22. The specification (a) asserts that the environment provides an input `x` for which the predicate `p x` will be satisfied eventually. At the same time, the system must guarantee that `p y` holds eventually too. The specification of (a) is realizable. The system can take the value of `x` as soon as `p x` is satisfied and then stores it in `y`. This way it guarantees that `p y` is satisfied eventually. However, the situation changes as soon as we consider the approximated LTL specification (b) that results from Construction 2. There, the semantics of the pure function `p` is lost. Instead, the evaluation of `p y` is reduced to an environmentally controlled value $a_{\texttt{p y}}$ that does not need to obey the consistency of the pure function `p`. As a consequence, the approximation becomes unrealizable, as witnessed by the spurious counter-strategy (c).

## 6.2 Refining the Approximation

As highlighted in the example above, it is possible that LTL synthesis returns a counter-strategy for the environment although the original TSL specification is realizable. We call such a counter-strategy *spurious* as it exploits the missing semantic restrictions that have been relaxed through the approximation. The result is a violation of the purity of functions and predicates.

From a more formal point of view, the LTL synthesizer returns a counter-strategy $\pi\colon (2^{\mathcal{T}^{\mathrm{AP}}_{\lhd/\mathrm{id}}})^* \to 2^{\mathcal{T}^{\mathrm{AP}}_{P}}$. This strategy can also be formulated as a strategy $\pi\colon \mathcal{C}^* \to 2^{\mathcal{T}_P}$ with respect to the additional properties that have been introduced in Construction 2. The strategy $\pi$ determines the predicate evaluations in response to possible update assignments of function terms $\tau_F \in \mathcal{T}_F$

to outputs $\mathtt{o} \in \mathtt{O}$, where w.l.o.g. we can assume that $\mathtt{O}$, $\mathcal{T}_F$ and $\mathcal{T}_P$ are finite, as they always can be restricted to the outputs and terms that appear in the formula. A counter-strategy then is spurious, if there is a branch $\pi \wr \varsigma$ for some $\varsigma \in \mathcal{C}^\omega$, for which the strategy chooses an inconsistent evaluation of two equal predicate terms at different points in time.

**Definition 17.** A counter-strategy $\pi \colon \mathcal{C}^* \to 2^{\mathcal{T}_P}$ is *spurious*, iff

$$\begin{aligned}&\exists \varsigma \in \mathcal{C}^\omega.\ \exists t, t' \in \mathbb{N}.\ \exists \tau_P, \tau'_P \in \mathcal{T}_P.\\ &\quad \tau_P \in \pi(\varsigma(0)\varsigma(1)\dots\varsigma(t-1)) \wedge \tau'_P \notin \pi(\varsigma(0)\varsigma(1)\dots\varsigma(t'-1)) \wedge\\ &\quad \forall \langle\cdot\rangle \colon \mathtt{F} \to \mathcal{F}.\ \eta_{\langle\cdot\rangle}(\varsigma, \pi \wr \varsigma, t, \tau_P) = \eta_{\langle\cdot\rangle}(\varsigma, \pi \wr \varsigma, t', \tau'_P).\end{aligned}$$

Note that a non-spurious strategy can be inconsistent along multiple branches, since according to the definition of realizability the environment can choose function and predicate assignments differently against every system strategy.

Due to the purity of predicates in TSL the environment is forced to always return the same value for predicate evaluations on equal inputs. However, this semantic property cannot be enforced implicitly in LTL. To resolve this issue we use the returned counter-strategy to identify spurious behavior in order to strengthen the LTL underapproximation with additional environment assumptions. After adding the derived assumptions, we re-execute the LTL synthesis in order to check whether the added assumptions are sufficient for obtaining a winning strategy for the system. If the solver still returns a spurious strategy, we continue the loop in a CEGAR like fashion until the set of added assumptions is sufficiently complete. However, if a non-spurious strategy is returned, we have a proof that the given TSL specification is indeed unrealizable and terminate.

We use Algorithm 1 to determine, whether a returned counter-strategy $\pi$ is spurious or not. The algorithm relies on $\pi$ being checked against system strategies that are bounded by the given bound $b$. However, this dependency is negligible, as long as we use bounded synthesis [45] as the underlying LTL synthesis approach. We also can assume that $\pi$ is always given as a finite state representation due to the finite model guarantee of LTL.

The algorithm iterates over all possible responses $v \in \mathcal{C}^{m \cdot b}$ of the system up to depth $m \cdot b$. This is sufficient, since every deeper exploration would result in a state repetition of the cross-product of the finite state representation of $\pi$ and any system strategy bounded by $b$. Hence, the same behaviour could also be generated by a sequence smaller than $m \cdot b$. At the same time, the algorithm iterates over predicates $\tau_P, \tau'_P \in \mathcal{T}_P$ appearing in $\varphi_{TSL}$ and times $t$

**Algorithm 1** Check-Spuriousness

**Input:** bound $b$, counter-strategy $\pi\colon \mathcal{C}^* \to 2^{\mathcal{T}_P}$ (finitely represented using $m$ states)

1: **for all** $v \in \mathcal{C}^{m \cdot b}$, $\tau_P \in \mathcal{T}_P$, $t, t' \in \{0, 1, \dots, m \cdot b - 1\}$ **do**
2: **if** $\eta_{\langle\cdot\rangle_{\mathtt{id}}}(v, \iota_{\mathtt{id}}, t, \tau_P) \equiv \eta_{\langle\cdot\rangle_{\mathtt{id}}}(v, \iota_{\mathtt{id}}, t', \tau_P) \wedge$ $\tau_P \in \pi(v_0 \dots v_{t-1}) \wedge \tau_P \notin \pi(v_0 \dots v_{t'-1})$ **then**
3: $w \leftarrow \mathit{reduce}(v, \tau_P, t, t')$
4: **return** $\square\big(\bigwedge_{i=0}^{t-1} \bigcirc^i w_i \wedge \bigwedge_{i=0}^{t'-1} \bigcirc^i w_i \rightarrow (\bigcirc^t \tau_P \leftrightarrow \bigcirc^{t'} \tau_P)\big)$
5: **return** "non-spurious"

and $t'$ smaller than $m \cdot b$. For each of these elements, spuriousness is checked by comparing the output of $\pi$ for the evaluation of $\tau_P$ and $\tau'_P$ at times $t$ and $t'$, which only differs, if the inputs to the predicates are different as well. This only happens, if the passed input terms have been constructed differently over the past. We check it by using the evaluation function $\eta$ equipped with the identity assignment $\langle\cdot\rangle_{\mathtt{id}}\colon \mathtt{F} \to \mathtt{F}$, with $\langle \mathtt{f} \rangle_{\mathtt{id}} = \mathtt{f}$ for all $\mathtt{f} \in \mathtt{F}$, and the input sequence $\iota_{\mathtt{id}}$, with $\iota_{\mathtt{id}}(t)(\mathtt{i}) = (t, \mathtt{i})$ for all $t \in \mathbb{N}$ and $\mathtt{i} \in \mathtt{I}$, that always generates a fresh input. Syntactic inequality of $\eta_{\langle\cdot\rangle_{\mathtt{id}}}(v, \iota_{\mathtt{id}}, t, \tau_P)$ and $\eta_{\langle\cdot\rangle_{\mathtt{id}}}(v, \iota_{\mathtt{id}}, t', \tau'_P)$ then is a sufficient condition for the existence of an assignment $\langle\cdot\rangle\colon \mathtt{F} \to \mathcal{F}$, for which $\tau_P$ and $\tau'_P$ evaluate differently at times $t$ and $t'$.

If spurious behaviour of $\pi$ is found, then the revealing response $v \in \mathcal{C}^*$ is first simplified using *reduce*, a function that reduces $v$ again to a sequence of sets of updates $w \in (2^{\mathcal{T}_{\lhd/\mathtt{id}}})^*$ and removes updates that do not affect the behavior of $\tau_P$ at times $t$ and $t'$ to accelerate the termination of the CEGAR loop. Afterwards, the sequence $w$ is turned into a new assumption prohibiting the spurious behavior, generalized even for arbitrary points in time. This assumption then is added to the previously approximated LTL formula and the synthesis process is started once again. The whole CEGAR loop continues until either a realizing strategy or a non-spurious counter-strategy is found.

For an example of the process, reconsider the spurious counter-strategy of Figure 22c. Already after the first system response $a_{[\mathtt{y} \leftarrow\!\lhd \mathtt{x}]}$, the environment produces a purity violation by evaluating $a_{\mathtt{p\,x}}$ and $a_{\mathtt{p\,y}}$ differently. Purity is violated, since the cell $\mathtt{y}$ holds the same value at time $t = 1$ as the input $\mathtt{x}$ at time $t = 0$. After processing the strategy with Algorithm 1, a new assumption $\square([\,\mathtt{y} \leftarrow\!\lhd\, \mathtt{x}\,] \rightarrow (\mathtt{p\ x} \leftrightarrow \bigcirc \mathtt{p\ y}))$ is generated that can be used to strengthen the LTL approximation. The generated assumption then is added to the initial approximation and synthesis is re-executed. Indeed, the updated LTL formula now turns out to be realizable yielding the aforementioned TSL winning strategy.

## 6.3 Synthesizing Control Flow

With the CEGAR based TSL synthesis method at hand, we finally have a reliable approach for synthesizing TSL specifications. However, in order to produce real-world applications, like the kitchen timer or the music player, it still remains to turn the synthesized strategy into executable code. To this end, we provide a modular approach, where we leverage an intermediate control flow model. This model not only covers all control flow decisions of the returned strategy, as well as the utilized functions and predicates, but also introduces an intermediate level of abstraction that is compatible with different real-world execution engines. In this way, we accomplish a more flexible environment for the system designer. On the one hand, the TSL system design gets efficiently decoupled from control flow independent read-world system specifics on the specification level. On the other hand, the synthesized control flow is applicable to multiple application frameworks and, thus, also allows to postpone the determination of the finally used application context, even after the control flow has been specified and synthesized. To this end, it then is also possible to use the same synthesized control flow for different application domains.

Remember that concrete function and predicate implementations are also not fixed yet. Instead, they are considered as literals only, without any particular pre-assigned semantics. This exactly comes along with our initial consideration of separating data and control. According to this separation, the purpose of the synthesized control flow is purely to determine the respective control, but abstracts from the actual data. The conrectization of the data and their transformations thus must come afterwards. Furthermore, the approach also accompanies with the idea of a modular refinement according to the system design, where we start with the control flow as the high level entry point and then concretize the design towards the system specifics.

If synthesis is successful, it produces a *Control Flow Model* (CFM) $\mathcal{M}$ that satisfies the given TSL specification $\varphi$. It represents the control flow structure of the final program. Intuitively, the model can be considered as flow chart or network, that passes input data to cells or outputs over temporally different enabled connections. Along these connections the stream data is manipulated by pure data transforming functions and checked by pure predicates. The overall flow of the data through the network thus is not static over time. Instead, it depends on the evaluation of predicates guiding the data through corresponding transformations on its way to outputs or cells. Formally, this leads to the following definition.

**Definition 18.** A *CFM* $\mathcal{M}$ is a tuple $\mathcal{M} = (\mathtt{I}, \mathtt{O}, \mathtt{C}, \mathtt{F}, V, \ell_{\mathcal{M}}, \delta_{\mathcal{M}})$ where

- $\mathtt{I}$ is a finite set of input literals,
- $\mathtt{O}$ is a finite set of output literals,
- $\mathtt{C}$ is a finite set of cells,
- $\mathtt{F}$ is a finite set of function literals,
- $V$ is a finite set of vertices,
- $\ell_{\mathcal{M}} \colon V \to \mathtt{F} \cup \mathcal{F}$ is labeling function assigning each vertex either a function literal or a function, and
- $\delta_{\mathcal{M}} \colon \big(\mathtt{O} \cup \mathtt{C} \to V \cup \mathtt{I} \cup \mathtt{C}\big) \cup \big(V \to \bigcup_{n \in \mathbb{N}} (V \cup \mathtt{I} \cup \mathtt{C})^n\big)$ is a dependency relation that relates outputs, cells, and vertices to inputs, cells, and vertices, where the vertex relation must match with the arity of the vertex label, *i.e.*, $\forall v \in V.\, \sharp(\ell_{\mathcal{M}}(v)) = \sharp(\delta_{\mathcal{M}}(v))$.

We assume w.l.o.g. that the sets $\mathtt{I}$, $\mathtt{O}$, $\mathtt{C}$, $\mathtt{F}$, and $V$ are pairwise disjoint. Furthermore, we require that the dependency relation $\delta_{\mathcal{M}}$ does not induce circular dependencies on $V$, *i.e.*, for every CFM there must be a ranking $r \colon V \to \mathbb{N}$ such that for all $v \in V$ and $j \in [\sharp(\ell_{\mathcal{M}}(v))]$, *i.e.*, if $pr_j(\delta_{\mathcal{M}}(v)) \in V$, then also $r(v) > r(pr_j(\delta_{\mathcal{M}}(v)))$.

Note that the above definition allows to mix uninterpreted function literals $\mathtt{F}$ with already determined functions $f \in \mathcal{F}$ as part of the vertex labeling of $V$. The functions $f$ are used to implement the data independent control flow, while keeping the data dependent transformations $\mathtt{F}$ abstract.

Regarding the underlying concept, we implicitly require that functions $f \in \mathcal{F}$ only impose structural changes to the data and do not depend on specific data types, such as numbers or strings, in the first place. In functional programming languages, these transformations are well known as *polymorphic* transformations, because they are applicable to any possible data kind. Classical examples are functions that restructure data within containers, such as tuples, lists or sets. In this sense, functional programming languages use a similar concept of separating data and control as leveraged for TSL. Hence, if we would consider extending the definition of a CFM with a type system, which equips the input, output and function literals, as well as the cells with types, then every CFM should be purely composed out of functions using

only Boolean and polymorphic types. However, since a formal introduction of such a type system and the corresponding backgrounds on the underlying type theory would go beyond the scope of this thesis and, moreover, would not add any value to the remaining considerations, we postpone a detailed analysis of such an extension to future work at this point.

Nevertheless, regarding the uninterpreted function literals $\mathtt{F}$, the CFM provides an implementation model for the program control that satisfies the given TSL specification independently of their concrete implementations. In other words, the CFM serves as a finite model for implementing the targeted TSL strategies $\sigma\colon (2^{\mathcal{T}_P})^+ \to \mathcal{C}$. This correspondence is formalized using the following transformation.

**Construction 3.** Every CFM $\mathcal{M} = (\mathtt{I}, \mathtt{O}, \mathtt{C}, \mathtt{F}, V, \ell_{\mathcal{M}}, \delta_{\mathcal{M}})$ induces a strategy $\sigma_{\mathcal{M}}$ with $\sigma_{\mathcal{M}}(w\nu)(x) = \eta_{\mathcal{M}}(\nu, \delta_{\mathcal{M}}(x))$ using the utility function $\eta_{\mathcal{M}}\colon 2^{\mathcal{T}_P} \times (V \cup \mathtt{I} \cup \mathtt{C}) \to \mathcal{T}_F$, which is defined as:

$$\eta_{\mathcal{M}}(\nu, x) = \begin{cases} x & \text{if } x \in \mathtt{I} \cup \mathtt{C} \\ \mathtt{f}\ \eta_{\mathcal{M}}(v_0) \cdots \eta_{\mathcal{M}}(v_{k-1}) & \text{if } x \in V \text{ and } \ell_{\mathcal{M}}(x) = \mathtt{f} \text{ with } \mathtt{f} \in \mathtt{F} \setminus \mathtt{P} \text{ and } \delta_{\mathcal{M}}(x) = (v_0, \ldots, v_{k-1}) \\ (\mathtt{p}\ \eta_{\mathcal{M}}(v_0) \cdots \eta_{\mathcal{M}}(v_{k-1})) \in \nu & \text{if } x \in V \text{ and } \ell_{\mathcal{M}}(x) = \mathtt{p} \text{ with } \mathtt{p} \in \mathtt{P} \text{ and } \delta_{\mathcal{M}}(x) = (v_0, \ldots, v_{k-1}) \\ f\ \eta_{\mathcal{M}}(v_0) \cdots \eta_{\mathcal{M}}(v_{k-1}) & \text{if } x \in V \text{ and } \ell_{\mathcal{M}}(x) = f \text{ with } f \in \mathcal{F} \text{ and } \delta_{\mathcal{M}}(x) = (v_0, \ldots, v_{k-1}) \end{cases}$$

The construction shows that the model of a CFM indeed provides a suitable representation for winning strategies with respect to TSL. Thus, it only remains to bridge the connection of how to get a CFM in the context of our approximation approach. Therefore, reconsider the LTL formula $\varphi_{LTL}$ constructed from a TSL specification $\varphi_{TSL}$ according to Construction 2.

**Theorem 12.** *If $\varphi_{LTL}$ is realizable, then there exists a CFM $\mathcal{M}$ that implements $\varphi_{TSL}$, i.e., $\sigma_{\mathbb{M}}$ satisfies $\varphi_{TSL}$.*

*Proof.* Let $\sigma\colon (2^{\mathcal{T}_P^{\mathtt{AP}}})^+ \to 2^{\mathcal{T}_{\lhd/\mathtt{id}}^{\mathtt{AP}}}$ be a realizing strategy for $\varphi_{LTL}$. W.l.o.g. we can assume that the strategy is given as a circuit, consisting of a finite number of latches, AND, OR, and NOT gates, since the specified system of every realizable LTL formula can be implemented by a finite Mealy machine. Furthermore, circuits and Mealy machines are equally expressive.

Next consider: it is straightforward to integrate circuits as part of the CFM. Latches can be implemented with cells. The different gate types can be expressed using pure functions $f_\wedge$, $f_\vee$, and $f_\neg$. Hence, we only need to copy the circuit graph to be part of the dependency relation, such that the CFM vertices are labeled with the matching gate functions like they appear in the original circuit. After then having the circuit as part of the CFM, we add the function transformations, as they appear in the terms of $\mathcal{T}_P^{\mathtt{AP}}$ and $\mathcal{T}_{\lhd/\mathtt{id}}^{\mathtt{AP}}$. Note that every corresponding term induces a finite DAG that links every function literal to the arguments it is applied to. Therefore, using the dependency relation $\delta_{\mathcal{M}}$ and the labeling $\ell_{\mathcal{M}}$, this DAG thus can be easily expressed using the graph structure of the CFM.

Finally, the puzzle gets completed by correctly connecting functions and predicates with the circuit structure. Therefore, the inputs of the circuit structure are linked to the vertices labeled with the corresponding predicate terms. Intuitively, the connection expresses that the results of the predicate evaluations are passed to the circuit inputs. At the other end, the outputs of the circuits are connected to vertices labeled with functions that select the computed function transformations, as enabled at the current point in time according to the circuit evaluation. Formally, the corresponding function labels can be defined for all $n \in \mathbb{N}$ by

$$\mathit{select}_n\ b_0\ x_0\ b_1\ x_1 \cdots b_{n-1}\ x_{n-1} = \begin{cases} x_0 & \text{if } b_0 \\ x_1 & \text{if } b_1 \\ \vdots & \vdots \\ x_{n-1} & \text{if } b_{n-1} \end{cases}$$

and are chosen such that they match the corresponding number of output choices as given by $\mathcal{T}_{\lhd/\mathtt{id}}^{\mathtt{AP}}$. Note that the mutual exclusion property, as introduced in Construction 2, ensures that the selected updates are always unique.

It is easy to observe that the created structure indeed witnesses the existence of a realizing strategy for $\varphi_{TSL}$ as stated in Theorem 11. On the one hand, the direct embedding of the realizing circuit of $\varphi_{LTL}$ clearly induces a similar temporal behavior. On the other hand, the matching embedding of the term structure in combination with the $\mathit{select}_n$ components clearly fits the semantic selection, as induced by the corresponding update terms. □

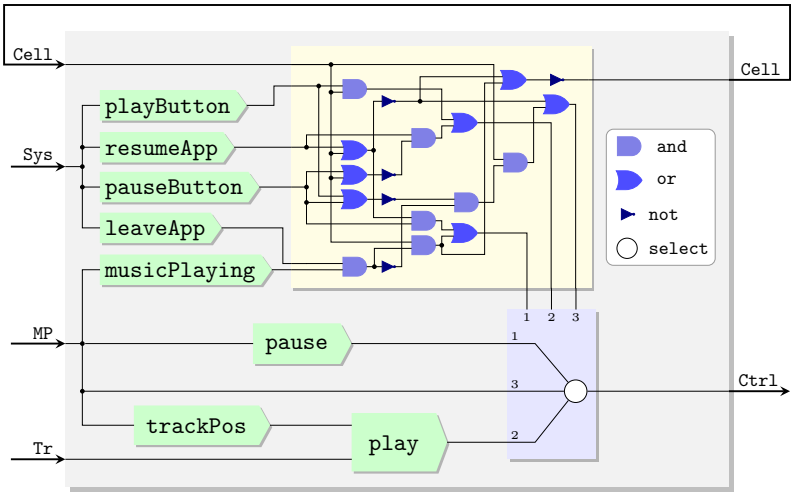



Figure 23: CFM that implements the music player specification.

An example for a CFM, which results from synthesizing the music player specification of Section 2.2, is depicted in Figure 23. The inputs enter the system from the left and the produced outputs leave it on the right. The example only requires a single cell, which originates from the circuit structure and therefore has been introduced during synthesis. The green, arrow shaped boxes denote vertices being labeled with function and predicate literals. The Boolean operations $f_\wedge$, $f_\vee$, and $f_\neg$ are depicted with the corresponding circuit symbols for conjunction, disjunction, and negation. The Boolean outputs of the circuit are piped to the selectors, which forward the corresponding value streams according to the chosen update behavior, *i.e.*, each update stream is passed to an output stream if and only if the respective Boolean trigger evaluates correspondingly.

The CFM instantiates the control behavior of the system, while still abstracting form concrete data transformations. These data transformations are only indicated using the corresponding function and predicate literals instead, where the synthesis guarantees that the desired control behavior is always executed correctly, independently of how the functions are implemented in the end. Hence, regarding the overall system design process, the developer solely has to provide the remaining function implementations. Furthermore, an execution engine must be chosen to executes the CFM in combination with the selected function implementations. Corresponding frameworks for such engines that satisfy the respective purity and temporal requirements can be found in the research field of *Functional Reactive Programming*.

# 7 Functional Reactive Programming

The classical models for reactive systems, as presented in Section 1, feature a rich number of capabilities for expressing temporal behavior in combination with functional processing requirements. Nevertheless, they still miss some important features, when it comes to the practical application of synthesis in real world development scenarios.

To understand these missing features at first, reconsider the capabilities of classical implementation models, such as Mealy machines or circuits. There, it is easy to observe that all of these models only deliver a quite rudimentary model for the representation and transformation of internal state. State is either represented explicitly, as a unique configuration of the system, which is inspected through a global view, or given as a set of variables or latches holding purely Boolean values. Similarly, state is transformed using a global transition function or using simple local Boolean transformations. Therefore, the view discards any insights into the internal structure of the system through reducing the structure to a monolithic model of uniform transformations. Latches and variables, on the other hand, support to give structure to the state and the corresponding transformations. However, therefore they are limited to the Boolean data level of operation only.

We can conclude that these classical models clearly miss some advanced abstraction techniques, as they are already used in programming languages today. Hence, we need to ask ourselves: why exactly are we not using such modern programming languages to implement reactive systems already? The answer is that traditional programming languages usually are not designed to be executed in a reactive environment. They instead rely on the assumption of sequential execution, as well as on a step by step transformation for transfering only single inputs to an output at a time. Reactivity then usually is just added on top of this assumption, resulting in thread models, event handlers, or other instances of such kind, which however often only offer poor guarantees with respect to a robust temporal behavior.

What we have learned, however, from these models is that the infinite interaction and the correct handling of time are the key challenges for creating correct and robust reactive systems in the end. Hence, they should be included as part of the core models of reactive programming languages as well.

## 7.1 Paradigm

A first advance into this direction has been introduced using so called *reactive programming languages*. Their core idea is that a program does not represent

a sequence of instructions, executed one after another, but instead represents a network of dependencies that describe how output is computed depending on the given input. Therefore, the model describes a network of data flows, piping input values through different transducers, which finally produces data at be output at the end. Correspondingly, the model is natively suited for the description of reactive systems, where the network transforms infinite streams of inputs into infinite output streams. Thus, by using the reactive programming paradigm, the classical model of reactive systems can be natively lifted from the Boolean data level to arbitrary data streams.

There exist many models for reactive programs, but the mathematically most appealing ones are suited under the category of *functional reactive programs* (FRP) [36, 32]. As the name adumbrates, functional reactive programming also encourages to leverage a clean separation between data transformation, as denoted by pure mathematical function applications, and the temporal evolution of state within the transducer network. Furthermore, functional reactive programming also features a fully flexible model of time, which even covers the spectrum of continuous time in some of the frameworks. Unfortunately, there is no canonical model of FRP. Instead, practice has driven some diverge development resulting in different incarnations of FRP implementations that often are part of application specific libraries [142, 5, 141, 32, 147, 7, 4]. None of these libraries is based on a uniform reactive system model. Instead, they utilize denotative semantics and design patterns from existing host languages, like the programming language *Haskell* [66]. While using denotative semantics feature the direct expression of the natural intuition behind the individual design primitives, it at the same time blurs the language's scope with respect to expressivity.

Initially, FRP development was driven by the idea to utilize time as an adaptable parameter for the creation of reactive animations [36]. The underlying goal was to be indefinitely scalable, similar to space being used as an adaptable parameter in scalable vector graphics. Later on, the goals of FRP have been extended to replacing complex event based systems with a more modular design [13]. The necessity for such systems especially rose for graphical user interfaces [33], as well as for more advanced computation management related back-ends [32].

Today, FRP is used in many application areas such as embedded devices [61], interactive games [111], robotics [75], GUIs [33], hardware circuits [15], and interactive multimedia [126]. In comparison to classical programs, FRP programs can be exceptionally more efficient. For example, an FRP implementation that has been created to run on a network controller was able to outperformed all its contemporary competing implementations [144].

## 7.2 Time as a Type

The most fundamental concept of FRP is the idea of encapsulating time into a type. The core abstraction of FRP is that of a signal

$$\mathit{Signal}\ \alpha\ ::\ \mathit{Time}\ \rightarrow\ \alpha$$

which determines the value of some arbitrary (polymorph) type $\alpha$ at any point in time. In general, the concept of a signal in FRP is more powerful than the concept of infinite words and, therefore, generalizes the notions for classical reactive systems. On the one hand, signals cover arbitrary (possibly infinite) data domains. On the other hand, the time domain is well defined, even in the case of continuous time. However, in order to rely on clean alignment with TSL, we restrict ourselves to $\mathit{Time} \mathrel{\hat{=}} \mathbb{N}$ at this point. Similar to the classical setting, values of type $\alpha$ can be arbitrary inputs from the world, such as the current position of a mouse, as well as arbitrary outputs to the world, such as text that is rendered to the screen.

The transformation of signals is expressed through signal functions. For example for rendering the position of the mouse on a screen. There are two different variants of how of a signal function can be typed:

1. $\mathit{Signal}\ \alpha\ \rightarrow\ \mathit{Signal}\ \beta$
2. $\mathit{Signal}\ (\alpha\ \rightarrow\ \beta)$

The first variant conceptually receives all inputs at every point in time before it determines the corresponding outputs. The second variant, on the other hand, provides a potentially different transformation at any point in time. Mathematically, both variants are equivalent, but depending on the FRP instantiation they are not both supported natively or only one of the variants supports specific operations with respect to the internal realization.

A standard approach for representing signals as part of a programming language is using infinite lists. Therefore, the approach requires a lazy evaluation environment, which is not supported by all functional programming languages. This is the reason, why most FRP libraries have their origin in the functional programming language *Haskell*, which is built on the concept of laziness from ground up [66]. Signal functions then are implemented through the step by step processing of the infinite lists. This way, a conceptual view of working with infinite data streams is provided from the programmers perspective, which still can be executed in reactive environments due to the implicit time model being built-in.

## 7.3 Design Patterns

Even with a fixed programming language environment, like Haskell, at hand the FRP paradigm still can be realized in many different ways, each with well-defined levels of expressive power. In the functional programming world, these levels usually are realized through so called design patterns, like *Applicative* [103], *Monads* [145], or *Arrows* [67], that support standardized ways of how data and control flow is structured and executed within the programming language. Depending of which design pattern is used, there are different advantages, but also restrictions of how the stream processing network is built and executed eventually. We shortly introduce the most common patterns that are used by Applicative, Monadic, and Arrowized FRP.

### 7.3.1 Applicative FRP

In Haskell, the Applicative class is defined as follows:

```
class Applicative f where
  pure  :: α → f α
  (⊛)   :: f (α → β) → f α → f β
```

The class characterizes type instances that are build on top of two basic operations. On the one hand, the `pure` operation takes an arbitrary value and puts it into the context of the Applicative type. On the other hand, the compose operation ⊛ supports the application of a function being encapsulated within the Applicative type to a value within the same context. Therefore, every Applicative instance must satisfy the following laws:

**identity:** `pure id ⊛ v ≡ v`

**composition:** `pure (∘) ⊛ u ⊛ v ⊛ w ≡ u ⊛ (v ⊛ w)`

**homomorphism:** `pure f ⊛ pure x ≡ pure (f x)`

**interchange:** `u ⊛ pure y ≡ pure (λf → f y) ⊛ u`

where `id` denotes the identity function and ∘ denotes the function concatenation operation. In the context of FRP, the signal type can be considered to be an instance of the applicative class:

```
instance Applicative Signal where
  pure  :: α → Signal α
  (⊛)   :: Signal (α → β) → Signal α → Signal β
```

With respect to the concept of time, the `pure` operation then allows to lift a value of type $\alpha$ to the temporal domain leading to a constant stream of

that value for all points in time. The compose operation $\circledast$ allows to convert a stream of point-wise potentially different function transformations into a uniform stream transformer turning an input stream into a corresponding output stream.

The Applicative instance provides the closest incarnation of FRP with respect to the underlying concept of time. However, it is only rarely used in practice. The reason is that there is no straightforward way to directly embed the time concept into a host language like Haskell. Note that just because we consider infinite lists to represent data streams of temporally changing values, there is still no information available for the compiler of how to translate this concept into a corresponding executable in the end. Therefore, this information must be embedded into the corresponding language compiler as well, which is non-trivial regarding the history of the Haskell compiler development. Furthermore, the time model may also be application specific, which is why a new compiler must be generated for each application domain.

Nevertheless, the challenge has been accepted with the functional hardware description language `ClaSH` [7]. The language allows to use a purely Applicative FRP instance to describe synchronous hardware circuits. Due to the circuits being synchronous, there is a well-defined source of time given by the hardware clock. As it turns out, `ClaSH` indeed comes with a dedicated language compiler that builds on the syntactic front-end of Haskell, but translates to low level hardware descriptions in the core. Therefore, to the best of our knowledge, `ClaSH` is the only instance of FRP today that solely builds on a purely Applicative framework yet.

### 7.3.2 Monadic FRP

Just because the Applicative framework on its own is to weak for integrating FRP into Haskell, it does not mean that there is no solution. Remember that the primary problem with purely Applicative FRP is that there is a missing link of how the temporal behavior is executed as part of an application specific executable in the end. This problem can be solved by using a Monad [145], which introduces the missing evaluation context. In Haskell, the Monad class is defined as follows:

```
class Monad m where
  return :: α → m α
  (>>=)  :: m α → (α → m β) → m β
```

The class provides an operation `return` that puts an object into the monadic context, similar to the `pure` method for Applicative. Furthermore, it also supports a sequential composition operator `>>=` that allows to take a value

out of the context in order to apply a monadic transformation putting the value again back into the context. Every Monad instance must satisfy the following laws:

**left identity:** `return a >>= f ≡ f a`
**right identity:** `m >>= return ≡ m`
**associativity:** `(m >>= f) >>= g ≡ m >>= (λx → f x >>= g)`

While the Monad class can be seen as an extension to the Applicative class, in particular every Monad instance is also an Applicative, it is not used for adding more functionality to the signal type in the context of FRP, but instead to provide a separate evaluation context. Hence, with respect to Monadic FRP the signal type still is an Applicative, exactly as introduced in the previous section, but every signal function is executed in a monadic context `m`:

$$\texttt{SF}\ \alpha\ \beta :: \texttt{Signal}\ \alpha \rightarrow\ \texttt{m}\ (\texttt{Signal}\ \beta)$$

Thus the Monad is used to provide the evaluation context that was missing for purely Applicative FRP. Monadic FRP is one of the most popular implementation frameworks and used by many FRP libraries, such as `FRPNow` [142], `Elerea` [109], `Reactive-Banana` [4], `Threepenny-GUI` [5], or `Reflex` [141].

Conceptually, FRP allows to introduce circular dependencies, as long as every circular path of the network is intercepted by at least one delaying component, similar to the usage of a cell in TSL. To this end, in Monadic FRP such circular dependencies also require the monadic context to be an instance of the MonadFix [39] type class:

```
class MonadFix m where
  mfix :: (α → m α) → m α
```

The class comes in combination with the following additional laws:

**purity:** `mfix (return ∘ h) ≡ return (fix h)`
**left shrinking:** `mfix (λx → a >>= λy → f x y) ≡ a >>= λy → mfix (λx → f x y)`
**sliding:** `mfix (liftM h ∘ f) ≡ liftM h (mfix (f ∘ h))`
**nesting:** `mfix (λx → mfix (λy → f x y)) ≡ mfix (λx → f x x)`

where `fix f = let x = f x in x` denotes the least fixpoint of `f` and `liftM f x = x >` is a utility function that promotes a function to the Monad.

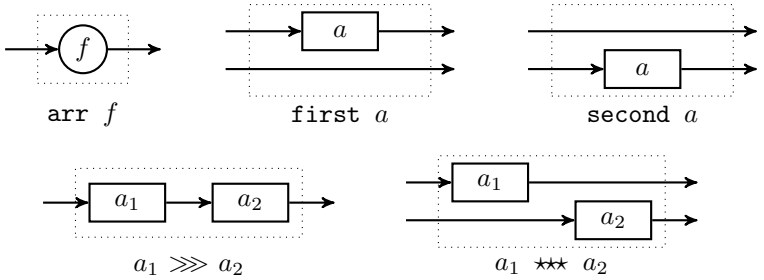



Figure 24: Graphical representation of the core Arrow operations.

### 7.3.3 Arrowized FRP

Although Monadic FRP is expressive enough to embed FRP frameworks as part of the Haskell language it still comes with the disadvantage of enforcing a strict sequential evaluation through the composition operator `>>=`. Regarding the conceptual idea of FRP to create a stream processing network, this is a strong limitation. Therefore, another approach has established building on the Arrow class [67], which offers a builtin evaluation model for the parallel execution of the components. In Haskell, the Arrow class is defined as follows:

```
class Arrow a where
  arr   :: (α → β) → a α β
  (⋙)   :: a α β → a β γ → a α γ
  first :: a α β → a (α, γ) (β, γ)
```

The class offers three core operations. The `arr` operation puts pure functions into the arrow context. Therefore, the operation supports a similar concept as the `pure` and `return` operations in Applicative and Monadic FRP, respectively. Nevertheless, `arr` is more expressible in general, since the direction of the computation from $\alpha$ to $\beta$ is maintained as well. Similarly, arrows can be composed sequentially using the composition operator $\ggg$. However, they can also be put into a parallel execution context using `first`. Therefore, note that the operation does not add any concrete parallel execution yet, but only adds the corresponding context. Using the core operations above, they usually are extended by `second`, the dual operation of `first`, as well as by the parallel composition operator ***:

```
second :: a α β → a (γ, α) (γ, β)
second f = arr swap ⋙ first f ⋙ arr swap
  where swap (a,b) = (b,a)
```

```
(***) :: a α β → a α′ β′ → a (α, α′) (β, β′)
f *** g = first f >>> second g
```

A graphical representation of the corresponding operations is given in Figure 24. Additionally, every Arrow instance must satisfy the following laws:

**left identity:** `arr id >>> f ≡ f`
**right identity:** `f >>> arr id ≡ f`
**associativity:** `(f >>> g) >>> h ≡ f >>> (g >>> h)`
**composition:** `arr (g ∘ f) ≡ arr f >>> arr g`
**extension:** `first (arr f) ≡ arr (f × id)`
**functor:** `first (f >>> g) ≡ first f >>> first g`
**exchange:** `first f >>> arr (id × g) ≡ arr (id × g) >>> first f`
**unit:** `first f >>> arr fst ≡ arr fst >>> f`
**association:** `first (first f) >>> arr assoc ≡ arr assoc >>> first f`

where × denotes the binary tuple constructor, `fst` the projection to the first tuple component, and `assoc` is defined as `assoc ((a,b),c) = (a,(b,c))`.

In arrowized FRP, a signal is processed as part of a signal function `SF α β`, which represents the corresponding instance of the Arrow. Arrowized FRP was initially introduced to plug a space leak in the original work of FRP [36, 97]. Also consider that the abstractions used by the different implementations of FRP vary in their expressive power. Therefore, as it turns out, arrowized FRP has a smaller interface than a monadic FRP [95], which however restricts the particular constructs that caused the aforementioned space leak.

Arrowized FRP is for example used by the libraries `Yampa` [65], `UISF` [147], `Rhine` [9], `Dunai` [112], or `Midair` [108]. Moreover, the libraries `Dunai` and `Rhine` use even a further extension of arrowized FRP incorporating monadic properties into the framework as well. The extension is a called monadic stream functions (MSF) and originally was introduced in [112].

Similar to the MonadFix class for monadic FRP, arrowized FRP also requires an additional class in order to express circular dependencies within the network. In arrowized FRP this extension is given by ArrowLoop [110]:

```
class ArrowLoop a where
  loop :: a (α, γ) (β, γ) → a α β
```

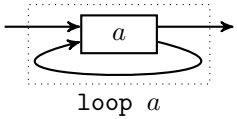



Figure 25: Graphical representation of the Arrow `loop` operation.

The `loop` operation allows to loop an output of type $\gamma$ back to the component. A graphical representation of the operation is given in Figure 25. The ArrowLoop class requires the following additional laws to be satisfied:

**left tightening:** $\texttt{loop}\ (\texttt{first h} \ggg \texttt{f}) \equiv \texttt{h} \ggg \texttt{loop f}$

**right tightening:** $\texttt{loop}\ (\texttt{f} \ggg \texttt{first h}) \equiv \texttt{loop f} \ggg \texttt{h}$

**sliding:** $\texttt{loop}\ (\texttt{f} \ggg \texttt{arr}\ (\texttt{id} \times \texttt{k})) \equiv \texttt{loop}\ (\texttt{arr}\ (\texttt{id} \times \texttt{x}) \ggg \texttt{f})$

**vanishing:** $\texttt{loop}\ (\texttt{loop f}) \equiv \texttt{loop}\ (\texttt{arr assoc}^{-1} \ggg \texttt{f} \ggg \texttt{arr assoc})$

**superposing:** $\texttt{second}\ (\texttt{loop f}) \equiv \texttt{loop}\ (\texttt{arr assoc} \ggg \texttt{second f} \ggg \texttt{arr assoc}^{-1})$

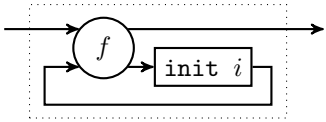
**extension:** $\texttt{loop}\ (\texttt{arr f}) \equiv \texttt{arr}\ (\texttt{trace f})$

where `trace f b = let (c, d) = f (b, d) in c`.

### 7.3.4 Causal Commutative Arrows

Arrowized FRP explicitly supports parallel composition and therefore allows for a more efficient evaluation of stream processing networks than Applicative or Monadic FRP. On the contrary, arrowized FRP is less expressive than monadic FRP due to the strong coupling of inputs with outputs as part of the signal function type. Moreover, it turns out that even the original introduction of the Arrow type class still misses some important properties, when it comes to the schedulebility of the individual components' execution. An important property that has not been targeted so far considers the commutativity of executing components that are composed using the parallel composition operator $\star\!\!\star\!\!\star$.

The corresponding nuisance is introduced by the ability of Arrow instances to be defined such that they are able to internally carry state, which then is updated differently depending on how the corresponding network components are scheduled. Note that the usage of internal state in general does not have




loopD $i$ $f$

Figure 26: The special `loopD` operation of CCA that is initialized with the user provided value $i$.

to be malicious at first. Consider for example an arrow equipped with a global counter that keeps track of the amount of processed data at run time. Due to addition being a commutative operation as well, this arrow respects the commutativity law. However, non-commutative state is also possible. For example, in the arrowized FPR library `UISF` [147] arrows are used to position graphical user interface elements. According to the library design, the order of new elements strictly depends on the previously laid out ones, as given by the underlying arrow structure.

The problem can be avoided through an extension of the arrow class called Causal Commutative Arrows (CCA) [151, 96]. CCA introduces additional laws, which explicitly enforce the parallel operation to be commutative, as well as the well-defined initialization of state as part of looping components. Therefore, CCA also comes with a special initialization operator `init`, which is introduced as part of a new ArrowInit class.

```
class ArrowInit a where
  init :: α → a α α
```

In addition to implementing the `init` operator, every CCA instance then must satisfy the corresponding commutativity and initialization laws:

**commutativity:** `first f ⋙ second g ≡ second g ⋙ first f`
**product:** `init i *** init j ≡ init (i, j)`

Another advantage of the `init` operator is that is allows the introduction of `loopD`, which is a loop that includes initialization as shown in Figure 26.

```
loopD :: γ → ((α, γ) → (β, γ)) → a b c
loopD i f = loop (f ⋙ second (init i))
```

Due to CCA again being restricted in its interface, there are more libraries that can simulate CCA than Arrowized FRP. Furthermore, CCA also supports a normalization procedure with respect to the network structure, which offers more optimization opportunities than Arrowized FRP in general.

## 7.4 Code Generation

Based on the aforementioned insights, we created a framework for FRP program generation from synthesized CFMs. It starts from the CFM, as it is synthesized from the orginal TSL specification utilizing a finite set of predicate and function literals. The user then selects the target FRP abstraction (Applicative, Monads, or Arrows) and receives an executable FRP program in the end. Therefore, the process is clearly separated from the design of the initial CFM due to the postponed selection of the application specific FRP framework and the corresponding function and predicate implementations.

With a CFM that satisfies the original TSL specification at hand, we first compile it into a universal template for the later FRP program. The generated code is organized as follows:

```
control
  :: _ signal   -- FRP abstraction
  ⇒ _           -- cell implementation
  → (_ → _)     -- functions and predicates
  → _           -- initial values
  → signal _    -- input signals
  → signal _    -- output signals

control _ ··· _ =
  rec
    ∀c ∈ C.  c ← δ_M(c)
    ∀v ∈ V.  v ← ℓ_M(v) δ_M(v)
    ∀o ∈ O.  o ← δ_M(o)
  return
    ∀o ∈ O. o
```

First, the stream processor is specialized towards the desired FRP framework using the required class constraints of the corresponding design pattern and the cell implementation of the targeted FRP library. Next, the function and predicate implementations are provided, as well as the initial values of all utilized cells. The result is a concrete stream processor implementation that receives input streams and produces the corresponding output streams over time. Examples of the concertized interfaces for the Applicative FRP framework `ClaSH`, Monadic and Arrowized FRP are given in Figure 27.

Reconsider that the model of a CFM only requires the expressivity of CCA to be integrated within FRP. Correspondingly, we are able to generate code for any FRP library that is at least as powerful as CCA [142, 112, 108, 109, 51].

```
control
  :: (HiddenClockReset d g s)
                ⋮
  → Signal d β_{i_0} → ··· → Signal d β_{i_{n-1}}
  → (Signal d γ_{o_0}, ..., Signal d γ_{o_{m-1}})
```

(a) The Applicative FRP control interface (specialized for `ClaSH`).

```
control
  :: (Monad m, MonadFix m, Applicative s)
  ⇒ (forall α. α → s α → m (s α))
                ⋮
  → s β_{i_0} → ··· → s β_{i_{n-1}}
  → m (s γ_{o_0}, ..., s γ_{o_{m-1}})
```

(b) The Monadic FRP control interface.

```
control
  :: (Arrow sf, ArrowLoop sf, ArrowInit sf)
  ⇒ (forall α. α → sf α α))
                ⋮
  → sf (β_{i_0}, ..., β_{i_{n-1}}) (γ_{o_0}, ..., γ_{o_{m-1}})
```

(c) The Arrowized FRP control interface utilizing CCA.

Figure 27: The type signatures of the created control interfaces for each of the aforementioned design pattern: Applicative, Monads, and CCA.

```
-- Yampa
iPre :: SF α α

-- ClaSH
register
  :: HiddenClockReset d g s
  ⇒ α → Signal d α → Signal d α

-- Threepenny-GUI
cell
  :: MonadIO m
  ⇒ α → Behavior α → m (Behavior α)

cell v x = stepper v (x <@ allEvents)
```

Figure 28: Cell implementations of the utilized FRP libraries.

In other words, first-order control and the ability to express circular dependencies already are sufficient to capture the expressive power of both: the CFM and CCA. Although many FRP libraries support more powerful operations than CCA, *e.g.*, `switch`, which is used for the dynamic reconfiguration of the stream processing network at run time, we do not particularly rely on them for synthesis. This is especially of interest, since the unpredictable behavior of dynamically evolving networks naturally limits the availability of statically computable run-time and memory consumption guarantees. Moreover, the use of dynamically calculated networks often is largely impractical for many FRP applications due to their additional resource overhead. Such overhead must be for example avoided on embedded devices [127] or in applications that are implemented in hardware [7]. Prior work on CCA also showed that the expressive power of higher-order arrows makes the support for automatic optimization more difficult. Furthermore, for most FRP programs a static networks structure is more than enough [148].

We close the section with a review of the Kitchen Timer application from the introduction to give a feeling of the concrete code that is generated by our approach. We first generate a CFM utilizing six additionally synthesized cells and 1188 vertices from the presented TSL specification using our framework and the synthesis tool `strix` [105]. This CFM then is translated into the corresponding control structures for three tested application domains. We create

Table 1: Synthesis and compilation times for creating the timer applications.

| Executed Tool | Time (sec) |
|---|---|
| Synthesis → `strix` | 4.965 |
| Compilation | |
| Desktop → `Yampa` | 19.403 |
| Web → `Threepenny-GUI` | 18.344 |
| Hardware | |
| → `ClaSH` | 11.218 |
| → `yosys` | 6.405 |
| → `nextpnr` | 7.276 |

a desktop program that is built on top of the FRP library `Yampa` and a web application using `Threepenny-GUI`. Furthermore, we also implement the timer in hardware using the functional hardware description language `ClaSH`. Therefore, the dedicated `ClaSH` compiler generates verilog code, which then is translated into the `blif` format using the open synthesis suite `yosys` [132]. Next, the generated `blif`-file is placed using the place-and-route tool `nextpnr` [132]. Afterwards, the resulting package is uploaded to an iCEblink40HX1K Evaluation Kit Board from Lattice Semiconductor, featuring an ICE40HX1K FPGA with 100 IO-pins and 1280 logic cells, which additionally is equipped with all the required hardware components. The interfaces of the corresponding timer applications are depicted in Figure 6 from Chapter I. The respective synthesis and compilation times of the different tools are depicted in Table 1.

Finally remember that each FRP instantiation requires a library specific cell implementation to be passed to the generated `control`. The FRP libraries `Yampa` and `ClaSH` provide these natively, as shown in Figure 28. `Threepenny-GUI`, on the other hand, does not provide a native implementation on its own. However, the missing piece can be easily implemented with the primitives `stepper` and (`<@`), which are provided by the library instead.

# 8 Experimental Results

We evaluate the synthesis approach using a created tool set, called `tsltools`. Using this framework, given TSL specifications are first approximated to LTL and then refined until the utilized LTL solver either produces a realizability result or returns a non-spurious counter-strategy. Therefore, we utilize the bounded synthesis tool BoSy [41] for LTL synthesis. As soon as the refinement terminates with a realizing strategy it is translated to a CFM, which then is

used to generate the corresponding FRP program.

Our benchmark set comprises of various application domains, where every benchmark class focuses on a different feature of TSL. To this end, all of the listed specifications have been created from scratch with the goal of either targeting existing textual specifications or other real world scenarios. For every benchmark, we consider its size $|\varphi|$, the number of input literals $|\mathtt{I}|$, the number of output literals $|\mathtt{O}|$, the number of predicate literals $|\mathtt{P}|$, and the number of function literals $|\mathtt{F}|$ (including the literals of $\mathtt{P}$). Regarding the synthesis process, we then measure the synthesis times in seconds, the number of cells $|\mathtt{C}_{\mathcal{M}}|$ utilized by the generated CFM $\mathcal{M}$, as well as the number of vertices $|V_{\mathcal{M}}|$. The corresponding results are listed in Tables 2 and 3. All results of Table 2 did not require any refinement, *i.e.*, the initial approximation already was sufficient, whereas the benchmarks of Table 3 required the listed number of refinements $n$. The synthesis was executed on a quad-core Intel Xeon processor (E3-1271 v3, 3.6GHz, 32 GB RAM, PC1600, ECC), running Ubuntu 64bit LTS 16.04.

The button benchmark represents a simple GUI application that requires a button to be pressed in order to increase a counter. The music player benchmarks cover the specification of Section 2.2, as well as some preliminary variants of reduced complexity. The FRPZoo benchmark set refers to a standard online benchmark suite, designed to compare FRP library language designs [51]. Therefore, the online available textual specification separates between three different behaviors, given as scenarios 0, 5, and 10. In every scenario, two buttons can be clicked: a `clickCount` button, which counts the number of clicks, and a `toggle` button, which toggles the enable/disable state of the `clickCount` button. The value of the counter is displayed via an output interface. The three scenarios differ with respect to the exact conditions of when the counter is updated, reset or displayed. The escalator benchmarks cover different TSL translations of the LTL escalator example from the preliminaries. The slider benchmarks specify different variants of a small graphical game, where a slider moves back and forth and a player has to push a button whenever the slider is at the center to score points.

The TORCS specifications build upon examples of the Haskell-TORCS bindings for building FRP controllers [44] in The Open Race Car Simulator (TORCS) [150]. The benchmarks describe controllers for autonomous vehicles, which in TORCS have access to limited sensor data about the environment (*e.g.* the distance to the nearest obstacles) and to actuators in the car (*e.g.* the steering wheel). The TSL specifications describe different controllers, for which the sensors and actuators act as input and output signals, respectively. Utilized function literals are for example `slowDown` or

Table 2: Number of cells $|\mathtt{C}_{\mathcal{M}}|$ and vertices $|V_{\mathcal{M}}|$ of the resulting CFM $\mathcal{M}$ and synthesis times for a collection of TSL specifications $\varphi$. A * indicates that the benchmark additionally has an initial condition as part of the specification.

| Benchmark ($\varphi$) | $\lvert\varphi\rvert$ | $\lvert\mathtt{I}\rvert$ | $\lvert\mathtt{O}\rvert$ | $\lvert\mathtt{P}\rvert$ | $\lvert\mathtt{F}\rvert$ | $\lvert\mathtt{C}_{\mathcal{M}}\rvert$ | $\lvert V_{\mathcal{M}}\rvert$ | Time (s) |
|---|---|---|---|---|---|---|---|---|
| **Button** | | | | | | | | |
| default | 7 | 1 | 2 | 1 | 3 | 3 | 8 | 0.364 |
| **Music App** | | | | | | | | |
| simple | 91 | 3 | 1 | 4 | 7 | 2 | 25 | 0.77 |
| system feedback | 103 | 3 | 1 | 5 | 8 | 2 | 31 | 0.572 |
| motivating example | 87 | 3 | 1 | 5 | 8 | 2 | 70 | 1.783 |
| **FRPZoo** | | | | | | | | |
| $\text{scenario}_0$ | 54 | 1 | 3 | 2 | 8 | 4 | 36 | 1.876 |
| $\text{scenario}_5$ | 50 | 1 | 3 | 2 | 7 | 4 | 32 | 1.196 |
| $\text{scenario}_{10}$ | 48 | 1 | 3 | 2 | 7 | 4 | 32 | 1.161 |
| **Escalator** | | | | | | | | |
| non-reactive | 8 | 0 | 1 | 0 | 1 | 2 | 4 | 0.370 |
| non-counting | 15 | 2 | 1 | 2 | 4 | 2 | 19 | 0.304 |
| counting | 34 | 2 | 2 | 3 | 7 | 3 | 23 | 0.527 |
| counting* | 43 | 2 | 2 | 3 | 8 | 4 | 43 | 0.621 |
| bidirectional | 111 | 2 | 2 | 5 | 10 | 3 | 214 | 4.555 |
| bidirectional* | 124 | 2 | 2 | 5 | 11 | 4 | 287 | 16.213 |
| smart | 45 | 2 | 1 | 2 | 4 | 4 | 159 | 24.016 |
| **Slider** | | | | | | | | |
| default | 50 | 1 | 1 | 2 | 4 | 2 | 15 | 0.664 |
| scored | 67 | 1 | 3 | 4 | 8 | 4 | 62 | 3.965 |
| delayed | 71 | 1 | 3 | 4 | 8 | 5 | 159 | 7.194 |
| **Haskell-TORCS** | | | | | | | | |
| simple | 40 | 5 | 3 | 2 | 16 | 4 | 37 | 0.680 |
| **advanced** | | | | | | | | |
| gearing | 23 | 4 | 1 | 1 | 3 | 2 | 7 | 0.403 |
| accelerating | 15 | 2 | 2 | 2 | 6 | 3 | 11 | 0.391 |
| **steering** | | | | | | | | |
| simple | 45 | 2 | 1 | 4 | 6 | 2 | 31 | 0.459 |
| improved | 100 | 2 | 2 | 4 | 10 | 3 | 26 | 1.347 |
| smart | 76 | 3 | 2 | 4 | 8 | 5 | 227 | 3.375 |

Table 3: Set of programs that use purity to keep one or two counters in range.

| Benchmark ($\varphi$) | $\lvert\varphi\rvert$ | $\lvert\mathtt{I}\rvert$ | $\lvert\mathtt{O}\rvert$ | $\lvert\mathtt{P}\rvert$ | $\lvert\mathtt{F}\rvert$ | $\lvert\mathtt{C}_{\mathcal{M}}\rvert$ | $\lvert V_{\mathcal{M}}\rvert$ | $n$ | Synthesis Time (s) |
|---|---|---|---|---|---|---|---|---|---|
| inrange-single | 23 | 2 | 1 | 2 | 4 | 2 | 21 | 3 | 0.690 |
| inrange-two | 51 | 3 | 3 | 4 | 7 | 4 | 440 | 6 | 173.132 |
| graphical-single | 55 | 2 | 3 | 2 | 6 | 4 | 343 | 9 | 1767.948 |
| graphical-two | 113 | 3 | 5 | 4 | 9 | - | - | - | $>10000$ |

`turnLeft` providing an intuitive interface for describing high level control. In this way, guarantees of the overall system behavior can be specified, while at the same time numerically sensitive, data specific manipulations are still allowed. The first *simple* controller combines behavior, which does not require state, whereas the *advanced* controllers include a more detailed planning procedure for approaching a turn. The advanced versions are also kept modular, in the sense that the control of the steering wheel and the control of the gears are given by separate specifications, which then are combined to a single FRP program after synthesis again.

The benchmarks of Table 3 are inspired by examples of the Reactive Banana FRP library [4]. For the realizability of these benchmarks purity of function and predicate applications must be utilized to ensure that the value of one or two counters never goes out of range. In this context, the system not only needs purity to be able to verify the condition, but also to take the correct decisions in the resulting implementation to be synthesized.

In summary, our results show that TSL indeed successfully lifts the applicability of synthesis from the Boolean domain to arbitrary data domains, allowing for new applications that can utilize all of the required levels of abstraction. For all of the benchmarks, we could find a realizable system within a reasonable amount of time. The results often required synthesized cells to realize the resulting control flow behavior.

# 9 Discussion

We introduced Temporal Stream Logic, a logic that leverages a clean separation between data and control. To this end, the logic focuses on describing the control behavior of reactive systems, while keeping the data specific considerations abstract. More precisely, the concrete data representations and the implementation of respective data transformations is hidden from the system designer using a universal abstraction over the space of possible data instan-

tiations. The logic has been especially designed for the synthesis of reactive systems with the focus on creating a correct behavior design at first, since we assume that this design step precedes all other development steps in general. Accordingly, in an initial development stage, the major challenge lies in finding a suitable behavior of the control, while technical details of how data is transformed can be postponed to later stage instead.

TSL clearly is more expressive than classical temporal logics, like LTL, due to the universal abstraction of the data and the corresponding data transformations. However, as a trade-off, this abstraction also renders the logic to be undecidable in general. Therefore, we showed that undecidability even stays alive, if being restricted to a singe binary function transformation and a single unary predicate that can be checked.

In another direction, we analyzed the game models that underlie TSL. We found that the standard notion of determinacy differs from the classical game settings, but could not answer the question, whether TSL games are determined or not. Furthermore, we discovered that TSL games do not always obey a finite game arena such that we introduced the distinction between creational and non-creational TSL. Finally, we revealed that even for finite arenas, the players may need infinite memory in order to win, whereas these results already hold for games with reachability or safety winning conditions. Another open question that we could not answer asks, whether TSL games with weaker winning conditions are decidable or not.

As a consequence, we went back to the logic itself and instead introduced an approximate reduction to classical LTL in combination with a CEGAR based refinement approach that iteratively improves the approximation until a solution is found. Therefore, the approximation is sound but not complete. However, our experimental results show that it is indeed sufficient for successfully synthesizing TSL specifications of real world systems in practice.

Finally, we considered the connection between synthesized CFMs and FRP. Although there are many incarnations of FRP, build on top of different design patterns, we showed the CFM model to be fundamental enough, to support translations to all of them.

In conclusion, we observe that TSL offers a different perspective compared to other temporal logics, when it comes to the exploration of control behavior of reactive systems design. However, at the same time it also comes with new and open challenges, whose solution will help us to better understand the peculiarities of reactive system design, even under the scope of keeping the processed data abstract.

# Chapter IV

# Output Sensitive Synthesis

Reactive synthesis from TSL specifications leverages a clean separation of data and control. It enables the specification of even complex real-world applications and opens new design methodologies for system developers. Nevertheless, the resulting development process also introduces some new challenges. A system designer now has to create specifications instead of implementations, where specifications describe solution spaces in contrast to deterministic implementations. Accordingly, there may be many implementations that satisfy the designer's intends. Hence, choosing the solution space correctly becomes of critical importance for the system design. As a consequence, system developers also need tools and methods, which verify the created specifications against the original design indents.

A first method for supporting developers with intention validations, in case of realizable specifications, is given by the inspection of the created implementations. System designers not being satisfied with the synthesis results immediately indicate missing behavior properties or erroneous formulations within the specification. An inspection, however, assumes that the designers always are able to sufficiently understand the synthesized implementations. Or in other words, it is assumed that synthesis results always are easily human readable. Unfortunately, with respect to most of the currently available synthesis tools, this assumption gets hard to defend. The problem is that classical synthesis approaches only require solutions that satisfy the specification, *i.e.*, are functionally correct, but ignore additional quality metrics. The automata transformations and infinite games that are utilized by these approaches never have been designed to maintain any notion of quality. These traditional solutions are designed to run in optimal time, with respect to the corresponding complexity classes, but ignore the complexity of the synthesized results.

Nevertheless, synthesis approaches that produce comprehensible implementations are unavoidable for a synthesis based development. Especially, since missing behavior properties cannot be verified. Note that without an additional inspection, developers are not even aware of their existence. Otherwise, they would have added them to the system specification in the first place.

In this chapter we consider synthesis approaches producing implementations that are not only correct with respect to the specification, but also must be human comprehensible. To this end, we analyze output sensitive synthesis approaches that introduce additional quality metrics to the created implementations. Through the control of these metrics, developers then can impose additional non-functional requirements on the synthesized results. These additional constraints not only help with improving the readability, but also are able to ensure additional system requirements that may be important for subsequent system integration as well. In this sense, the kind of applicable quality metric also depends on the representation of the system to be synthesized. Therefore, we also consider the impact of choosing different implementation models together with matching quality metrics, which we evaluate with respect to the effect on the overall synthesis complexity. We explicitly only consider synthesis approaches for LTL, since all results always also are applicable to TSL, due to the soundness of the approximation in case of realizable specifications, as discussed in Chapter III.

# 1 Bounded Synthesis

Output sensitive synthesis adds quality parameters to the synthesis process that are provided as inputs by the developers in addition to the behavior specification. The first output sensitive synthesis method was introduced by bounded synthesis [45], which imposes an additional bound $n \in \mathbb{N}$ on the state size of the created Mealy machines. Bounded synthesis searches only for Mealy machines that satisfy the specification and are no larger than the given bound. The approach not only reduces the size of the produced implementations, but also the synthesis times in case of practical applications [42, 43, 69].

## 1.1 Constraint based Synthesis

In order to introduce a bound $n$ into the synthesis process, the search for a satisfying solution is reduced to constraint system such that the synthesis process naturally splits into two phases. In the first phase, all non-deterministic choices are removed from the specification, which especially resolves decisions that have been introduced locally, under the scope of a specified property, but require to be postponed into to the future with respect to a system implementation. Regarding the standard transformations for automata on infinite words, the classic approach that implements such a reduction first translates the negated LTL property into an alternating Büchi word automaton, which then is reduced to a language-equivalent non-deterministic Büchi word au-

tomaton afterwards. With respect to the negation, as introduced initially, this automaton then is equivalent to a universal co-Büchi automaton that represents the same language as the initial LTL property. Note that the universal co-Büchi automaton is free of any non-deterministic choices.

The first phase, thus, reduces the specification to a set of local requirements that are still distributed over time, but are independent from each other according to this global distribution. Accordingly, it only remains to resolve these constraints with respect to a uniform system implementation. Therefore, the elimination of non-determinism in the first phase guarantees that all remaining constrains can be verified locally. Thus, the only remaining challenge is to combine them within a single system implementation. On the one hand, the problem, thus, is reducible to a much simpler constraint system that not especially requires to be aware of the notion of time. On the other hand, the encoded constraints can be easily extended with further requirements, like a bound on solution size as utilized by bounded synthesis. The most classical example for a constraint system is *Boolean Satisfiability* (SAT), where the constraints are encoded as part of a Boolean formula. Beside Boolean Satisfiability, there are, however, also more advanced solutions, such as *Quantified Boolean Formulas* (QBF), *Dependency Quantified Boolean Formulas* (DQBF), or *Boolean Satisfiability Modulo Theories* (SMT).

Formally, the bounded synthesis approach first translates a given LTL specification $\varphi$ to an equivalent universal co-Büchi automaton $\mathfrak{A}$, such that $\mathcal{L}(\mathfrak{A}_\varphi) = \mathcal{L}(\varphi)$. The problem, thus, reduces to finding implementations $\mathbb{M}$ that are accepted by $\mathfrak{A}_\varphi$. More precisely, we search for an implementation $\mathbb{M}$, for which the run graph $G_{\mathfrak{A}_\varphi,\mathbb{M}}$ of $\mathbb{M}$ and $\mathfrak{A}_\varphi$ contains no cycle with a rejecting vertex. This property can be witnessed by a ranking $\lambda$, which annotates each vertex of $G_{\mathfrak{A}_\varphi,\mathbb{M}}$ with a natural number that bounds the number of possible visits to rejecting states. The ranking itself is bounded by $n \cdot k$, where $n$ is the provided bound restricting the size of $\mathbb{M}$ and $k$ is the number or rejecting states of $\mathfrak{A}_\varphi$. The search for $\mathbb{M}$ then is reduced to constraint system that guesses the Mealy machine $\mathbb{M}$ itself, the corresponding run graph $G_{\mathfrak{A}_\varphi,\mathbb{M}}$ as a cross-product of $\mathbb{M}$ and the universal co-Büchi specification automaton $\mathfrak{A}_\varphi$, and a validating ranking $\lambda$ that proves the correctness of $\mathbb{M}$ with respect to $\mathfrak{A}_\varphi$. Thus, if all constraints are satisfiable, then it is proven that the encoded Mealy machine $\mathbb{M}$ indeed satisfies the specification $\varphi$.

In the scope of this thesis we use SAT as our primary constraint system of choice, since we consider it as a basis of the available constraint systems. There are, however, also other lines of work that especially focus on the implications of choosing more advanced systems for the bounded synthesis encodings, such as QBF, DQBF, or SMT [40, 41].

## 1.2 SAT Encoding

Let a universal co-Büchi automaton $\mathfrak{A}_\varphi = (2^{\mathcal{I}\cup\mathcal{O}}, Q, q_I, \Delta_\mathfrak{A}, \textsc{coBüchi}(R))$ with $k = |R|$ and a bound $n \in \mathbb{N}^+$ be given. We introduce Boolean variables to guess a satisfying Mealy machine $\mathbb{M} = (2^\mathcal{I}, 2^\mathcal{O}, M, m_I, \delta_\mathbb{M}, \ell)$ and a valid ranking $\lambda\colon M \times Q \to [n \cdot k] \cup \{\_\}$.

- $\textsc{trans}(m, \nu, j)$ for all $m \in M$, $\nu \in 2^\mathcal{I}$, and $0 \leq j \leq \log n$ describing the transitions of the Mealy machine $\mathbb{M}$. We only require a logarithmic number of bits to encode the target of a transition in binary, where we use $\textsc{trans}(m, \nu) \circ j$ for $\circ \in \{<, \leq, =, \geq, >\}$ to denote an appropriate encoding of the relation of the ranking to some value $j \in [n]$ or other rankings $j = \textsc{trans}(m', \nu')$.

- $\textsc{label}(m, \nu, o)$ for all $m \in M$, $\nu \in 2^\mathcal{I}$ and $o \in \mathcal{O}$ describing the labels of each transition.

- $\textsc{rgstate}(m, q)$ for all $m \in M$ and $q \in Q$, to denote the reachable states of the run graph $G_{\mathfrak{A}_\varphi,\mathbb{M}}$ of $\mathbb{M}$ and $\mathfrak{A}_\varphi$. Reachable vertices $(m, q)$ denote that their ranking $\lambda(m, q)$ is a natural number, *i.e.*, that $\lambda(m, q) \in \mathbb{N}$.

- $\textsc{ranking}(m, q, i)$ for all $m \in M$, $q \in Q$ and $0 \leq i \leq \log(n \cdot k)$ denoting the ranking of a vertex $(m, q)$ of $G_{\mathfrak{A}_\varphi,\mathbb{M}}$. Similar to the variables that encode the transition relation, we only require a logarithmic number of bits to encode the ranking in binary. In the same fashion, these encoded values can be compared using operations $\circ \in \{<, \leq, =, \geq, >\}$.

The Bounded Synthesis problem then is encoded through the SAT formula $\Psi_{BS}(\mathfrak{A}, n)$ consisting of the following constraints:

1. The rankings are bounded by $n \cdot k$ and the transitions are bounded by $n$ (the conditions are necessary due to the logarithmic encodings):

$$\bigwedge_{m\in M,\, \nu\in 2^\mathcal{I}} \textsc{trans}(m, \nu) < n \;\wedge \bigwedge_{m\in M,\, q\in Q} \textsc{ranking}(m, q) < n \cdot k$$

2. The initial state $(m_I, q_I)$ of the run graph is reachable:

$$\textsc{rgstate}(m_I, q_I)$$

3. Each ranking of a vertex of the run graph bounds the number of visited accepting vertices, not counting the current vertex itself:

$$\bigwedge_{m \in M,\, q \in Q} \textsc{rgstate}(m, q) \rightarrow \bigwedge_{\nu \in 2^{\mathcal{I}}, q' \in Q} \Delta_{\mathfrak{A}}(q, q')[\nu \mapsto \mathit{true}, (\mathcal{I} \setminus \nu) \mapsto \mathit{false}][o \mapsto \textsc{label}(m, \nu, o)] \rightarrow \bigwedge_{m' \in M} \textsc{trans}(m, \nu) = m' \rightarrow \textsc{rgstate}(m', q') \wedge \textsc{ranking}(m, q) \prec_q \textsc{ranking}(m', q')$$

where $\prec_q$ equals $<$, if $q \in R$, and $\prec_q$ equals $\leq$, otherwise. The assignment $\Delta_{\mathfrak{A}}(q, q')[\nu \mapsto \mathit{true}, (\mathcal{I} \setminus \nu) \mapsto \mathit{false}][x \mapsto \textsc{label}(m, \nu, x)]$ denotes that we replace every variable $y$ of the formula $\Delta_{\mathfrak{A}}(q, q')$ with *true*, if $y \in \nu$, and with *false*, otherwise. Afterwards, all remaining variables $o \in \mathcal{O}$ are replaced with $\textsc{label}(m, \nu, o)$.

**Theorem 13.** *The SAT formula $\Psi_{BS}(\mathfrak{A}, n)$ is satisfiable if and only if there exists a Mealy machine $\mathbb{M}$ with $|\mathbb{M}| = n$ such that $\mathcal{L}(\mathbb{M}) \subseteq \mathcal{L}(\mathfrak{A})$.*

*Proof.* Let $M = [n]$. The satisfaction of the first constraint limits the targets $\textsc{trans}(m, \nu)$ by $n$ and the rankings $\textsc{ranking}(m, q)$ by $n \cdot k$ for all $m \in M$ and $\nu \in 2^{\mathcal{I}}$ such that they induce a valid Mealy machine $\mathbb{M}$. The second constraint then initiates the run graph construction through marking the initial vertex. Finally, it is executed by the third constraint through fixing the variables $\textsc{rgstate}(m, q)$ and $\textsc{ranking}(m, q, j)$ for all $m \in M$, $q \in Q$ and $0 \leq j \leq \log(n \cdot k)$ such that they yield a ranking $\lambda \colon M \times Q \to [n \cdot k] \cup \{\_\}$ with $\lambda(m_0, q_0) \in \mathbb{N}$. To this end, the value encoded by $\textsc{ranking}(m, q, j)$ for $\lambda(m, q)$ is chosen if and only if $\textsc{rgstate}(m, q)$ equals *true*. Theorem 2 states that a proof that $\lambda$ indeed validates $G_{\mathfrak{A},\mathbb{M}}$ is sufficient, which is given by construction, due to the last constraint matching the requirements of Definition 11. □

With the result for Theorem 13 at hand, we obtain a synthesis procedure that implements the bounded synthesis approach. In terms of complexity, we first need to convert the LTL specification into an equivalent universal co-Büchi word automaton, which imposes an exponential blow up in the size of the specification in the worst case. Correspondingly, the created SAT encoding is bounded by $m$ with $m \in O(2^{|\varphi|} \times n)$ and the problem can be solved in non-deterministic polynomial time in $m$.

## 2 Bounded Cycle Synthesis

The advantage of Bounded Synthesis against game based synthesis approaches is the availability of the bound as additional input parameter constrainting the search space. Imposing a bound on the solution size comes with the intention that small solutions are preferable in contrast to larger ones. From the system creator's perspective, there are, however, more output requirements than just the solution size in general. Reconsidering our previous discussions, solutions that are easy to inspect and easy to understand usually are preferred, since inspectability and understandability are fundamental requirements in order to verify that the created solutions indeed realize the system designer's indents. While small solutions clearly improve the situation, they still cannot guarantee that there is no unnecessary complexity in the solution that is not strongly required as part of the specification. Towards this end, we, thus, need better output sensitive metrics that target the structural quality of the generated solutions directly.

Regarding the definition of a Mealy machine $\mathbb{M}$, one major structural metric, which is not already considered with bounded synthesis, is the transition relation $\delta_{\mathbb{M}}$ of $\mathbb{M}$. Correspondingly, unnecessary complexity may be introduced by the synthesizer using additional system transitions that are avoidable in general.

We are interested in a quality metric that forbids these unnecessary transitions through the adjustment of additional parameters, which precisely capture the underling complexity. A simple adjustment is given by a bound on the number of edges of $\delta_{\mathbb{M}}$. However, similar to a bound on the number of states, such a bound still is completely disconnected from the actual execution behavior. Hence, while the bound can lead to solutions that are structurally more simple, it is still completely at random, whether the solution finally is easier to understand. Therefore, we want a metric that is affected by the complexity of the execution behavior as well.

Regarding this behavior, we are confronted with an infinite set of infinite traces that result form unrolling the finite system model of a Mealy machine into the infinite tree of possible system executions. Infinite models, however, are not suitable to be handled by finite sets of system constraints. Instead, we need an intermediate finite representation that still sufficiently coves the infinite execution behavior. Our basic idea is to cut the infinite system executions to into finitely many pieces, each of finite length, such that they can be considered individually, but still are connected sufficiently enough to cover all important aspects of the system behavior. To this end, we choose the simple cycles, as induced by the graph structure of the realizing Mealy machine.

A simple cycle $c$ is defined as a sequence of states of the Mealy machine $\mathbb{M}$ that contains no state more than once and every pair of subsequent states, as well as the last and first states, are connected by at least one transition.

**Definition 19.** Let $G = (V, E)$ be a directed graph. A *simple cycle* $c$ of $G$ is a a tuple $(C, \eta)$, consisting of a non-empty set $C \subseteq V$ and a bijection $\eta \colon C \mapsto C$ such that

- $\forall v \in C.\ (v, \eta(v)) \in E$ and
- $\forall v \in C.\ k \in \mathbb{N}.\ \eta^k(v) = v \ \Leftrightarrow\ k \bmod |C| = 0$,

where $\eta^k$ denotes $k$ times the application of $\eta$.

In other words, a simple cycle of $G$ is a path through $G$ that starts and ends at the same vertex and visits every vertex of $V$ at most once. We say that a simple cycle $c = (C, \eta)$ has length $k$ iff $|C| = k$. We extend the definition of a simple cycle of a graph $G$ to a Mealy machine $\mathbb{M} = (2^{\mathcal{I}}, 2^{\mathcal{O}}, M, m_I, \delta_{\mathbb{M}}, \ell)$ such that $c$ is a simple cycle of $\mathbb{M}$ if and only if $c$ is a simple cycle of the graph $(M, E)$ for $E = \{(m, m') \mid \exists \nu \in 2^{\mathcal{I}}.\ \delta(m, \nu) = m'\}$. Thus, we ignore the input labels of the edges of $\mathbb{M}$. The set of all simple cycles of a Mealy machine $\mathbb{M}$ is denoted by $\mathtt{C}(\mathbb{M})$.

Every simple cycle covers a specific part of an execution trace that cannot only be repeated, but also immediately induces some infinite execution behavior. Therefore, we are not interested in properties of the behaviors that are induced by these cycles in particular, but already consider their existence as an interesting system metric in general. Our intuition is that a system with many of these simple cycles must be much more complex than one with less cycles, since there are more *"behavior pieces"* that can be used to be assembled up to infinity. Conversely, if we introduce a bound on the number of simple cycles of the resulting Mealy machine, then synthesis should produce systems that are much easier to understand.

Another interesting aspect of the number of cycles as a quality metric is that the amount of cycles, which may be required in order to satisfy an LTL specification, can explode in the size of the specification such that the corresponding increase is even worse than for the number of states. While the maximal number of states of a realizing Mealy machine is bounded doubly exponential the size of the formula, the maximal number of simple cycles can even rise up to triply exponential. Thus, the impact of the specification size on the number of cycles is even more dramatic than on the number of states.

We present a new synthesis algorithm that imposes two bounds on the solution space: the standard state bound $n$, bounding the number of states, and an additional bound $z$, which bounds the number of simple cycles. The algorithm is inspired by Tiernan's cycle counting algorithm for directed graphs from 1970 [138] and leverages an exhaustive search. To this end, the graph is unfolded into a tree from some arbitrary, but fix vertex $v$, such that no vertex repeats on any branch. The vertices in this tree with an outgoing edge that leads back to $v$ cover all simple cycles in the graph through $v$. The overall number of cycles, thus, can be counted by first unfolding the graph from $v$, counting the cycles through $v$, removing $v$ from the graph, and repeating the same procedure on the remaining sub-graph, until finally the graph is empty. The overall number of cycles results from the sum of the individual counts.

We integrate Tiernan's algorithm with the bounded synthesis constraint system. Therefore, the bounded synthesis constraints still witness the realizing Mealy machine $\mathbb{M}$ and the ranking $\lambda$, but we also add some additional *bounded cycle synthesis* constraints that, on the one hand, witness the corresponding forest of unfoldings, and, correspondingly, some rankings that bound the number of cycles.

## 2.1 Cycle Bounds

Our goal is a synthesis approach for producing systems that not only satisfy the specification, but at the same time are easy to understand. Therefore, we first give some theoretical arguments that underline the choice of the number of cycles as a system metric. We show that it is possible that the number of simple cycles explodes, even if the number of states stays small, and even if the specification enforces a large implementation. Our results indicate that a bound on the number of cycles is necessary in order to avoid these cases systematically. Moreover, our results show that bounding the number of states alone is not sufficient in order to obtain a simple and understandable solution.

### 2.1.1 Upper bounds

We prove that the number of cycles of a Mealy machine $\mathbb{M}$, implementing an LTL specification $\varphi$, indeed is bounded triply exponential in the size of $\varphi$. Towards this result, we first establish an upper bound the number of cycles of an arbitrary graph $G$ with a bounded out-degree $d$.

If there is no bound on the out-degree of the edge relation, then the maximal number of cycles is covered by a fully connected graph. In this case, each simple cycle is equivalent to a permutation of states. Correspondingly,

we derive an upper bound of $2^{n \log n}$ cycles for a graph with $n$ states. For our proof, however, we require a more involved argument based on an improved bound of $2^{n \log(d+1)}$ for graphs with bounded out-degree $d$. Consider that for LTL, the size of the implementing Mealy machine explodes in the number of states, while the out-degree remains constant in the number of input and output propositions.

**Lemma 1.** *Let $G = (V, E)$ be a directed graph with $|V| = n$ and with maximal out-degree $d$. Then $G$ has at most $2^{n \log(d+1)}$ simple cycles.*

*Proof.* We prove the result by induction over $n \in \mathbb{N}$. The base case is trivial, so let $n > 1$ and let $v \in V$ be some arbitrary vertex of $G$. By induction hypothesis, the subgraph $G'$, obtained from $G$ by removing $v$, has at most $2^{(n-1) \log(d+1)}$ simple cycles. Each of these simple cycles is also a simple cycle in $G$ leaving only the simple cycles of $G$ containing $v$. In each of these remaining simple cycles, $v$ has one of $d$ possible successors in $G'$ and from each such successor $v'$ we have again $2^{(n-1) \log(d+1)}$ possible simple cycles in $G'$ returning to $v'$. Hence, if we *redirect* these simple cycles to $v$ instead of $v'$, i.e., we insert $v$ before $v'$ in the simple cycle, then we cover all possible simple cycles of $G$ containing $v$. Note that not every such edge needs to exist for a concrete given graph. However, in our worst-case analysis, every possible cycle is accounted for. All together, we obtain an upper bound of

$$2^{(n-1) \log(d+1)} + d \cdot 2^{(n-1) \log(d+1)} = 2^{n \log(d+1)}$$

cycles in $G$. □

The result can be leveraged towards an upper bound on the number of simple cycles of a Mealy machine $\mathbb{M}$.

**Lemma 2.** *Let $\mathbb{M}$ be a Mealy machine. Then $|\mathfrak{C}(\mathbb{M})| \in O(2^{|\mathbb{M}| \cdot |\mathcal{I}|})$.*

*Proof.* The Mealy machine $\mathbb{M}$ has an out-degree of $2^{|\mathcal{I}|}$. Thus, by Lemma 1, the number of simple cycles is bounded by $2^{|\mathbb{M}| \log(2^{|\mathcal{I}|}+1)} \in O(2^{|\mathbb{M}| \cdot |\mathcal{I}|})$. □

We obtain an upper bound on the number of simple cycles for every Mealy machine that realizes an LTL specification $\varphi$.

**Theorem 14.** *For every realizable LTL specification $\varphi$ there is a Mealy machine $\mathbb{M}$ with $\mathcal{L}(\mathbb{M}) \subseteq \mathcal{L}(\varphi)$, which has at most triply exponential many cycles in $|\varphi|$.*

*Proof.* According to [90], [114], and [45] there exists a Mealy machine $\mathbb{M}$ with $\mathcal{L}(\mathbb{M}) \subseteq \mathcal{L}(\varphi)$, whose size is bounded doubly exponential in $|\varphi|$. In combination with the result of Lemma 2 we obtain the desired result. □

### 2.1.2 Lower bounds

Next, we prove that the bound of Theorem 14 is tight. We prove that for all $n \in \mathbb{N}$ there exists a realizable LTL specification $\varphi$ with $|\varphi| \in \Theta(n)$, for which every implementation of $\varphi$ has at least triply exponential many cycles in $n$. The structure of the presented proof is inspired by Alur [1], who uses a similar argument to prove lower bounds on the distance of the longest paths through synthesized Mealy machines $\mathbb{M}$. We utilize a lemma showing that the overall number of cycles can be exponential in the length of the longest cycle of $\mathbb{M}$.

**Lemma 3.** *Let $\varphi$ be a realizable LTL specification, for which every realizing Mealy machine $\mathbb{M}$ contains a simple cycle of length $n$. Then there is a realizable LTL specification $\psi$, such that every Mealy machine $\mathbb{M}'$ implementing $\psi$ contains at least $2^n$ many simple cycles.*

*Proof.* Let $a$ and $b \in$ be a fresh input and output atomic propositions, respectively, which do not appear in $\varphi$, and let $\mathbb{M} = (2^{\mathcal{I}}, 2^{\mathcal{O}}, M, m_I, \delta_{\mathbb{M}}, \ell)$ be some implementation that realizes $\varphi$. We define $\psi := \varphi \wedge \square(a \leftrightarrow \bigcirc b)$ and choose the implementation $\mathbb{M}'$ such that

$$\mathbb{M}' = (2^{\mathcal{I} \cup \{a\}}, 2^{\mathcal{O} \cup \{b\}}, M \times 2^{\{b\}}, (m_I, \emptyset), \delta'_{\mathbb{M}}, \ell'),$$

with $\ell'((m, s), \nu) = \ell(m, \mathcal{I} \cap \nu) \cup s$ for all $m \in M, s \in 2^{\{b\}}$ and $\nu \in 2^{\mathcal{I} \cup \{a\}}$ and

$$\delta'((m, s), \nu) = \begin{cases} (\delta(m, \mathcal{I} \cap \nu), \emptyset) & \text{if } a \in \nu \\ (\delta(m, \mathcal{I} \cap \nu), \{b\}) & \text{otherwise} \end{cases}$$

We clearly have that $\mathbb{M}'$ is an implementation of $\psi$. The Mealy machine remembers each input $a$ for one time step and then outputs the stored value. Thus, it satisfies $\square(a \leftrightarrow \bigcirc b)$. At the same time, $\mathbb{M}'$ still satisfies $\varphi$. Hence, $\psi$ must be realizable as well.

We continue by picking an arbitrary implementation $\mathbb{M}''$ of $\psi$ that must exist according correspondingly. After projecting away the fresh signals $a$ and $b$ from $\mathbb{M}''$, we again obtain an implementation for $\varphi$ that contains a cycle $(C, \eta)$ of length $n$. Let $C = \{m_0, m_1, \ldots, m_{n-1}\}$. We obtain that $\mathbb{M}''$ contains at least the cycles

$$X = \{(\{(m_i, f(m_i)) \mid i \in [n]\}, (m, s) \mapsto (\eta(m), f(\eta(m)))) \mid f \colon C \to 2^{\{b\}}\},$$

which concludes the proof, since $|X| = 2^n$. □

With Lemma 3 at hand, we have everything together for proving that the aforementioned lower bounds indeed are tight. Note that in the following proof all LTL specifications only require the temporal operators $\bigcirc$, $\square$ and $\diamond$. Thus, the proven bounds even hold for a restricted fragment of LTL.

**Theorem 15.** *For every $n > 1$, there is a realizable specification $\varphi_n$ with $|\varphi_n| \in \Theta(n)$, for which every realizing Mealy machine $\mathbb{M}_n$ has at least triply exponential many cycles in $n$.*

*Proof.* According to Lemma 3, it suffices to provide a realizable LTL specification $\varphi_n$ that contains at least one cycle of doubly exponential length in $n$. We choose

$$\varphi_n \;:=\; \square\,(\underbrace{\diamond \bigwedge_{i=1}^{n} (a_i \to \diamond b_i)}_{\varphi_n^{prem}} \to \underbrace{\diamond \bigwedge_{i=1}^{n} (c_i \to \diamond d_i)}_{\varphi_n^{con}}) \;\leftrightarrow\; \square\diamond s$$

where $\mathcal{I} = \mathcal{I}_a \cup \mathcal{I}_b \cup \mathcal{I}_c \cup \mathcal{I}_d$ with $\mathcal{I}_x = \{x_1, x_2, \ldots, x_n\}$ for all $x \in \{a, b, c, d\}$ and $\mathcal{O} = \{s\}$. The specification describes a monitor, which checks whether the invariant $\diamond\varphi_n^{prem} \to \diamond\varphi_n^{con}$ over the input signals $\mathcal{I}$ is satisfied. The satisfaction is signaled through the output $s$, which needs to be triggered infinitely often, as long as the invariant stays satisfied.

In the sequel, we denote a subset $x \subseteq \mathcal{I}_x$ with the $n$-ary vector $\vec{x}$ over $\{0, 1\}$, where the $i$-th entry of $\vec{x}$ is set to 1 if and only if $x_i \in x$. The specification $\varphi_n$ is realizable. First, consider that for the fulfillment of $\varphi_n^{prem}$ ($\varphi_n^{con}$), an implementation $\mathbb{M}$ needs to store the set of all requests $\vec{a}$ ($\vec{c}$), whose 1-positions have not yet been released by a corresponding response $\vec{b}$ ($\vec{d}$). Furthermore, for monitoring the complete invariant $\diamond\varphi_n^{prem} \to \diamond\varphi_n^{con}$, $\mathbb{M}$ has to guess at each point in time, whether $\varphi_n^{prem}$ will be satisfied in the future (under

the current request $\vec{a}$). For the realization of this guess, $\mathbb{M}$ needs to store a mapping $f$, which maps each open request $\vec{a}$ to the corresponding set of requests $\vec{c}$ [1]. This way, $\mathbb{M}$ can look up the set of requests $\vec{c}$, tracked since the last occurrence of $\vec{a}$, whenever $\vec{a}$ gets released by a corresponding vector $\vec{b}$. If this is the case, it continues to monitor the satisfaction of $\varphi_n^{con}$ (if not already satisfied) and finally adjusts the output signal $s$, correspondingly. Note that $\mathbb{M}$ still has to continuously update and store the mapping $f$, since the next satisfaction of $\varphi_n^{prem}$ may already start while the satisfaction of the current $\varphi_n^{con}$ is still checked. There are double exponentially many such mappings $f$, hence, $\mathbb{M}$ needs to be at least doubly exponential in $n$.

It remains to show that every such implementation $\mathbb{M}$ contains a cycle of at least doubly exponential length. By the aforementioned observations, we can assign each state of $\mathbb{M}$ a mapping $f$, that maps vectors $\vec{a}$ to sets of vectors $\vec{c}$. By interpreting the vectors as numbers, encoded in binary, we obtain that $f$ is a function $f\colon \{1, 2, \ldots, 2^n\} \mapsto 2^{\{1,2,\ldots,2^n\}}$. We map each such mapping $f$ to a binary sequence $b_f = b_0 b_1 \ldots b_m \in \{0,1\}^t$ with $t = 2^n$, where each bit $b_i$ of $b_f$ is set to 1 if and only if $i \in f(i)$. It is easy to observe, that if two binary sequences are different, then their related states have to be different as well.

To conclude the proof, we show that the environment has a strategy to manipulate the bits of the associated sequences $b_f$ via the inputs $\mathcal{I}$. To set bit $b_i$, the environment chooses the requests $\vec{a}$ and $\vec{c}$ such that they represent $i$ in binary. The remaining inputs are fixed to $\vec{b} = \vec{d} = \vec{0}$. Hence, all other bits are not affected, as possible requests of previous $\vec{a}$ and $\vec{c}$ remain open. To reset bit $b_i$, the environment requires multiple steps. First, it picks $\vec{a} = \vec{c} = \vec{d} = \vec{0}$ and $\vec{b} = \vec{1}$. This does not affect any bit of the sequence $b_f$, since all requests introduced through vectors $\vec{c}$ are still open. Next, the environment executes the aforementioned procedure to set bit $b_j$ for every bit currently set to 1, except for the bit $b_i$, it wants to reset. This refreshes the requests introduced by previous vectors $\vec{a}$ for every bit, except for $b_i$. Furthermore, it does not affect the sequence $b_f$. Finally, the environment picks $\vec{a} = \vec{b} = \vec{c} = \vec{0}$ and picks $\vec{d}$ such that it represents $i$ in binary. This removes $i$ from every entry in $f$, but only resets $b_i$, since all other bits are still open due to the previous updates.

With these two operations, the environment can enforce any sequences of sequences $b_f$, including a binary counter counting up to $2^{2^n}$. As different states are induced by the different sequences, we obtain a cycle of doubly exponential length in $n$ by resetting the counter at every overflow. □

[1] Note that this representation is open for many optimizations. However, they will not affect the overall complexity result. Thus, we ignore them for the sake of readability here.

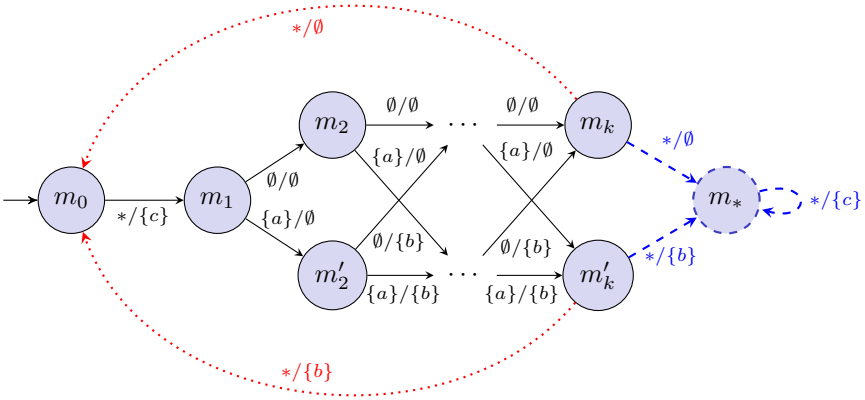



Figure 29: The Mealy machines $\mathbb{M}_n$ (red/dotted) and $\mathbb{M}'_n$ (blue/dashed), which share all solid black edges of the illustration.

### 2.1.3 Trade-offs between states and cycles

We close this section with some observations that consider the trade-offs between synthesizing implementations that are minimal in the number of states and implementations that are minimal in the number of cycles. Unfortunately, it turns out that we cannot archive both optima simultaneously.

**Theorem 16.** *For every $n > 1$, there is a realizable LTL specification $\varphi_n$ with $|\varphi| \in \Theta(n)$, for which*

- *there is an implementation of $\varphi$ consisting of $n$ states and*
- *there is an implementation of $\varphi$ containing $m$ simple cycles,*
- *but there is no realization of $\varphi$ with $n$ states and $m$ simple cycles.*

*Proof.* Consider the specification

$$\varphi_n \;:=\; \big(\neg b \wedge c\big) \;\wedge\; \bigcirc^{k+2}\big(\neg b \wedge c\big) \;\wedge\; \bigwedge_{i=1}^{k} \bigcirc^{i}\big(\neg c \wedge \bigcirc \neg c \wedge (a \leftrightarrow \bigcirc b)\big)$$

over $\mathcal{I} = \{a\}$ and $\mathcal{O} = \{b, c\}$, where $\bigcirc^i$ denotes $i$ times the application of $\bigcirc$. The specification $\varphi_n$ is realizable with at least $n = 2k + 1$, as witnessed by

the Mealy machine $\mathbb{M}_n$ depicted in Figure 29. In particular, $\mathbb{M}_n$ has $z = 2^k$ many cycles. The blowup can be avoided by spending the implementation at least one more state, which reduces the number of cycles to $z = 1$. The corresponding implementation $\mathbb{M}'_n$ is also depicted in Figure 29. □

The result underlines the possibility of an explosion in the number of simple cycles and illustrates that this explosion cannot be avoided in general. However, at the same time it demonstrates that there are cases, where the amount of simple cycles can only be improved by choosing a larger solution.

## 2.2 Counting Cycles

Our goal is extending the bounded synthesis constraint system such that it supports an additional bound on the number of simple cycles of the synthesized Mealy machines. Unfortunately, the pure introduction of a bound still does not solve our problem. We also must be able to verify it against the concrete number of simple cycles of the solution graph, which, on the other hand, requires that we have knowledge of this number in the first place. But how to obtain the number of simple cycles of a Mealy machine? To answer this question, we first review a classical algorithm from Tiernan [138] that was invented in 1970 to count the number of simple cycles of a directed graph $G$. On the one hand, Tiernan's algorithm highlights important insights on the complexity of the problem. On the other hand, it serves as an inspiration for the construction of our bounded cycle synthesis constraint system.

**Algorithm 2** Tiernan's Cycle Counting Algorithm

**Input:** directed graph $G = (V, E)$

```
 1: c := 0
 2: procedure COUNT(Ṽ, Ẽ, v_r, v)
 3:     if (v, v_r) ∈ Ẽ then
 4:         c := c + 1
 5:     for all v' ∈ Ṽ with (v, v') ∈ Ẽ do
 6:         COUNT(Ṽ \ {v}, Ẽ, v_r, v')
 7: while V ≠ ∅ do
 8:     pick some arbitrary v_r ∈ V
 9:     V := V \ {v_r}
10:     COUNT(V, E, v_r, v_r)
11:     E := E ∩ (V × V)
12: return c
```

The algorithm starts with an iteration over all cycles that contain the first picked vertex $v_r$ (Line 10). This iteration is realized through an unfolding of the graph into a tree, as implemented by COUNT (Line 2), rooted in $v_r$, such that there is no repetition of vertices on any path from the root to a leaf, as guaranteed via the restriction to $\tilde{V}$. The number of edges of $E$ that lead back to $v_r$ represents the corresponding number of cycles through $v_r$, which is counted in Line 4. All remaining cycles of $G$, not counted during this instantiation of COUNT, do not contain $v_r$ and, thus, are cycles of the sub-graph $G'$, from which $v_r$ has been removed (Lines 9 and 11). Therefore, the remaining cycles are counted by recursively counting the cycles of $G'$. The algorithm terminates as soon as $G'$ gets empty and returns the sum of all individual countings COUNT.

The algorithm is correct [138]. However, the unfolded trees can be exponential in the size of the graph, even if none of their vertices are connected to the root, because there is no cycle to be counted. An example of this weakness of Tiernan's algorithm is illustrated by the graph $\mathbb{M}'_n$, as depicted in Figure 29. Fortunately, the problem can be solved by first reducing the graph to its strongly connected components (SCCs) such that the cycles of each SCC can be counted individually [146, 77]. Therefore, note that a simple cycle never leaves an SCC of the graph.

The result is an efficient counting algorithm that is exponential in the size of the graph, but linear in the number of cycles. Furthermore, the algorithm stays within a limit on the execution time between two cycles detections, which is bounded linear in the size of the graph.

## 2.3 SAT Encoding

We leverage the insights from Tiernan's algorithm to create an extended constraint system, which not only bounds the number of states of the resulting Mealy machine $\mathbb{M}$, but also the number of simple cycles of $\mathbb{M}$. Our targeted constraint system is SAT. Thus, our encoding requires a witness structure for imposing a bound on the number of cycles that can be verified in polynomial time in the size of the encoding. For this purpose, we use the unfolded trees, as they are induced recursively by the calls of Tiernan's algorithm to COUNT, in combination with a ranking function that labels those trees. The combination imposes a limit on the number of cycles, as they are induced by the witnessed trees individually. We call a tree that witnesses $z$ cycles in $G$, all containing the root $r$ of the tree, a *witness-tree* $t_{r,z}$ of $G$.

**Definition 20.** A *witness-tree* $t_{r,z}$ of $G = (V, E)$ is a labeled graph $t_{r,z} = ((W, B \cup R, \eta)$, consisting of a graph $(W, B \cup R)$ with $z = |R|$ and a labeling function $\eta\colon W \to V$, such that:

1. The edges are partitioned into blue edges $B$ and red edges $R$.
2. All red edges lead back to the root: $R \subseteq W \times \{r\}$
3. No blue edges lead back to the root: $B \cap W \times \{r\} = \emptyset$
4. Each non-root has at least one blue incoming edge:
$$\forall w' \in W \setminus \{r\}.\ \exists w \in W.\ (w, w') \in B$$
5. Each vertex has at most one blue incoming edge:
$$\forall w_1, w_2, w \in W.\ (w_1, w) \in B \wedge (w_2, w) \in B \Rightarrow w_1 = w_2$$
6. The graph is labeled by an unfolding of $G$:
$$\forall (w, w') \in B \cup R.\ (\tau(w), \tau(w')) \in E$$
7. The unfolding is complete:
$$\forall w \in W.\ \forall v' \in V.\ (\tau(w), v') \in E \\ \rightarrow \exists w' \in W.\ (w, w') \in B \cup R\ \wedge\ \tau(w') = v'$$
8. Let $w_i, w_j \in W$ be two different vertices that appear on a path from the root to a leaf in the $r$-rooted tree $(W, B)$[a]. Then the labeling of $w_i$ and $w_j$ differs, i.e., $\tau(v_i) \neq \tau(v_j)$.
9. The root identifies the corresponding vertex of $G$, i.e., $\tau(r) = r$.

[a] The tree property is enforced by Conditions 3 to 5

This definition covers all properties that are required for bounding the number of simple cycles of a Mealy machine. For proving the statement, we first consider the simplified situation of a graph consisting only of a single SCC.

**Lemma 4.** *Let $G = (V, E)$ be a graph consisting of a single SCC, $r \in V$ be a vertex of $G$ and $m$ be the number of cycles of $G$ containing $r$. Then there is a witness-tree $t_{r,z} = ((W, B \cup R), \tau)$ of $G$ with $|W| \leq m \cdot |V|$.*

*Proof.* We construct the tree $t_{r,z}$ according to the strategy of Algorithm 2. Hence, an edge is colored red if and only if it leads back to the root. The constructed tree satisfies all Conditions 1 to 9. By correctness of Algorithm 2, we have that $|R| = z$.

Now, for the sake of contradiciton, assume $|W| > z \cdot |V|$. First we observe, that the depth of the tree $(W, B)$ must be bounded by $|V|$ to satisfy Condition 8. Hence, as there are at most $z$ red edges in $t_{r,z}$, there must be a vertex $w \in W$ without any outgoing edges. However, since $G$ is a single SCC, this contradicts the completeness of $t_{r,z}$ (Condition 7). □

**Lemma 5.** *Let $G = (V, E)$ be a graph consisting of a single SCC and let $t_{r,z}$ be a witness-tree of $G$. Then there are at most $z$ cycles in $G$ that contain $r$.*

*Proof.* Let $t_{r,z} = ((W, R \cup B), \tau)$. Assume for the sake of contradiction that $G$ has more than $z$ cycles and let $c = (C, \eta)$ be one of these cycles. By the completeness of $t_{r,z}$, there is path $w_0 w_1 \dots w_{|C|-1}$ with $w_0 = r$ and $\tau(w_i) = \eta^i(r)$ for all $0 \leq i < |C|$. From $w_i \neq r$ and Condition 2, it follows $(w_{i-1}, w_i) \in B$ for all $0 < i < |C|$, $\eta^{|C|}(r) = r$, and $(w_{|C|-1}, w_0) \in R$. By the tree shape of $(W, B)$, we get $|R| > z$, yielding the desired contradiction. □

From Lemmas 4 and 5 we derive that $t_{r,z}$ is a suitable witness for bounding the number of simple cycles of an implementation $\mathbb{M}$. Furthermore, from Lemma 4 we also obtain an upper bound on the size of $t_{r,z}$.

Note that the results of Lemmas 4 and 5 are only valid for a graphs that consist of a single SCC. In general, this is no restriction, since for counting and bounding the simple cycles of a graph, the graph can always be split into its individual SCCs at first. Remember that no cycle can be part of multiple SCCs. Such a split, however, also must be realized as part of our encoding.

In the following, we describe an encoding that guesses SCC annotations $\Psi_{SCC}(n, k)$ for each sub-graph $G_k$ of a given graph $G$ bounded by $n \in \mathbb{N}$. Concretely, we fix some vertex $v$ in each SCC, for which we guess two spanning trees that are rooted in $v$, where the edge relation of the second tree is inverted with respect to $G_k$, *i.e.*, the edges lead back to the root. This way, we ensure that, starting at vertex $v$, each vertex is reachable and from each other, as required for a sub-graph being an SCC. Correspondingly, the spanning trees witness the guessed SCCs. In order to ensure that the SSCs are maximal, we enforce that the DAG of all SCCs is totally ordered.

Let $W_k = [k, n-1]$ be some ordered set. We introduce the following variables that witness the SCCs of the sub-graph $G_k$:

- $\text{EDGE}(w, w')$ for all $w, w' \in W_k$ representing the edges of the abstraction of the Mealy machine $\mathbb{M}$ to $G_k$.
- $\text{SCC}_k(w, i)$ for all $w \in W_k$, and $0 \leq i \leq \log n$ denoting one SCC of $w$ in the $k$-th sub-graph $G_k$ of $G$.
- $\text{FORWARD}_k(w, w')$ for all $w, w' \in W_k$ representing the edges of the first spanning tree.
- $\text{BACKWARD}_k(w, w')$ for all $w, w' \in W_k$ representing the edges of the second spanning tree.
- $\text{FRANK}_k(w, i)$ for all $w \in W_k$ and $0 \leq i \leq \log n$ denoting a ranking that measures the distance from the root of the first spanning tree.
- $\text{BRANK}_k(w, i)$ for all $w \in W_k$ and $0 \leq i \leq \log n$ denoting a ranking that measures the distance to the root of the second spanning tree.

The split of $G_k$ into the individual SCCs then is realized through the SAT formula $\Psi_{SCC}(n, k)$ consisting of the following constraints:

1. The SCCs are totally ordered:

$$\bigwedge_{w,w' \in W_k} \text{EDGE}(w, w') \rightarrow \text{SCC}_k(w) \leq \text{SCC}_k(w')$$

2. Only edges of the same SCC are connected through a forward edge:

$$\bigwedge_{w,w' \in W_k} \text{FORWARD}_k(w, w') \rightarrow \text{EDGE}(w, w') \wedge \text{SCC}_k(w) = \text{SCC}_k(w')$$

3. Only edges of the same SCC are connected through a backward edge:

$$\bigwedge_{w,w' \in W_k} \text{BACKWARD}_k(w, w') \rightarrow \text{EDGE}(w, w') \wedge \text{SCC}_k(w) = \text{SCC}_k(w')$$

4. The roots of both rankings are equivalent:

$$\bigwedge_{w \in W_k} \text{FRANK}_k(w) = 0 \leftrightarrow \text{BRANK}_k(w) = 0$$

5. Each SCC has a root that is annotated with the smallest ranking:

$$\bigwedge_{0<i\leq n}\Big(\big(\bigvee_{w\in W_k}\textsc{scc}_k(w)=i\big)\rightarrow\big(\bigvee_{w\in W_k}(\textsc{scc}_k(w)=i\ \wedge\ \textsc{frank}_k(w)=0)\big)\Big)$$

6. Every SCC root is unique:

$$\bigwedge_{\substack{w,w'\in W_k\\ w\neq w'}}\neg\Big(\textsc{scc}_k(w)=\textsc{scc}_k(w')\ \wedge\ \textsc{frank}_k(w)=0\ \wedge\ \textsc{frank}_k(w')=0\Big)$$

7. Roots neither have incoming forward edges nor outgoing backward edges:

$$\bigwedge_{w'\in W_k}\textsc{frank}_k(w')=0\rightarrow\bigwedge_{w\in W_k}\neg\textsc{forward}_k(w,w')\wedge\neg\textsc{backward}_k(w,w')$$

8. All non-roots have exactely one incoming forward edge:

$$\bigwedge_{w'\in W_k}\textsc{frank}_k(w')\neq 0\rightarrow \mathit{exactly}_1(\{\textsc{forward}_k(w,w')\mid w\in W_k\})$$

9. All non-roots have exactely one outgoing backward edge:

$$\bigwedge_{w'\in W_k}\textsc{brank}_k(w')\neq 0\rightarrow \mathit{exactly}_1(\{\textsc{backward}_k(w',w)\mid w\in W_k\})$$

10. Forward edges preserve the ranking:

$$\bigwedge_{w,w'\in W_k}\textsc{forward}_k(w,w')\rightarrow\textsc{frank}_k(w)<\textsc{frank}_k(w')$$

11. Backward edges preserve the ranking:

$$\bigwedge_{w,w'\in W_k}\textsc{backward}_k(w,w')\rightarrow\textsc{brank}_k(w)>\textsc{brank}_k(w')$$

**Lemma 6.** *Let $G_k=(W_k,E)$ be a graph with $|W|=[k,n-1]$, encoded through variables* $\textsc{edge}(w,w')$ *with* $(w,w')\in E\leftrightarrow\textsc{edge}(w,w')$. *Then the SAT formula $\Psi_{SCC}(n,k)$ is satisfiable and for all $w,w'\in W_k$ we have that* $\textsc{scc}_k(w)=\textsc{scc}_k(w')$ *iff $w$ and $w'$ are part of the same maximal SCC.*

*Proof.* The variables $\text{SCC}_k(w)$ assign every vertex a unique SCC and rank them, correspondingly. Therefore, Constraint 1 ensures that the edge relation of $G_k$ preserves the ranking, which guarantees that the selected SCC partitioning must be maximal with respect to $G_k$. It remains to proof that each selected partition indeed is an SCC. To this end, the forward edges $\text{FORWARD}_k(w, w')$ and backward edges $\text{BACKWARD}_k(w, w')$ witness two spanning trees for each SCC with opposite edge directions, respectively, *i.e.*, in one of the trees the edges always point downwards from the root to the leaves and in the other one, the edges always point up from the leaves to the root. At the same time, the edges still follow the edge relation $\text{EDGE}(w, w)$ of $G$ and never leave an SCC (Constraints 2 and 3).

The rankings $\text{FRANK}_k(w)$ and $\text{BRANK}_k(w)$ ensure that the selected trees indeed are spanning trees. Each spanning tree has a unique root with a minimal rank. Moreover, we require that the two spanning trees of the same SCC have the identical root (Constraint 4). The existence of these roots is guaranteed through Constraint 5. That the roots are unique is guaranteed through Constraint 6. Finally, roots must have no incoming edges in case of the forward directed tree and no leaving edges in case of a backward direct tree, as provided through Constraint 7. With the roots being identified, the spanning trees unfold according to the respective edge relations. Therefore, inner nodes and leafs must have a unique parent (Constraints 8 and 9), while their ranks increase according to the depth of the tree (Constraints 10 and 11). The combination of all the requirements ensures that $\text{FORWARD}_k(w, w')$ and $\text{BACKWARD}_k(w, w')$ indeed induce two spanning trees for each SCC with an equivalent root and the aforementioned properties.

We claim that the existence of two such spanning trees witnesses that each set of vertices $w$, with equal identifier $\text{SCC}_k(w)$, indeed is an SCC of $G_k$. To this end, there must be a path from each vertex $w$ to every other vertex $w'$. Due to both trees being spanning tress, we always can construct this path as follows: starting in $w$, first follow the backward edges to the common root, from there use the forward edges to select a path to $w'$. Correspondingly, the annotation $\text{SCC}_k(w)$ indeed partitions $G_k$ into its maximal SCCs. □

For fixed $0 \leq k < n$, each formula $\Psi_{SCC}(n, k)$ is of quadratic size in $n$ and consists of $n^2$ many variables.

Now, we have everything at hand for the construction of our final encoding. At first, we derive a directed graph $G$ from the guessed implementation $\mathbb{M}$. Then, we guess the corresponding sub-graphs of $G$ via iteratively removing vertices and splitting them into their corresponding SCCs. Finally, we guess the witness-trees for each such SCC.

We further introduce some optimizations that are required for receiving a compact encoding. First, consider that there is no reason for the introduction of a fresh copy of each SCC, because the SCC of each vertex is always unique. Hence, it suffices to guess a ranking for each vertex separately. Next, the constraint system guesses $n$ trees $t_{i,r_i}$ consisting of at most $i \cdot n$ vertices each, such that the sum of all $i$ equals the overall number of simple cycles $z$. We could overestimate each $i$ with $z$ or guess the exact distribution of cycles over the different witness-trees $t_{i,r_i}$. However, there is a better solution. We guess all trees together in a single graph bounded by $z \cdot n$ instead. Furthermore, in order to avoid possible interleavings, we add an annotation of each vertex by its corresponding witness-tree $t_{i,r_i}$. Hence, instead of bounding the number of each $t_{i,r_i}$ separately through $i$, we bound the number of all red edges in the whole forest by $z$. Therefore, we not only reduce the size of the encoding, but also save the additional constrains, which otherwise would be necessary to sum the different witness-tree bounds $i$ to $z$.

Let $W = [n]$ and $S = W \times [z]$, where $W$ denotes the vertices of $G$ and $S$ the vertices of the forests $t_{i,r_i}$. Furthermore, let $T = W \times \{0\}$ be the roots and $N = S \setminus T$ be the non-roots of the corresponding trees. We introduce the following variables:

- $\textsc{bedge}(s, s')$ for all $s \in S$ and $s' \in N$ representing the blue edges.
- $\textsc{redge}(s, s')$ for all $s \in S$ and $s' \in T$ representing the red edges.
- $\textsc{wtree}(s, i)$ for all $s \in S$, $0 \leq i \leq \log n$ denoting the witness-tree of each $s \in S$. As for the bounded synthesis encoding, we use $\textsc{wtree}(s) \circ x$ to relate values with the underlying encoding.
- $\textsc{visited}(s, w)$ for all $s \in S$ and $w \in W$ denoting the set of vertices $t$ that already have been visited at $s$ since leaving the root of the witness-tree.
- $\textsc{rbound}(c, i)$ for all $c \in [z]$, $0 \leq i \leq \log(n \cdot z)$ representing ordered lists of edges that bound the number of red edges of the forest.

Note that we introduce $m$ explicit copies for each vertex of $G$, which is sufficient, due to each simple cycle containing each vertex at most once. Therefore, the labeling $\eta$ of vertices $s$ is derivable from the first component of $s$.

The selection of the corresponding witness trees then is realized through the formula $\Psi_{CS}(\mathfrak{A}, n, z)$ consisting of the following constraints:

1. The graph $G$ matches $\mathbb{M}$:

$$\bigwedge_{w,w' \in W} \Big( \textsc{edge}(w, w') \leftrightarrow \bigvee_{\nu \in 2^I} \textsc{trans}(w, \nu) = w' \Big)$$

2. Roots identify the witness-tree:

$$\bigwedge_{r\in W} \textsc{wtree}((r,0)) = r$$

3. Every available blue edge must be taken.

$$\bigwedge_{\substack{(w,c)\in S,\, r,w'\in W \\ w\neq w', w\neq r}} \begin{array}{l} \textsc{edge}(w,w') \ \wedge\ \textsc{scc}_r(w) = \textsc{scc}_r(w') \\ \wedge\ \textsc{wtree}((w,c)) = r \ \wedge\ \neg\textsc{visited}((w,c),w') \\ \qquad \rightarrow \bigvee_{0<c'\leq w} \textsc{bedge}((w,c),(w',c')) \end{array}$$

4. Every available red edge must be taken:

$$\bigwedge_{\substack{(w,c)\in S,\, r\in W \\ w\geq r}} \begin{array}{l} \textsc{edge}(w,r) \ \wedge\ \textsc{scc}_r(w) = \textsc{scc}_r(r) \ \wedge\ \textsc{wtree}((w,c)) = r \\ \qquad \rightarrow \textsc{redge}((w,c),(r,0)) \end{array}$$

5. The red edges match to the edges of the graph $G$:

$$\bigwedge_{(w,c)\in S,\, r\in W,} \textsc{redge}((w,c),(r,0)) \rightarrow \textsc{edge}(w,r)$$

6. The blue edges match to the edges of the graph $G$:

$$\bigwedge_{(w,c)\in S,\, (w',c')\in N} \textsc{bedge}((w,c),(w',c')) \rightarrow \textsc{edge}(w,w')$$

7. Red edges only connect vertices of the current $t_{i,r_i}$:

$$\bigwedge_{s\in S,\, (r,0)\in T} \textsc{redge}(s,(r,0)) \rightarrow \textsc{wtree}(s) = r$$

8. Blue edges only connect vertices of the current $t_{i,r_i}$:

$$\bigwedge_{s\in S,\, s'\in N} \textsc{bedge}(s,s') \rightarrow \textsc{wtree}(s) = \textsc{wtree}(s')$$

9. Every non-root has exactly one blue incoming edge:

$$\bigwedge_{s'\in N} \mathit{exactly}_1(\{\textsc{bedge}(s,s') \mid s\in S\})$$

10. Every vertex appears at most once on a path from the root to a leaf:

$$\bigwedge_{\substack{(w,c)\in S,\\ s\in N}} \text{BEDGE}((w,c),s) \rightarrow \neg\text{VISITED}(s,w) \wedge (\text{VISITED}(s,w') \leftrightarrow \text{VISITED}((w,c),w'))$$

11. Only non-roots can be successors of a root:

$$\bigwedge_{r\in W} \Big( \bigwedge_{\substack{w\in W\\ w\leq r}} \neg\text{VISITED}((r,0),w) \ \wedge \bigwedge_{\substack{w\in W\\ w>r}} \text{VISITED}((r,0),w)\Big)$$

12. The red edges are strictly ordered:

$$\bigwedge_{0<c\leq m} \text{RBOUND}(c) < \text{RBOUND}(c+1)$$

13. The list of red edges is complete. We use $f(s)$ to map each state of $S$ to a unique number in $\{1,\ldots,n\cdot z\}$:

$$\bigwedge_{s\in S,\, s'\in W} \text{REDGE}(s,s') \ \rightarrow \bigvee_{0<c\leq z} \text{RBOUND}(c) = f(s)$$

Given a universal co-Büchi automaton $\mathfrak{A}$, a bound $n$ on the number of states of $\mathbb{M}$, and a bound $z$ on the number of cycles of $\mathbb{M}$, the Bounded Cycle Synthesis problem is encoded via the following SAT formula:

$$\Psi_{CS}(\mathfrak{A},n,z) \ := \ \Psi_{BS}(\mathfrak{A},n) \ \wedge \bigwedge_{0<k\leq n} \Psi_{SCC}(n,k) \ \wedge \ \Psi_{WT}(\mathfrak{A},n,z)$$

**Theorem 17.** *For bounds $n,z\in\mathbb{N}$ and a universal co-Büchi automaton $\mathfrak{A}$, the formula $\Psi_{CS}(\mathfrak{A},n,z)$ is satisfiable if and only if there is a Mealy machine $\mathbb{M}$ with $|\mathbb{M}|=n$ and $\mathfrak{C}(\mathbb{M})=z$, that is accepted by $\mathfrak{A}$.*

*Proof.* "⇒": Assume that $\Psi_{CS}(\mathfrak{A},n,z)$ is satisfiable. Thus, $\Psi_{BS}(\mathfrak{A},n)$ is satisfiable as well. Hence, according to Theorem 13, every correspondingly encoded Mealy machine $\mathbb{M}$ must be accepted by $\mathfrak{A}$ and satisfies $|\mathbb{M}|=n$. It only remains to prove that $\mathbb{M}$ contains exactly $z$ simple cycles.

The number of simple cycles is not influenced by the transition labels of $\mathbb{M}$. Thus, it suffices to consider the underlying graph, as reflected through the variables $\text{EDGE}(w,w')$ for all $w,w'\in W$ (Constraint 1). We use the order

that is induced by $W$ for the iterative selection of roots of the witness trees $t_{i,r_i}$ such that the roots uniquely identify the tree (Constraint 2). Starting at some root $r$, every witness tree then expands along vertices that are larger than $r$ according to the edge relation of $G$. Therefore, every edge that does not lead back to $r$ and does not visit a vertex that already has been visited above in the tree must be colored blue (Constraint 3). On the other hand, if the edge leads back to the root it must be colored red (Constraint 4). Moreover, Constraints 5 and 6 assure that blue and red edges indeed match with the edge relation of $G$. Note that we only consider edges that are part of an SCC of the corresponding sub-graph $G_r$, which are restricted to vertices larger than $r$. According to Lemma 6, it is guaranteed that the variables $\text{SCC}_r(w)$ correctly identify the SCCs of the sub-grap $G_r$.

Even a single SCC can contain multiple witness-trees. Therefore, they are distinguished through the identifier $\text{WTREE}(s)$ for every $s \in S$. This identifier validates the selection of blue and red edges, because it stays consistent along every witness-tree $t_{i,r_i}$ (Constraints 7 and 8).

It still needs to be guaranteed that we indeed guess trees having the required shape of a witness-tree. To this end, Constraint 9 guarantees that every inner node and leaf of the tree has a unique parent. Constraint 10 guarantees that no vertex appears more than once on every path from a root to a leaf. It does so by tagging every vertex on the path with all vertices above in the tree that already have been visited. The tagging is initialized at the root of the tree (Constraint 11), through stating that no vertex is visited initially.

Together, all aforementioned constraints guarantee the correct identification of the witness-trees of $G$, where every red edge of a tree identifies a simple cycle of $G$. Thus, it only remains to limit the number of red edges by the given bound $z$. To this end, we utilize a list of length $z$ containing unique identifiers for each red edge, as part of the witness-trees. The list must be strictly ordered such that no entry appears more than once (Constraint 12). Furthermore, Constraint 13 guarantees that each red edge is validated through a corresponding entry in the list. Thus, $\mathtt{C}(\mathbb{M})$ indeed is bounded by $z$.

"$\Leftarrow$": Assume that there is a Mealy machine $\mathbb{M}$ with $|\mathbb{M}| = n$ and $\mathtt{C}(\mathbb{M}) = z$ that is accepted by $\mathfrak{A}$. We construct the corresponding graph $G$, the SCCs of $G$, and the witness-trees $t_{i,r_i}$ according to Algorithm 2. It is easy to verify that the results are witnesses for all constraints of $\Psi_{CS}(\mathfrak{A}, n, z)$. □

Let $\mathfrak{A} = (2^{\mathcal{I}\cup\mathcal{O}}, Q, q_I, \Delta_{\mathfrak{A}}, \text{coBÜCHI}(R))$ be a universal co-Büchi automaton with $|Q| = m$ and $\max\{|\Delta_{\mathfrak{A}}(q, q')| \mid q, q' \in Q\} = d$. The formula $\Psi_{CS}(\mathfrak{A}, n, z)$ consists of $j \in O(n^3 + n^2(z^2 + 2^{|\mathcal{I}|}) + n|\mathcal{O}| + nm\log(nm))$ many variables and $|\Psi_{CS}(\mathfrak{A}, n, z)| \in O(n^3 + n^2(z^2 + dm2^{|\mathcal{I}|}))$.

# 3 Compact Implementation Models

As we have seen, the reduction of the synthesis problem to a constraint system provides many advantages. The constraint system cannot only be utilized for finding some implementation that is accepted by the universal co-Büchi automaton, but also allows for imposing additional constraints that guarantee output sensitive properties. The constraints, we considered so far, bound the size of the Mealy machine implementing the specification or the number of simple cycles of the graph structure, as induced by the Mealy machine. In general, however, the model of Mealy machines is not the first choice that comes along, if it comes to an implementation model that is required in practice. The model more serves as a theoretical baseline in terms of semantics defining the system behavior. For practical applications, on the other hand, representation models such as circuits, tree-shaped programs, or register machines are preferred. However, with respect to the synthesis output, these models introduce different representation characteristics than their underlying flat counterpart of a Mealy machine.

## 3.1 Bounded Circuits

Circuits, for example, focus on a distributed evaluation model that matches the physical realization of most computation hardware today. In a circuit, computations are broken apart into parallel evaluations of AND gates and negations. State is realized through a set of latches, each one holding exactly one bit. The overall implementation is composed out of these components reading from Boolean input streams and producing Boolean output streams. Semantically, circuits are equivalent to Mealy machines in the sense that the cross-product of all latches defines the Mealy state. However, in terms of representation, they can be exponentially more succinct.

Rembember that in a Mealy machine internal state is given explicitly, since every possible configuration of the system results in distinguishable state. Similarly, every possible state update and production of output is bundled into single transitions and only fanned out according to the possible inputs. Thus, while giving us a simple, but expressive, mathematical model, Mealy machines are still limited when it comes to applications that need to handle a huge amount of internal state. Circuits provide a more succinct model that distributes state to latches, which are manipulate via Boolean operations. The latches hold state, as they delay Boolean inputs for a moment and, thus, allow to pass information over time.

**Definition 21.** A *circuit* $\mathbb{C} = (\mathcal{I}, \mathcal{O}, L, \gamma)$ is a tuple, where

- $\mathcal{I}$ is the set of Boolean input streams,
- $\mathcal{O}$ is the set of Boolean outputs streams,
- $L$ is the set of latches, and
- $\gamma\colon (\mathcal{O} \cup L) \to \mathbb{B}^*(\mathcal{I} \cup L)$ is the gate function that connects each circuit output and latch input to a Boolean combination of circuit inputs and latch outputs.

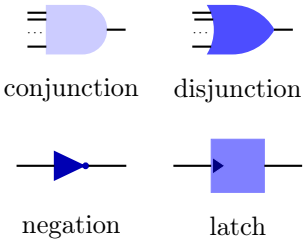
The size of a circuit is defined to be $|\gamma|$. The latches of the circuit serve as inputs and outputs at the same time. Every value that is written to a latch is delayed till the next time step. At the initial point of time, every latch outputs *false*. Since every latch holds only a single Boolean value at every point in time, the overall amount of different configurations is exponential in the number of latches.

Every circuit $\mathbb{C}$ can be transformed into an equivalent Mealy machine $\mathbb{M}_\mathbb{C}$. We identify the input alphabet $\Sigma_I$ of the Mealy machine with $2^\mathcal{I}$ and $\Sigma_O$ with $2^\mathcal{O}$, since multiple inputs are served at the same time and similarly multiple outputs are produced,

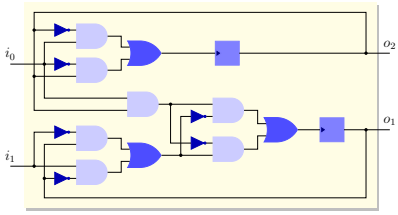
**Construction 4.** Every circuit $\mathbb{C} = (\mathcal{I}, \mathcal{O}, L, \gamma)$ corresponds to an equivalent Mealy machine $\mathbb{M}_\mathbb{C} = (2^\mathcal{I}, 2^\mathcal{O}, M, m_I, \delta_\mathbb{M}, \ell)$, where

- $M = \mathcal{B}^{[L]}$ is the set of functions assigning latches to Boolean values,
- $m_I \in M$ is the function that initiates each latch with *false*, *i.e.*, $\forall x \in L.\ m_I(x) = \mathit{false}$,
- $\delta_\mathbb{M}$ is the transition function that updates each latch according to the gate function, *i.e.*, $\forall m \in M.\ \forall \upsilon \in 2^\mathcal{I}.\ \delta(m, \upsilon) = m'$ such that $m'(x) = (\upsilon \cup \{y \in L \mid m(y)\} \vDash \gamma(x))$,
- $\ell$ is the output function that updates each output according to the gate function of the circuit, *i.e.*,

$$\forall m \in M.\ \forall \upsilon \in 2^\mathcal{I}.\ \ell(m, \upsilon) = \{o \in \mathcal{O} \mid \upsilon \cup \{y \in L \mid m(y)\} \vDash \gamma(o)\}$$




(a) Graphical representation of latches, conjunction, disjunction and negation.




(b) illustration of a two-bit integrator summing two-bit inputs over time

Figure 30: The individual components of a circuit and a circuit example.

Construction 4 defines the semantics of a circuit $\mathbb{C}$ and determines it's word language $\mathcal{L}(\mathbb{C})$ to be $\mathcal{L}(\mathbb{M}_{\mathbb{C}})$. Circuits are illustrated using a graphical representation. The individual components are depicted in Figure 30a. A complete circuit consists of a connected set of such components, connecting inputs to outputs. If a global input or a component output is shared among multiple inputs, then the connection point is marked using a black dot. Figure 30b presents an example circuit of a two bit integrator, that sums up two-bit input values over time. Note that conjunction and disjunction are allowed to have multiple inputs. This is a valid simplification, since both operations are associative and commutative according to their semantics.

Circuits consider an implementation model that can be exponentially more succinct than Mealy machines. Therefore, they distribute the Mealy state over the cross-product of multiple latches, each one holding only a single bit. Hence, with respect to bounded synthesis, the state bound is turned into a latch bound, which grows only logarithmically as fast as the bound on the number of Mealy states.

However, as part of their distributed architecture, circuits also introduce another natural bound: the number of gates that are used as part of the circuit representation. The corresponding bound measures the complexity of computing the output valuations from the given input and latch valuations in terms of the number of Boolean operations. We only consider conjunctions and disjunctions to be captured as part of this bound, since the number of negations always can be kept linear in this number. Finally, note that compared to Mealy machines, the complexity of computing outputs from inputs at a time is completely hidden as part of the transition relation.

As a consequence, if circuits are targeted as the underlying implementation model, output sensitive synthesis approaches must consider two bounds in

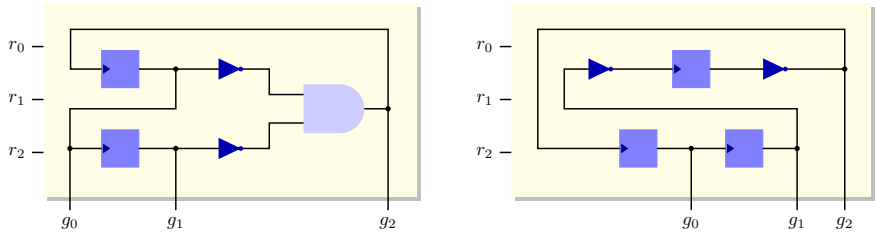



Figure 31: The tradeoff between minimizing latches and gates. Both circuits implement a simple arbiter for three clients. The left one is minimal in the number latches, while the right one is minimal in the number of gates.

terms of the possible search space metrics. However, two bounds always raise an immediate question: which of them should have higher importance with respect to a minimal solution. As we have seen already for the Bounded Cycle Synthesis approach, it is not always possible to minimize multiple bounds simultaneously. A similar result arises for bounding the number of latches and gates. To this end, consider the circuits of Figure 31, which implement the specification of a simple arbiter with three clients. The circuit on the left is minimal in the number of latches, but requires a single conjunction in order to archive the required behavior. On the other hand, the circuit on the right needs no conjunction at all, but at the cost of an additional latch. The comparison shows that the preference of minimality with respect to the number of gates and latches must be resolved by the user, since there is no natural precedence in general.

We continue with the encoding of circuits into SAT, where we leverage some simplifications of the representation of the gate function $\gamma$. In general, the gate structure that is used to calculate outputs and latch inputs utilizes all kinds of Boolean operations, i.e., conjunction, disjunction, and negation. In terms of simplification, however, we can assume that disjunctions are replaced through a combination of conjunctions and negations according to the equivalence, as given with De-Morgan's law. Furthermore, we can assume that conjunctive gates are always restricted to a binary set of inputs, since conjunctions with more inputs always can be transferred into a chain of binary gates, accordingly. Moreover, similar to our graphical representation, gates can be shared among multiple evaluations if they correspond to the same Boolean structure. Finally, negations are coupled with the latch and conjunctive gate representations such that multiple variants of conjunctions

and latches are introduced, according to the possible combinations of placing a single negation afterwards or not. In both cases, we receive two possible variants: the standard latch or conjunction representation, without any negations, and a complemented variant, where the output is negated after being read from the component.

All of these simplifications are standard, as they are already used by AIGER format [12]. The format is standardized and currently used for example by the hardware model checking competition HWMCC [20] and the synthesis competition SYNTCOMP [70] for the representations of circuits. Many choices of our encoding have been inspired by the AIGER format design intentions, correspondingly.

According to the aforementioned simplifications, we consider the gate function $\gamma$ to be represented as a graph, where vertices indicate binary conjunctions or latches and edges connect them according to their evaluation in $\gamma$. Note that every cycle of the graph structure must contain at least one latch according to the definition of $\gamma$. Inputs and outputs serve as sources and sinks for the connections, where *true* is a special source that always provides the corresponding constant. Similar to vertices, sources can appear in two variants: equipped with or without a negation afterwards. Moreover, with respect to the aforementioned variants, negations are part of the conjunction and latch vertices.

Formally, we partition vertices into latches $L$ and gates $G$. For the connection of vertices with inputs and outputs we utilize the concept of *endpoints*. Endpoints describe the possible connection capabilities of inputs, outputs, and vertices according to the represented elements. We denote the set of all endpoints by $E_P = E_P^I \uplus E_P^O$, which is partitioned into input endpoints $E_P^I$ and output endpoints $E_P^O$. To this end, every input and vertex of the structure offers two output endpoints. One for the positive provided value and one for the complementary result. Similarly, every output and latch provides one input endpoint for passing a value to the component. Correspondingly, conjunctions have two input endpoints: one for each input. Given the number of inputs, outputs, latches, and gates, the overall number of endpoints $|E_P|$, thus, is determined by

$$|E_P| = |E_P^I| + |E_P^O| = 2 + 2|\mathcal{I}| + 3|L| + 4|G| + |\mathcal{O}|,$$

where the special source for the constant *true* is also included. For the sake of readability, we use $E_P^I = \mathcal{O} \uplus L \uplus (G \times \{1, 2\})$ and $E_P^O = \mathcal{B} \uplus ((\mathcal{I} \uplus L \uplus G) \times \mathcal{B})$, which not only matches the above calculation, but also supports an intuitive reasoning of the different endpoints. Note that we use $\mathcal{B}$ to distinguish the complementary endpoints from the positive ones.

For our encoding of the circuit as part of a bounded approach, we introduce the bounds $n \in \mathbb{N}$ and $z \in \mathbb{N}$ for the number of latches $L$ and gates $G$, respectively. To this end, we assume that $L = [n]$ and $G = [z]$, such that the circuit itself is representable by a mapping $c\colon E_P^I \to E_P^O$. Note that under the knowledge of $\mathcal{I}$, $\mathcal{O}$, $n$ and $z$, a mapping from $E_P^I$ to $E_P^O$ indeed uniquely determines the structure of a circuit $\mathbb{C}$, as long as there is latch free cycle induced by $c$.

**Construction 5.** For given inputs $\mathcal{I}$, outputs $\mathcal{O}$, and $n, z \in \mathbb{N}$, every mapping $c\colon E_P^I \to E_P^O$ induces a circuit $\mathbb{C} = (\mathcal{I}, \mathcal{O}, [n], c^*)$ with:

$$c^*(x) = \begin{cases} i \leftrightarrow b & \text{if } c(x) \in (\mathcal{I} \times \mathcal{B}) \text{ with } c(x) = (i, b) \\ l \leftrightarrow b & \text{if } c(x) \in ([n] \times \mathcal{B}) \text{ with } c(x) = (l, b) \\ (c^*(g,1) \wedge c^*(g,2)) \leftrightarrow b & \text{if } c(x) \in ([z] \times \mathcal{B}) \text{ with } c(x) = (g, b) \\ c(x) & \text{otherwise} \end{cases}$$

Note that $c^*$ is only well-defined if every cycle induced by $c$ contains at least one latch. Let $L = [n]$ and $G = [z]$ with $z, n \in \mathbb{N}$ and a universal co-Büchi automaton $\mathfrak{A}_\varphi = (2^{\mathcal{I} \cup \mathcal{O}}, Q, q_I, \Delta_\mathfrak{A}, \textsc{coBüchi}(R))$ with $k = |R|$ be given. We introduce the following variables to be used in our encoding:

- $\textsc{connect}(e_I, k)$ for all $e_I \in E_P^I$, and $0 \leq k \leq \log |E_P^O|$ denoting the mapping $c\colon E_P^I \to E_P^O$. To this end, we identify every output endpoint with a unique number to be encoded with logarithmically many bits.
- $\textsc{eprank}(e_I, j)$ for all $e_I \in E_P^I$, $0 \leq j \leq \log |E_P|$ used for a ranking of the endpoints to ensure that there is no latch-free cycle.
- $\textsc{val}_\theta(e_I)$ for all $\theta \in 2^{L \cup \mathcal{I}}$ and $e_I \in E_P^I$ representing the Boolean valuation of every endpoint with respect to a given input evaluation and the bits stored by the latches.
- $\textsc{rgstate}(m, q)$ for all $m \in 2^L$ and $q \in Q$ denoting the reachable vertices of the run graph under the reachable latch evaluations.
- $\textsc{ranking}(m, q, i)$ for all $m \in 2^L$, $q \in Q$ and $0 \leq i \leq \log(k \cdot 2^n)$ denoting the ranking of the run graph.

We use the variables $\textsc{connect}(e_I, k)$ to guess a mapping $c$ that represents a circuit $\mathbb{C}$ according to Construction 5, as long as $c$ does not contain latch-free cycles. The latter is guaranteed through the ranking induced by the variables

$\textsc{eprank}(e_I, j)$. The circuit must be evaluated under every possible input and latch configuration for the construction of a run graph that witnesses the acceptance of $\mathfrak{A}_\varphi$, similar to the standard bounded synthesis encoding. The corresponding evaluation is propagated along the endpoints through the variables $\textsc{val}_\theta(e_I)$. The remaining variables are used to construct the corresponding run-graph. The only difference to the standard bounded synthesis encoding is that instead of using the states of the Mealy machine directly, they are induced through the cross-product of the individual bits of the latches.

Our encoding is split into two sub-formulas. The first part $\Psi_{VAL}(n, z, \theta, e_I)$ encodes the deterministic evaluation of the mapping $c$ according to its semantics $c^*$ as presented in Construction 5. The respective formulas consist of the following constraints for given $\theta \in 2^{L \cup \mathcal{I}}$ and $e_I \in E_P^I$. We use the mapping $f_{E_P^O} : E_P^O \to [|E_P^O|]$ to denote the aforementioned index mapping for input endpoints $E_P^O$:

1. Evaluation of input endpoints of constants:

$$\bigwedge_{b \in \mathcal{B}} \textsc{connect}(e_I) = f_{E_P^O}(b) \;\rightarrow \big(b \leftrightarrow \textsc{val}_\theta(e_I)\big)$$

2. Evaluation of input endpoints of latches and inputs:

$$\bigwedge_{\substack{X \in \{L, \mathcal{I}\} \\ b \in \mathcal{B}, x \subseteq X}} \textsc{connect}(e_I) = f_{E_P^O}((x, b)) \rightarrow \big((x \in \theta \cap X \leftrightarrow b) \leftrightarrow \textsc{val}_\theta(e_I)\big)$$

3. Evaluation of input endpoints of gates:

$$\bigwedge_{b \in \mathcal{B}, g \in G} \textsc{connect}(e_I) = f_{E_P^O}((g, b)) \;\rightarrow \big(((\textsc{val}_\theta((g,1)) \wedge \textsc{val}_\theta((g,2))) \leftrightarrow b) \leftrightarrow \textsc{val}_\theta(e_I)\big)$$

The second part $\Psi_{MC}(\mathfrak{A}, n, z)$ guesses the mapping $c$, the corresponding endpoint ranking, and the run graph ranking including the reachability tags, as already familiar from the standard bounded synthesis encoding.

4. Connection mappings and endpoint rankings are bounded:

$$\bigwedge_{e_I \in E_P^I} \textsc{connect}(e_I) < |E_P^O| \;\wedge\; \textsc{eprank}(e_I) < |E_P|$$

5. The endpoint annotation strictly increases over conjunctions:

$$\bigwedge_{\substack{e_I \in (G \times \{1,2\}) \\ (g,b) \in (G \times \mathcal{B})}} \textsc{connect}(e_I) = f_{E_P^O}((g,b)) \;\rightarrow\; \bigwedge_{x \in \{1,2\}} \textsc{eprank}((g,x)) < \textsc{eprank}(e_I)$$

6. The initial state $(\emptyset, q_I)$ is reachable and the run graph ranking is bounded:

$$\textsc{rgstate}(\emptyset, q_I) \;\wedge \bigwedge_{m \in 2^L,\, q \in Q} \textsc{ranking}(m,q) < k \cdot 2^n$$

7. Each ranking of a vertex of the run graph bounds the number of visited accepting vertices, not counting the current vertex itself:

$$\begin{aligned}
&\bigwedge_{m \in 2^L,\, q \in Q} \textsc{rgstate}(m,q) \;\rightarrow \\
&\quad \bigwedge_{\nu \in 2^{\mathcal{I}}, q' \in Q} \Delta_{\mathfrak{A}}(q,q')[\nu \mapsto \mathit{true}, (\mathcal{I} \setminus \nu) \mapsto \mathit{false}][o \mapsto \textsc{val}_{(m \cup \nu)}(o)] \;\rightarrow \\
&\qquad \bigwedge_{m' \in 2^L} \Big( \bigwedge_{l \in m'} \textsc{val}_{(m \cup \nu)}(l) \;\wedge \bigwedge_{l \in L \setminus m'} \neg \textsc{val}_{(m \cup \nu)}(l) \Big) \\
&\qquad\quad \textsc{rgstate}(m',q') \;\wedge\; \textsc{ranking}(m,q) \prec_q \textsc{ranking}(m',q')
\end{aligned}$$

Through the composition of both pieces, we finally receive the bounded circuit encoding $\Psi_{CIR}(\mathfrak{A}, n, z)$:

$$\Psi_{CIR}(\mathfrak{A}, n, z) \;:=\; \Psi_{MC}(\mathfrak{A}, n, z) \wedge \bigwedge_{\theta \in 2^{L \cup \mathcal{I}},\, e_I \in E_P^I} \Psi_{VAL}(n, z, \theta, e_I)$$

**Theorem 18.** *For bounds $n, z \in \mathbb{N}$ and a universal co-Büchi automaton $\mathfrak{A}$, the formula $\Psi_{CIR}(\mathfrak{A}, n, z)$ is satisfiable if and only if there is a circuit $\mathbb{C}$ consisting of $n$ latches and $z$ gates such that $\mathcal{L}(\mathbb{C}) \subseteq \mathcal{L}(\mathfrak{A})$.*

*Proof.* "⇒": Assume that $\Psi_{CIR}(\mathfrak{A}, n, z)$ is satisfiable, then the set of variables $\textsc{connect}(e_I, j)$ induce a mapping from $E_P^I$ to $[|E_P^O|]$ due to Constraint 4, which, given some chosen bijection $f_{E_P^O} : E_P^O \to [|E_P^O|]$, is equivalent to a mapping $c\colon E_P^I \to E_P^O$. The existence of an endpoint $\textsc{eprank}(e_I)$, which strictly

increases along conjunctions (Constraint 5), witnesses that every cycle induced by $c$ must contain at least one latch. Hence, according to Construction 5 the mapping induces a circuit $\mathbb{C}$ with $n$ latches and $z$ gates.

It remains to proof that $\mathbb{C}$ indeed is accepted by $\mathfrak{A}$. To this end, the variables $\text{VAL}_\theta(e_I)$ assign every input endpoint $e_I$ a Boolean value under the scope of some latch assignment $\theta \cap L$ and some inputs $\theta \cap I$. If the input endpoint connects to the constant source *true*, then it is *true*, if selecting the positive port, and *false*, otherwise (Constraint 1). If the input endpoint connects to a latch output or a circuit input, then it reflects the value of the latch or the input according to $\theta$, while it is negated, if connecting to the complementary port (Constraint 2). Finally, if the input endpoint connects to the output of a conjunction, then it holds the same value as the evaluation of the conjunction on the respective input endpoints taking the also selected port into account (Constraint 3). A simple induction along the gate elements of the structure of $\mathbb{C}$ shows that the input endpoints of latches and the circuit outputs correctly reflect the evaluation of $\mathbb{C}$ according to the induced gate function $c^*$.

The remaining Constraints 6 and 7 ensure that $\mathbb{C}$ indeed is accepted by $\mathfrak{A}$ using the same arguments as for proof of Theorem 13. Note that instead of the variables $\text{TRANS}(m, \nu)$ and $\text{LABEL}(m, \nu, o)$ the cross-product of the latch bits $\text{VAL}_{w \cup \nu}(l)$ and the circuit outputs $\text{VAL}_{m \cup \nu}(o)$ under the current inputs and latch evaluation $\theta$ are used.

"$\Leftarrow$": Assume that there is a circuit $\mathbb{C}$ with $n$ latches and $z$ gates that is accepted by $\mathfrak{A}$. Then by starting with zero and increasing the annotation by one, whenever passing a gate, it is easy to construct a ranking that satisfies Constraints 4 and 5. Furthermore, the valuation $\text{VAL}_\theta(e_I)$ can be selected according to the matching semantics of latches and conjunctions such that it is straightforward to satisfy $\Psi_{VAL}(n, z, \theta, e_I)$ for every possible $\theta$. Finally, since $\mathbb{C}$ is accepted by $\mathfrak{A}$ the run graph of $\mathfrak{A}$ and the corresponding latch evaluations must have only finitely many visits to rejecting states. Therefore, it must be possible to select a corresponding ranking $\text{RANKING}(m, q)$, which satisfies the given constraints. □

Let $\mathfrak{A} = (2^{\mathcal{I} \cup \mathcal{O}}, Q, q_I, \Delta_\mathfrak{A}, \text{coBÜCHI}(R))$ be a universal co-Büchi automaton with $|Q| = m$ and $\max\{|\Delta_\mathfrak{A}(q, q')| \mid q, q' \in Q\} = d$. The formula $\Psi_{CIR}(\mathfrak{A}, n, z)$ consists of

$$j \in O\big(|\mathcal{O}| \cdot 2^{|\mathcal{I}|} + z \cdot 2^{|\mathcal{I}|} + n \cdot 2^n \cdot m \log m\big)$$

many variables and for $t = |\mathcal{O}| + |\mathcal{I}| + n + z$ has size:

$$|\Psi_{CS}(\mathfrak{A}, n, z)| \in O\big(n \cdot m^2 \cdot 2^{2n+|\mathcal{I}|} + t^2 \log t \cdot 2^{n+|\mathcal{I}|}\big).$$

## 3.2 Bounded Register Machines

Circuits are a canonical choice for an exponentially more succinct representation of Mealy machines. However, in practice multi-purpose devices need to be re-configurable even after production, which is hard and costly to archive using a circuit. Instead, embedded devices with a central processing unit are used, which manipulates the internal state using configurable sequences of instructions. The most basic such model is reflected by the model of register machines, which operate on a set of Boolean registers while reading Boolean inputs from the environment and producing Boolean outputs.

Register machines utilize finite sets of instructions $\mathbb{I}$ and work registers $r_w$ for holding intermediate results of a computation. In practice, the exact set of instructions often widely varies and depends on the final application. Therefore, we restrict ourselves to a minimal set of instruction here, which still suffices to model any other instruction set in terms of expressivity.

**Definition 22.** Let inputs $\mathcal{I}$, outputs $\mathcal{O}$, and a finite set of registers $R$ be given. The set of *instructions* $\mathbb{I}_{\mathcal{I},\mathcal{O},R}$ consists of the following elements:

| | |
|---|---|
| `CONST` $b$ | for $b \in \mathcal{B}$, setting $r_w$ to $b$ |
| `NOT` | negating $r_w$ |
| `AND` $x$ | for $x \in R \cup \mathcal{I}$, storing the conjunction of $r_w$ and $x$ to $r_w$ |
| `READ` $x$ | for $x \in R \cup \mathcal{I}$, storing $x$ to $r_w$ |
| `WRITE` $x$ | for $x \in R \cup \mathcal{O}$, storing $r_w$ to $x$ |
| `JMP` $j$ | for $j \in \mathbb{N}$, jumping to the instruction at position $j$ |
| `CJMP` $j$ | for $j \in \mathbb{N}$, jumping to position $j$, if and only if $r_w = \mathit{true}$ |
| `NEXT` | proceeding to the next step in time |

The model of a register machine uses finite sets of inputs, outputs, and registers and executes an instruction sequence of finite length, not including jump instructions that point to positions outside this sequence.

**Definition 23.** A *register machine* $\mathbb{R} = (\mathcal{I}, \mathcal{O}, R, \varrho)$ is a tuple, where

- $\mathcal{I}$ is the set of inputs,
- $\mathcal{O}$ is the set of outputs,
- $R$ is a set of registers, and
- $\varrho \in \mathbb{I}^+_{\mathcal{I},\mathcal{O},R}$ is a finite sequence of instructions.

To obtain the semantics of a register machine $\mathbb{R}$, we cover all it's possible executions by a corresponding Mealy machine $\mathbb{M}_\mathbb{R}$, where we assume that $\mathbb{R}$ is reactive, *i.e.*, all possible infinite executions infinitely often execute some instruction `NEXT`. Registers are set to *false* initially.

**Construction 6.** Every register machine $\mathbb{R} = (\mathcal{I}, \mathcal{O}, R, \varrho)$ corresponds to an equivalent Mealy machine $\mathbb{M}_\mathbb{R} = (2^\mathcal{I}, 2^\mathcal{O}, M, m_I, \delta_\mathbb{M}, \ell)$, where

- $M = \mathcal{B} \times 2^{R \cup \mathcal{O}} \times [|\varrho|]$,
- $m_I = (\mathit{false}, \emptyset, 0)$,
- $\delta_\mathbb{M}((r_w, X, i), \nu) = \begin{cases} (r_w, X, i\texttt{++}) & \text{if } \varrho_i \equiv \texttt{NEXT} \\ \delta_\mathbb{M}(\xi((r_w, X, i), \nu, \varrho_i), \nu) & \text{otherwise, and} \end{cases}$
- $\ell((m, \nu) = pr_1(\delta_\mathbb{M}(m, \nu)) \cap \mathcal{O}$,

using $\xi \colon M \times 2^\mathcal{I} \times \mathbb{I}_{\mathcal{I},\mathcal{O},R} \to M$ with

$$\xi((r_w, X, i), \nu, t) = \begin{cases} (b, X, i\texttt{++}) & \text{if } t \equiv \texttt{CONST } b \\ (\neg r_w, X, i\texttt{++}) & \text{if } t \equiv \texttt{NOT} \\ (r_w \wedge x, X, i\texttt{++}) & \text{if } t \equiv \texttt{AND } x \\ (x \in X \cup \nu, X, i\texttt{++}) & \text{if } t \equiv \texttt{READ } x \\ (r_w, X \cup \{x\}, i\texttt{++}) & \text{if } t \equiv \texttt{WRITE } x \\ (r_w, X, j) & \text{if } t \equiv \texttt{JMP } j \\ (r_w, X, j) & \text{if } t \equiv \texttt{CJMP } j \wedge r_w \\ (r_w, X, i\texttt{++}) & \text{if } t \equiv \texttt{CJMP } j \wedge \neg r_w \\ (r_w, X, i) & \text{if } t \equiv \texttt{NEXT} \end{cases}$$

where $i\texttt{++} \equiv (i + 1) \bmod |\varrho|$. Note that we require that $\mathbb{R}$ is reactive.

The semantics of a register machine $\mathbb{R}$ are induced by the semantics of $\mathbb{M}_\mathbb{R}$. Similarly, the language $\mathcal{L}(\mathbb{R})$ of a register machine is defined to be $\mathcal{L}(\mathbb{M}_\mathbb{R})$.

An example of a register machine, which delays some input $\mathcal{I} = \{\texttt{i}\}$ two time steps until it is output via $\mathcal{O} = \{\texttt{o}\}$ using registers $R = \{\texttt{r}_0, \texttt{r}_1\}$, is given by $\mathbb{R}^e = (\mathcal{I}, \mathcal{O}, R, \varrho)$, with $\varrho$ consisting of

| | | |
|---|---|---|
| $\varrho_0 \equiv \texttt{READ i}$ | $\varrho_4 \equiv \texttt{WRITE o}$ | $\varrho_8 \equiv \texttt{READ r}_0$ |
| $\varrho_1 \equiv \texttt{WRITE r}_0$ | $\varrho_5 \equiv \texttt{READ i}$ | $\varrho_9 \equiv \texttt{WRITE o}$ |
| $\varrho_2 \equiv \texttt{NEXT}$ | $\varrho_6 \equiv \texttt{WRITE r}_1$ | |
| $\varrho_3 \equiv \texttt{READ r}_1$ | $\varrho_7 \equiv \texttt{NEXT}$ | |

Note that the program restarts at $\varrho_0$ after the execution of the last instruction.

Every output sensitive synthesis approach that targets register machines as the underlying implementation representation must at least impose a bound on the number of registers $R$ and on the length of the instruction sequence $\varrho$ in order to restrict the corresponding search space to a finite range. We obtain two bounds $n \in \mathbb{N}$ and $z \in \mathbb{N}^+$, bounding the number of registers and the number of instructions, respectively. Furthermore, the sets of inputs $\mathcal{I}$ and outputs $\mathcal{O}$ are known, as they are fixed by the specification. Thus, the number of possible instructions is bounded by $|\mathbb{I}_{\mathcal{I},\mathcal{O},R}| = 3n + 2z + 2|\mathcal{I}| + |\mathcal{O}| + 4$.

Nevertheless, having well-defined ranges of the underlying search space provides only half of the requirements for our encoding. The witnessed register machine also must be accepted by the specification automation. Therefore, we evaluate the guessed instruction sequence under all possible inputs of the environment explicitly inducing the Mealy machine, as derived by Construction 6. This underlying Mealy machine then can be verified against the specification automaton using the standard bounded synthesis encoding.

We fix the set of registers to be $R = [n]$ and fix $I = [z]$ representing the ordered list of instructions. Moreover, let $M = \mathcal{B} \times 2^{R \cup \mathcal{O}} \times I$ be the set of induced Mealy states, as introduced by Construction 6. The specification is given by a universal co-Büchi automaton $\mathfrak{A} = (2^{\mathcal{I} \cup \mathcal{O}}, Q, q_I, \Delta_{\mathfrak{A}}, \text{coBÜCHI}(R))$ with $k = |R|$ rejecting states. We introduce the following variables to be used by our encoding:

- $\text{INSTR}(i, j)$ for all $i \in I$ and $0 \leq j \leq \log |\mathbb{I}_{\mathcal{I},\mathcal{O},R}|$ denoting the selected instructions at positions $i$ of $\varrho$. We assume the existence of some given bijection $f_{\mathbb{I}} \colon \mathbb{I}_{\mathcal{I},\mathcal{O},R} \mapsto [|\mathbb{I}|_{\mathcal{I},\mathcal{O},R}]$ that uniquely indexes every instruction.
- $\text{TARGET}(m, \nu, j)$ for all $m \in M$, $\nu \in 2^{\mathcal{I}}$, and $0 \leq j \leq \log |M|$ mapping each configuration to the next interaction with the environment, as indicated through a `NEXT` instruction. We use some given bijection $f_M \colon M \to [|M|]$ to index the states of $M$.
- $\text{EVALO}(m, \nu, o)$ for all $m \in M$, $\nu \in 2^{\mathcal{I}}$, and $o \in \mathcal{O}$ reflecting the selected outputs at every environment interaction.
- $\text{EVALRANK}(m, \nu, j)$ for all $m \in M$, $\nu \in 2^{\mathcal{I}}$, and $0 \leq j \leq \log |M|$ bounding the lengths of the evaluated instruction sequences to ensure reactivity.

Additionally, the encoding inherits the $\text{RGSTATE}(m, q)$ and $\text{RANKING}(m, q, j)$ variables, as they are used by the standard bounded synthesis encoding.

The register machine is witnessed through the variables $\text{INSTR}(i, j)$ fixing the instruction sequence $\varrho$. Semantics then induce a Mealy machine $\mathbb{M}_{\mathbb{R}}$ through the step-by-step evaluation of the instructions. However, not all of

these individual steps are relevant in terms of $\mathbb{M}_{\mathbb{R}}$, since $\mathbb{M}_{\mathbb{R}}$ only captures the state between environment interactions, as it is indicated through `NEXT` instructions. Therefore, the variables $\textsc{target}(m, \nu, j)$ provide a shortcut to the next such interaction. The variables $\textsc{evalo}(m, \nu, o)$ are used to indicate the selected outputs at every state $m \in M$. Finally the $\textsc{evalrank}(m, \nu, j)$ variables introduce a ranking on the instruction evaluations between two interactions, which ensures that the program is reactive, since every evaluation must reach a `NEXT`-instruction eventually.

Given a universal co-Büchi automaton $\mathfrak{A}$ with $k$ rejecting states, a bound $n$ on the number of registers, and a bound $z$ on the number of instructions, then the Bounded Register Machine Synthesis problem is encoded via the SAT formula $\Psi_{RM}(\mathfrak{A}, n, z)$ consisting of the following constraints

1. The initial state is reachable and all rankings are bounded:

$$\textsc{rgstate}((\mathit{false}, \emptyset, z-1), q_I) \ \wedge \bigwedge_{m \in M,\, \nu \in 2^{\mathcal{I}}} \textsc{evalrank}(m, \nu) < |M| \\ \wedge \bigwedge_{m \in M,\, q \in Q} \textsc{ranking}(m, q) < |M| \cdot k$$

2. Instruction types and evaluation variables are bounded:

$$\bigwedge_{i \in I} \textsc{instr}(i) < |\mathbb{I}_{\mathcal{I},\mathcal{O},R}| \ \wedge \bigwedge_{m \in M, \nu \in 2^{\mathcal{I}}} \textsc{target}(m, \nu) < |M|$$

3. `NEXT` instructions are sinks and determine the output of the evaluation:

$$\bigwedge_{m \in M,\, \nu \in 2^{\mathcal{I}}} \Big( \textsc{instr}(pr_2(m)) = f_{\mathbb{I}}(\texttt{NEXT}) \ \rightarrow \ \textsc{target}(m, \nu) = f_M(m) \\ \wedge \bigwedge_{o \in pr_1(m) \cap \mathcal{O}} \textsc{evalo}(m, \nu, o) \ \wedge \bigwedge_{o \in \mathcal{O} \setminus pr_1(m)} \neg \textsc{evalo}(m, \nu, o) \Big)$$

4. All other instructions are evaluated according to their semantics:

$$\bigwedge_{\substack{t \in \mathbb{I}_{\mathcal{I},\mathcal{O},R},\, m \in M,\, \nu \in 2^{\mathcal{I}} \\ t \neq \texttt{NEXT}}} \Big( \textsc{instr}(pr_2(m)) = f_{\mathbb{I}}(t) \ \rightarrow \\ \textsc{evalrank}(m, \nu) < \textsc{evalrank}(\xi(m, \nu, t), \nu)) \\ \wedge \ \textsc{target}(\xi(m, \nu, t), \nu) = \textsc{target}(w, \nu) \\ \wedge \bigwedge_{o \in \mathcal{O}} \textsc{evalo}(\xi(m, \nu, t), \nu, o) \leftrightarrow \textsc{evalo}(m, \nu, o) \Big)$$

5. Each ranking of a vertex of the run graph bounds the number of visited accepting vertices, not counting the current vertex itself:

$$\bigwedge_{(r_w,X,i)\in M,\, q\in Q} \textsc{rgstate}((r_w,X,i),q) \rightarrow$$
$$\bigwedge_{\nu\in 2^{\mathcal{I}},q'\in Q} \Delta_{\mathfrak{A}}(q,q')[\nu \mapsto \mathit{true}, (\mathcal{I}\setminus\nu)\mapsto \mathit{false}] \quad [o \mapsto \textsc{evalo}((r_w,X,i\texttt{++}),\nu,o)] \rightarrow$$
$$\bigwedge_{m'\in M} \textsc{target}((r_w,X,i\texttt{++}),\nu) = m' \rightarrow$$
$$\textsc{rgstate}(m',q') \wedge \textsc{ranking}((r_w,X,i),q) \prec_q \textsc{ranking}(m',q')$$

**Theorem 19.** *For bounds $n, z \in \mathbb{N}$ and a universal co-Büchi automaton $\mathfrak{A}$, the formula $\Psi_{RM}(\mathfrak{A}, n, z)$ is satisfiable iff there is a register machine $\mathbb{R} = (\mathcal{I}, \mathcal{O}, R, \varrho)$ with $|R| = n$ and $|\varrho| = z$ such that $\mathcal{L}(\mathbb{R}) \subseteq \mathcal{L}(\mathfrak{A})$.*

*Proof.* "⇒": Assume that $\Psi_{RM}(\mathfrak{A}, n, z)$ is satisfiable. Hence, the $\textsc{instr}(i, j)$ variables witness a mapping from $I$ to $\mathbb{I}_{\mathcal{I},\mathcal{O},R}$ inducing a sequence $\varrho$ with $|\varrho| = z$ and $\varrho_i = t$ if an only if $\textsc{instr}(i) = t$ for all $0 \leq i < z$. Accordingly, if we choose $R = [n]$, then we obtain a register machine $\mathbb{R} = (\mathcal{I}, \mathcal{O}, R, \varrho)$, where $\mathcal{I}$ and $\mathcal{O}$ are taken from $\mathfrak{A}$. It remains to proof that $\mathcal{L}(\mathbb{R}) \subseteq \mathcal{L}(\mathfrak{A})$.

The register machine $\mathbb{R}$ induces a Mealy machine $\mathbb{M}_{\mathbb{R}}$ with initial state $(\mathit{false}, \emptyset, 0)$. However, the evaluation is started at the last instruction of $\varrho$ due to the shift introduced by Constraint 5 ($i$++). The shift is necessary, since the evaluation would be trapped at `NEXT` instructions otherwise. Note that the initial state is the only one that does not have to reside on a `NEXT` instructions according to our construction. Constraint 1 ensures that the initial state is reachable, from which the register machine then is evaluated according the semantics of the witnessed instruction sequence. Furthermore, together with Constraint 2, it ensures that all binary encoded variables are correctly bounded. For the evaluation of the machine, we distinguish two general cases:

If the current instruction is a `NEXT`, then $\mathbb{M}_{\mathbb{R}}$ takes a transition, for which the outputs are selected according to $\nu \in 2^{\mathcal{I}}$ and the instructions that have been evaluated since the previous `NEXT` or the initial state. The corresponding behavior is realized through Constraint 3, which enforces all $\textsc{evalo}(m, \nu, o)$ to reflect the selected outputs of $w \in M$, and that the $\textsc{target}(m, \nu)$ variables point at the current configuration.

If the current instruction is no NEXT, then $\varrho$ is evaluated until a NEXT instruction is hit. The evaluation is realized through Constraint 4, which selects the successor according to the evaluation function $\xi$ from Construction 6. The evaluation cannot proceed indefinitely, since the ranking of the successor always must be strictly greater than the current one. Thus, there is maximum on the number of evaluated instruction that are no NEXT. During the evaluation, the pointer to the target configuration $\text{TARGET}(m, \nu)$ and the selected outputs are always copied from the successor configuration.

Together, Constraint 3 and Constraint 4 ensure that the $\text{TARGET}(m, \nu)$ and $\text{EVALO}(m, \nu, o)$ variables always point to the NEXT instruction that results from evaluating $\varrho$ from the current position. The property is proven by a simple induction running backwards from the targeted NEXT instruction. Termination of this backward induction is guaranteed by the $\text{EVALRANK}(m, \nu)$ ranking.

Constraint 5 finally induces the run graph construction, similar to the standard bounded synthesis approach. The run graph is iteratively constructed from the initial vertex such that every edge corresponds to a transition of $\mathbb{R}$ and a transition of $\mathfrak{A}$. The $\text{LABEL}(m, \nu, o)$ and $\text{TRANS}(m, \nu)$ variables, as used by the standard bounded synthesis encoding, are replaced by $\text{EVALO}(m, \nu, o)$ and $\text{TARGET}(m, \nu)$, respectively, with the only difference that the instruction pointer needs to be increased in order to obtain the result of the evaluation starting at the position after the current NEXT instruction. Remember that every Mealy state $\text{TARGET}(m, \nu)$ points at the reached NEXT instruction. Hence, the instruction afterwards ($i$++) must be selected.

Thus, the induced Mealy machine $\mathbb{M}_\mathbb{R}$ indeed is correctly reflected by the variables $\text{EVALO}(m, \nu, o)$ and $\text{TARGET}(m, \nu)$. As a consequence, $\mathbb{M}_\mathbb{R}$ must be accepted by $\mathfrak{A}$ using the same arguments as used for proving Theorem 13.

"$\Leftarrow$": Now, assume there is some $\mathbb{R}$ using $n$ registers and $z$ instructions that implements the specification given by $\mathfrak{A}$. We choose the variables $\text{INSTR}(i)$ for all $0 \leq i < z$ such that they reflect $\varrho$. Almost all remaining variables then are determined deterministically according to the semantics of the instructions. The only non-deterministic choice that remains, is the evaluation ranking, which can be chosen step-wise decreasing computed backwards from the reachable NEXT instructions. The upper bound of $|M|$ therefore always is sufficient according to Constraint 2. It is easy to verify that by using the resulting variable assignments, indeed all of the given constraints are satisfied. □

Let $\mathfrak{A} = (2^{\mathcal{I} \cup \mathcal{O}}, Q, q_I, \Delta_\mathfrak{A}, \text{coBÜCHI}(R))$ be a universal co-Büchi automaton with $|Q| = m$. Let $t = n + |\mathcal{I}| + |\mathcal{O}|$. Then $|\Psi_{RM}(\mathfrak{A}, n, z)| \in O(z^2 m^2 \cdot t 2^t)$ and $\Psi_{RM}(\mathfrak{A}, n, z)$ consists of $j \in O(zm \log(zm) \cdot t 2^t)$ many variables.

## 3.3 Bounded Programs

While the model of register machines is quite close to the actual execution model of existing hardware architectures, it is only rarely used by programmers in practice directly. Instead, tree based programming languages are used for most applications, that then are translated to register machines. Due to their tree based shape, such programming language are less error prone and, thus, offer more comfort to be used by a human programmer.

We consider programs that work on Boolean variables $V_{\mathcal{B}}$, read Boolean inputs $\mathcal{I}$, and write Boolean outputs $\mathcal{O}$. Our representation is by inspired Madhusudan [100], who uses a similar model to describe tree shaped programs. From this model, we also inherit our syntax and semantics. We represent programs as finite binary trees, labeled with command labels $\mathbb{L}$ that allow to traverse the program tree for reading inputs and producing outputs.

**Definition 24.** Let finite sets $V_{\mathbb{B}}$, $\mathcal{I}$, and $\mathcal{O}$ be given. The set of *command labels* $\mathbb{L}_{\mathcal{I},\mathcal{O},V_{\mathbb{B}}}$ consists of the following elements, which can be assigned to inner positions of the tree or leaf positions, respectively.

| | | |
|---|---|---|
| **;** | sequential execution | (inner, left + right child) |
| **if** | premise | (inner, left + right child) |
| **then** | consequence | (inner, left + right child |
| **while** | conditional loop | (inner, left + right child) |
| **and** | conjunction | (inner, left + right child) |
| **or** | disjunction | (inner, left + right child) |
| **not** | negation | (inner, single left child) |
| $x$ **:=** | assignment to $x \in V_{\mathbb{B}} \cup \mathcal{O}$ | (inner, single left child) |
| $x$ | variable value $x \in V_{\mathbb{B}} \cup \mathcal{I}$ | (leaf) |
| $b$ | constant $b \in \mathcal{B}$ | (leaf) |
| **skip** | effect-less command | (leaf) |
| **inout** | input/output interaction | (leaf) |

Reactive Programs are represented as binary trees. However, for the sake of readability, we represent them over directions $\Delta = \{\swarrow, \searrow\}$, moving either to the left sub-tree via $\swarrow$ or the right one via $\searrow$.

**Definition 25.** *Reactive programs* are $\mathbb{L}_{\mathcal{I},\mathcal{O},V_\mathbb{B}}$-labeled $\Delta$-directed trees

$$\mathbb{P}\colon \Delta^* \rightharpoonup \mathbb{L}_{\mathcal{I},\mathcal{O},V_\mathbb{B}}$$

with finite domain $\mathcal{D}_\Delta \subseteq \Delta^*$, which respect the labeling constraints of Definition 24. The root is never labeled with a value, a constant or a **skip** command. Furthermore, every Boolean expression, as given by sub-trees only labeled with $v$, **and**, **or**, or **not**, appears as the left child of positions labeled as assignments, premises, or conditionals. The root of the right sub-tree of every **if**-labeled position is always labeled by **then**.

Similar to register machines, we say that a program is reactive, if every execution infinitely often visits a position labeled with **inout**. Similar to the behavior of `NEXT` for register machines, **inout** is used to output the current values $\mathcal{O}$, proceeds to the next point in time, and then reads the next choices from the environment to $\mathcal{I}$. In Madhusudan's work, this operator is split into two separate operations **input** and **output**, which allows to execute both operations separately. However, for our considerations we merged them into a single operation not affecting the expressivity of the model in the first place.

The semantics of reactive programs are covered via Mealy machines $\mathbb{M}_\mathbb{P}$ that reflect all executions with respect to the choices of the environment.

**Construction 7.** Every program $\mathbb{P}\colon \Delta^* \rightharpoonup \mathbb{L}_{\mathcal{I},\mathcal{O},V_\mathbb{B}}$ with domain $\mathcal{D}_\Delta$ corresponds to a Mealy machine $\mathbb{M}_\mathbb{P} = (2^\mathcal{I}, 2^\mathcal{O}, M, m_I, \delta_\mathbb{M}, \ell)$, where

- $M = \mathcal{B} \times \{\downarrow, \nearrow, \nwarrow\} \times 2^{V_\mathbb{B} \cup \mathcal{O}} \times \mathcal{D}_\Delta$,
- $m_I = (\mathit{false}, \downarrow, \emptyset, \varepsilon)$,
- $\delta_\mathbb{M}((b, d, X, w), \nu)$
$$= \begin{cases} (\mathit{false}, \nwarrow, X, w) & \text{if } \mathbb{P}(w) = \textbf{inout} \text{ and } d = \downarrow \\ \delta_\mathbb{M}(\theta((b, d, X, w), \nu, \mathbb{P}(w)), \nu) & \text{otherwise, and} \end{cases}$$
- $\ell(m, \nu) = pr_2(\delta_\mathbb{M}(m, \nu)) \cap \mathcal{O}$,

with $\theta\colon M \times 2^\mathcal{I} \times \mathbb{L}_{\mathcal{I},\mathcal{O},V_\mathbb{B}} \to M$ being defined according to Table 4. We use the function $\circlearrowleft$ to turn the direction, *i.e.*, $\circlearrowleft(\swarrow) := \nearrow$ and $\circlearrowleft(\searrow) := \nwarrow$.

An example program $\mathbb{P}^e$ that delays an input $\mathcal{I} = \{i\}$ for two steps until it is output via $\mathcal{O} = \{o\}$ using $V_\mathbb{B} = \{x, y, z\}$ is depicted in Figure 32.

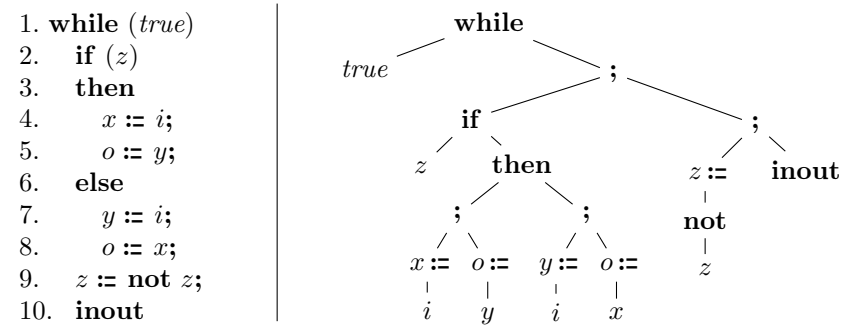



Figure 32: Example of the reactive program $\mathbb{P}^e$ on the left and it's corresponding tree representation on the right.

| $l = \mathbb{P}(w)$ | $d$ | $w$ | $b$ | $\theta((b, d, X, w), \nu, l)$ |
|---|---|---|---|---|
| **;**, **if**, **and**, **or**, **not**, $v :=$ | $\downarrow$ | | | $(false, \downarrow, X, w\swarrow)$ |
| **;**, **if** | $\nearrow$ | | | $(b, \downarrow, X, w\searrow)$ |
| **;**, **if**, **then**, **and**, **or** | $\nwarrow$ | $w'u$ | | $(b, \circlearrowleft(u), X, w')$ |
| **then** | $\downarrow$ | | *true* | $(false, \downarrow, X, w\swarrow)$ |
| **then** | $\downarrow$ | | *false* | $(false, \downarrow, X, w\searrow)$ |
| **while** | $\downarrow, \nwarrow$ | | | $(false, \downarrow, X, w\swarrow)$ |
| **while** | $\nearrow$ | | *true* | $(false, \downarrow, X, w\searrow)$ |
| **while** | $\nearrow$ | $w'u$ | *false* | $(false, \circlearrowleft(u), X, w')$ |
| **and** | $\nearrow$ | | *true* | $(false, \downarrow, X, w\searrow)$ |
| **and** | $\nearrow$ | $w'u$ | *false* | $(false, \circlearrowleft(u), X, w')$ |
| **or** | $\nearrow$ | | *false* | $(false, \downarrow, X, w\searrow)$ |
| **or** | $\nearrow$ | $w'u$ | *true* | $(true, \circlearrowleft(u), X, w')$ |
| **not** | $\nearrow$ | $w'u$ | | $(\neg b, \circlearrowleft(u), X, w')$ |
| $x :=$ | $\nearrow$ | $w'u$ | | $(false, \circlearrowleft(u), X \cup \{x\}, w')$ |
| $x$ | $\downarrow$ | $w'u$ | | $(x \in X \cup \nu, \circlearrowleft(u), X, w')$ |
| *false* | $\downarrow$ | $w'u$ | | $(false, \circlearrowleft(u), X, w')$ |
| *true* | $\downarrow$ | $w'u$ | | $(true, \circlearrowleft(u), X, w')$ |
| **skip**, **inout** | $\downarrow$ | $w'u$ | | $(false, \circlearrowleft(u), X, w')$ |
| **inout** | $\nwarrow$ | $w'u$ | | $(false, \circlearrowleft(u), X, w')$ |
| **inout** | | $\varepsilon$ | | $(false, \downarrow, X, \varepsilon)$ |

Table 4: Semantics of the operator $\theta$ as used in Construction 7.

Every reactive program introduces two natural bounds: a bound $z \in \mathbb{N}^+$ on the number of nodes $\mathcal{D}_\Delta$ of the program tree and a bound $n \in \mathbb{N}$ on the number of additionally utilized variables $V_\mathbb{B}$. Given those bounds, the number of command labels is bounded by $|\mathbb{L}_{\mathcal{I},\mathcal{O},V_\mathbb{B}}| = 12+2n+|\mathcal{O}|+|\mathcal{I}|$. Accordingly, let $N = [z]$ represent the set of program nodes and $V_\mathbb{B} = [n]$ the set of variables. Furthermore, let $M = \mathcal{B} \times \{\downarrow, \nearrow, \nwarrow\} \times 2^{V_\mathbb{B} \times \mathcal{O}} \times N$ be the set of states of the corresponding Mealy machine, as introduces in Construction 7. We introduce the following variables:

- $\textsc{cmd}(p, j)$ for all $p \in N$ and $0 \leq j \leq \log |\mathbb{L}_{\mathcal{I},\mathcal{O},V_\mathbb{B}}|$ denoting the command of the node $p$. We use some given bijection $f_\mathbb{L} \colon \mathbb{L}_{\mathcal{I},\mathcal{O},V_\mathbb{B}} \to [|\mathbb{L}_{\mathcal{I},\mathcal{O},V_\mathbb{B}}|]$ to uniquely index the corresponding set of commands.
- $\textsc{hasleft}(p)$ for all $p \in N$ indicating for every node, whether it has a left child in the program tree or not.
- $\textsc{hasright}(p)$ for all $p \in N$ indicating for every node, whether it has a right child in the program tree or not.
- $\textsc{left}(p, j)$ for all $p \in N$ and $0 \leq j \leq \log |N|$ pointing to the left child of a node, if the node has a left child.
- $\textsc{parent}(p, j)$ for all $p \in N$ and $0 \leq j \leq \log |N|$ pointing to the parent of a node. As the root is the only node with no parent, it points to itself.
- $\textsc{dest}(m, \nu, j)$ for all $m \in M$, $\nu \in 2^\mathcal{I}$, and $0 \leq j \leq \log |M|$ mapping each configuration to the next **inout** interaction. For indexing the states of $M$, we use some given bijection $f_M \colon M \to [|M|]$
- $\textsc{out}(m, \nu, o)$ for all $m \in M$, $\nu \in 2^\mathcal{I}$, and $o \in \mathcal{O}$ reflecting the selected output at every environment interaction.
- $\textsc{rk}(m, \nu, j)$ for all $m \in M$, $\nu \in 2^\mathcal{I}$, and $0 \leq j \leq \log |M|$ bounding the duration of a program evaluation until the next environment interaction.

Furthermore, the encoding again inherits the variables $\textsc{rgstate}(m, q)$ and $\textsc{ranking}(m, q, j)$ from the standard bounded synthesis encoding. The program is witnessed through the variables $\textsc{cmd}(p, j)$, $\textsc{hasleft}(p)$, $\textsc{hasright}(p)$, $\textsc{left}(p, j)$, $\textsc{parent}(p, j)$. On the one hand, the $\textsc{cmd}(p, j)$ variables assign every node a label of $\mathbb{L}_{\mathcal{I},\mathcal{O},V_\mathbb{B}}$, reflecting the program tree $\mathbb{P}$ of Construction 7. On the other hand, the tree structure, as induced by $\mathcal{D}_\Delta$, is determined through the variables $\textsc{hasleft}(p)$, $\textsc{hasright}(p)$, $\textsc{left}(p, j)$, and $\textsc{parent}(p, j)$. To this end, the $\textsc{hasleft}(p)$ and $\textsc{hasright}(p)$ variables indicate for every node,

whether it has a left or right child, respectively. If a left child exists, then the $\text{LEFT}(p)$ pointer identifies it. Similarly, the parent pointer $\text{PARENT}(p)$ identifies the parent. The only node without a parent is the root, which instead points to itself. Note that we avoid the additional introduction of a right child pointer and instead utilize the natural order of $N$, which allows for a linearization of the program tree such that the right child of a node $p \in N$ is determined to be $p + 1$, if it exists. According to this order, the root is indicated by 0 and the last entry indicates a leaf of the tree.

The variables $\text{DEST}(m, \nu, j)$, $\text{OUT}(m, \nu, o)$, and $\text{RK}(m, \nu, j)$ are used to simulate the evaluation of the program in the same fashion as the variables $\text{TARGET}(m, \nu, j)$, $\text{EVALO}(m, \nu, o)$, and $\text{EVALRANK}(m, \nu, j)$ are utilized in the bounded register encoding, respectively.

Thus, the new challenge introduced by the bounded program encoding, in comparison to the previous ones, is obtaining a valid program tree structure. We collect the corresponding requirements in the context of the formula $\Psi_{VC}(n, z)$. Note that in the previous work of Madhusudan [100], this validity check has been implemented by a non-deterministic tree automaton that guesses possible syntax violations. The automaton is joined to the complement of the specification, constructed as a two-way alternating tree automaton in a later step of the construction, which finally reduces the problem for checking the complement language to be empty. In later work [52, 53] it has been shown that the automata construction can be adapted to yield a universal co-Büchi automaton instead, such that it is bounded synthesis compliant. However, there the verification of the program structure still is bundled into the automaton construction, while our encoding only requires the original specification as a universal co-Büchi automaton. All additional requirements are solely introduced as part of the encoding.

Hence, let bounds $z$ on the program size and $n$ on the number of variables $V_{\mathbb{B}}$ be given. The formula $\Psi_{VC}(n, z)$ consists of the following constraints:

1. The root points to itself and the last entry must be a leaf:

$$\begin{aligned}&\text{PARENT}(0) = 0 \ \wedge \bigwedge_{p \in N} \text{PARENT}(p) \neq z - 1\\ &\wedge \ \neg\text{HASLEFT}(z-1) \ \wedge \ \neg\text{HASRIGHT}(z-1)\end{aligned}$$

2. If there is a right child (given by the next node index), then the parent pointer points back to the current node:

$$\bigwedge_{\substack{p \in N \\ p < z-1}} \text{HASRIGHT}(p) \ \leftrightarrow \ \text{PARENT}(p+1) = p$$

3. Childs are required to have a smaller index, in order to avoid the creation of loops. The constraint is only necessary if $z > 1$:

$$\bigwedge_{\substack{p \in N \\ p < z-2}} \textsc{hasleft}(p) \rightarrow \left( \textsc{left}(p) > p+1 \ \wedge \bigwedge_{\substack{p' \in N \\ p < p'}} \left( \textsc{left}(p) = p' \leftrightarrow \textsc{parent}(p') = p \right) \right)$$

4. The node labeling, the parent and left child pointers are bounded:

$$\bigwedge_{p \in N} \textsc{cmd}(p) < |\mathbb{L}_{\mathcal{I},\mathcal{O},V_{\mathbb{B}}}| \ \wedge \ \textsc{left}(p) < |N| \ \wedge \ \textsc{parent}(p) < |N| \ \wedge$$

5. The root can only be labeled with a restricted set of labels:

$$\bigvee_{l \in \{\mathbf{;},\mathbf{while},\mathbf{if},\mathbf{inout}\}} \textsc{cmd}(0) = f_{\mathbb{L}}(l)$$

6. The last entry is a leaf and can only be labeled with leaf labels:

$$\bigvee_{l \in V_{\mathbb{B}} \cup \mathcal{I} \cup \mathcal{B} \cup \{\mathbf{skip},\mathbf{inout}\}} \textsc{cmd}(z-1) = f_{\mathbb{L}}(l)$$

7. The second to last entry cannot have a left child and can only be labeled with a restricted set of labels. The constraint is only well-defined and necessary if $z > 1$:

$$\neg\textsc{hasleft}(z-2) \ \wedge \bigvee_{l \in \{\mathbf{;},\mathbf{while},\mathbf{if},\mathbf{then},\mathbf{and},\mathbf{or}\}} \textsc{cmd}(z-2) \neq f_{\mathbb{L}}(l)$$

8. Nodes labeled with 0-nary operations have no childs:

$$\bigwedge_{\substack{p \in N \\ p < z-1}} \left( \bigvee_{l \in \mathcal{B} \cup V_{\mathbb{B}} \cup \mathcal{I} \cup \{\mathbf{inout},\mathbf{skip}\}} \textsc{cmd}(p) = f_{\mathbb{L}}(l) \right) \leftrightarrow \left( \neg\textsc{hasleft}(p) \ \wedge \ \neg\textsc{hasright}(p) \right)$$

9. Nodes labeled with unary operations have only a right child:

$$\bigwedge_{\substack{p \in N \\ p < z-1}} \left( \textsc{cmd}(p) = f_{\mathbb{L}}(\mathbf{not}) \ \vee \bigvee_{x \in V_{\mathbb{B}} \cup \mathcal{O}} \textsc{cmd}(p) = f_{\mathbb{L}}(x \mathbf{:=}) \right) \leftrightarrow \left( \neg\textsc{hasleft}(p) \ \wedge \ \textsc{hasright}(p) \ \wedge \ \Psi_{\mathcal{B}}(p+1) \right)$$

10. Nodes labeled with binary operations have two childs:

$$\bigwedge_{\substack{p\in N\\ p<z-1}}\Bigg(\bigwedge_{l\in\{\mathbf{;},\mathbf{and},\mathbf{or},\mathbf{if},\mathbf{then},\mathbf{while}\}}\Big(\textsc{cmd}(p)=f_{\mathbb{L}}(l)\\ \rightarrow\big(\textsc{hasleft}(p)\ \wedge\ \textsc{hasright}(p)\ \wedge\ \Psi_C(p,l)\big)\Big)\Bigg)$$

$$\wedge\ \Bigg(\Big(\textsc{hasleft}(p)\ \wedge\ \textsc{hasright}(p)\Big)\\ \rightarrow\bigvee_{l\in\{\mathbf{;},\mathbf{or},\mathbf{and},\mathbf{if},\mathbf{then},\mathbf{while}\}}\textsc{cmd}(p)=f_{\mathbb{L}}(l)\Bigg)$$

where the formulas $\Psi_{\mathcal{B}}(p)$, $\Psi_{\mathcal{B}_{\swarrow}}(p,b)$, and $\Psi_C(p,l)$ check

- that the given node $p$ is labeled with a Boolean operation,
- that the left child of $p$ is labeled with a Boolean operation (if and only if $b$), and
- that both childs are labeled with Boolean operations or non-Boolean operations depending on the given label $l$, respectively.

Formally, $\Psi_{\mathcal{B}}(p)$, $\Psi_{\mathcal{B}_{\swarrow}}(p,b)$, and $\Psi_C(p,l)$ are defined as follows:

$$\Psi_{\mathcal{B}}(p)\ :=\ \bigvee_{l\in\mathcal{B}\cup V_{\mathbb{B}}\cup\mathcal{I}\cup\{\mathbf{not},\mathbf{and},\mathbf{or}\}}\textsc{cmd}(p)=f_{\mathbb{L}}(l)$$

$$\Psi_{\mathcal{B}_{\swarrow}}(p,b)\ :=\ \bigwedge_{\substack{p'\in N\\ p'>p+1}}\Big(\textsc{left}(p)=p'\ \rightarrow\ \big(b\ \leftrightarrow\ \Psi_{\mathcal{B}}(p)\big)\Big)$$

$$\Psi_C(p,l)\ :=\ \begin{cases}\Psi_{\mathcal{B}_{\swarrow}}(p,\mathit{false})\ \wedge\ \neg\Psi_{\mathcal{B}}(p+1) & \text{if } l\in\{\mathbf{;}\}\\ \Psi_{\mathcal{B}_{\swarrow}}(p,\mathit{true})\ \wedge\ \Psi_{\mathcal{B}}(p+1) & \text{if } l\in\{\mathbf{and},\mathbf{or}\}\\ \Psi_{\mathcal{B}_{\swarrow}}(p,\mathit{true})\ \wedge\ \neg\Psi_{\mathcal{B}}(p+1) & \text{if } l\in\{\mathbf{while}\}\\ \Psi_{\mathcal{B}_{\swarrow}}(p,\mathit{true})\\ \quad\wedge\ \textsc{cmd}(p+1)=f_{\mathbb{L}}(\mathbf{then}) & \text{if } l\in\{\mathbf{if}\}\\ \Psi_{\mathcal{B}_{\swarrow}}(p,\mathit{false})\ \wedge\ \neg\Psi_{\mathcal{B}}(p+1)\\ \quad\wedge\textsc{cmd}(p-1)=f_{\mathbb{L}}(\mathbf{if}) & \text{if } l\in\{\mathbf{then}\}\text{ and } p>0\end{cases}$$

**Lemma 7.** *The formula* $\Psi_{VC}(n, z)$ *is satisfiable if and only if the variables* CMD$(p, j)$, HASLEFT$(p)$, HASRIGHT$(p)$, LEFT$(p, j)$, *and* PARENT$(p, j)$ *witness a valid program* $\mathbb{P}\colon \Delta^* \rightharpoonup \mathbb{L}_{\mathcal{I},\mathcal{O},V_{\mathbb{B}}}$ *with* $|\mathcal{D}_\Delta| = z$ *and* $|V_{\mathbb{B}}| = n$.

*Proof.* "⇒": Assume that $\Psi_{VC}(n, z)$ is satisfiable. Then Constraints 1 to 3 enforce the correct linearization of the node pointers according the natural order of the elements of N. The root is the first entry and the last one must be a leaf. If there is a right child, then it always must be the next entry, as ensured by keeping the corresponding parent pointers of these right childs consistent. If there is a left child, then it must appear as a later entry behind the right child and the parent pointer is consistent as well. Note that there are no programs with only a left child, as guaranteed by the latter constraints. Hence, the constraints ensure that all branches only moving to the right are grouped together according to the indices of $N$ and nodes of the left branches all appear afterwards. Therefore, the induced shape must be a tree, since the parent pointers always point to smaller entries except for the root.

Constraint 4 ensures the correct ranges of the labels, child and parent pointers. The correct root labeling according to Definition 25 is guaranteed by Constraint 5. The last entry always being a leaf and the second to last entry at most having a right child imposes some restrictions on the possible node labels. They are captured by Constraints 6 and 7.

Finally, there are restrictions according to the number of childs with respect to the corresponding labeling. Furthermore, Definition 25 imposes some structural limitations regarding the correct usage of Boolean operations, conditionals and loops as well. Constraint 8 ensures that all 0-nary operations have no child and therefore represent leaves in the tree. All unary operations only have a right child and the corresponding child must be part of a Boolean expression. The requirement is guaranteed through Constraint 9. All remaining operations are binary and have two childs being either part of a Boolean expression or some other structure according to the corresponding operation. The correct usage is verified through Constraint 10 in combination with the usage of the formula $\Psi_C(p, l)$. The correct correspondence of **if** and **then** nodes is also ensured through $\Psi_C(p, l)$.

Note that the constraints do not ensure reactivity of the program, *i.e.*, that all possible executions infinitely often hit an **inout** labeled program node. While this requirement is necessary for the correct semantic evaluation of a program, in order to induce a well-defined Mealy machine, it is not part of our program definition. Reactivity will be ensured as part of the simulation ranking being part of later constraints.

"$\Leftarrow$". Assume there is a program $\mathbb{P}$ with $\mathbb{P}\colon \Delta^* \rightharpoonup \mathbb{L}_{\mathcal{I},\mathcal{O},V_\mathbb{B}}$, $|\mathcal{D}_\Delta| = z$ and $|V_\mathbb{B}| = n$ that satisfies all requirements of Definition 25. We choose $\text{CMD}(p,j)$, $\text{HASLEFT}(p)$, $\text{HASRIGHT}(p)$, $\text{LEFT}(p,j)$, and $\text{PARENT}(p,j)$ such that they reflect $\mathbb{P}$. The corresponding index of each node $p \in \mathcal{D}_\Delta$ is chosen by always starting with the rightmost branch, choosing some non-yet-indexed left child of already indexed branches and applying the procedure recursively until all nodes got an index assigned. All remaining variable instantiations follow deterministically from the definition of $\mathbb{P}$. It is an easy exercise to verify, that the corresponding result indeed satisfies all of the constraints of $\Psi_{VC}(n,z)$. □

We now are ready to complete the encoding by adding the remaining constraints. To this end, let a universal co-Büchi automaton with $k$ rejecting states, as well as $z \in \mathbb{N}^+$ and $n \in \mathbb{N}$ be given. Then the Bounded Program Encoding is given by the formula $\Psi_{BP}(n,z,\mathfrak{A}) := \Psi_{VC}(n,z) \wedge \Psi_R(n,z,\mathfrak{A})$, where $\Psi_R(n,z,\mathfrak{A})$ consists of the following constraints:

1. The initial state is reachable and all rankings and evaluation variables are bounded:

$$\text{RGSTATE}((\mathit{false},\downarrow,\emptyset,\epsilon),q_I) \;\wedge \bigwedge_{m\in M,\,\nu\in 2^\mathcal{I}} \text{RK}(m,\nu) < |M| \;\wedge$$
$$\bigwedge_{m\in M,\,q\in Q} \text{RANKING}(m,q) < |M|\cdot k \;\wedge \bigwedge_{m\in M,\nu\in 2^\mathcal{I}} \text{DEST}(m,\nu) < |M|$$

2. Nodes that are labeled with **inout** and are entered from top are sinks and determine the result of the program evaluation:

$$\bigwedge_{\substack{m\in M,\,\nu\in 2^\mathcal{I}\\ pr_2(m)=\downarrow}} \Big(\text{CMD}(pr_3(m)) = f_\mathbb{L}(\textbf{inout}) \;\rightarrow\; \text{DEST}(m,\nu) = f_M(m) \;\wedge \bigwedge_{o\in pr_2(m)\cap\mathcal{O}} \text{OUT}(m,\nu,o) \;\wedge \bigwedge_{o\in\mathcal{O}\setminus pr_2(m)} \neg\text{OUT}(m,\nu,o)\Big)$$

3. All other operations are evaluated according to their semantics:

$$\bigwedge_{\substack{l\in\mathbb{L}_{\mathcal{I},\mathcal{O},V_\mathbb{B}},\,m\in M,\,\nu\in 2^\mathcal{I}\\ l\neq\textbf{inout}\,\vee\, pr_2(m)\neq\downarrow}} \Big(\text{CMD}(pr_3(m)) = f_\mathbb{L}(l) \;\rightarrow\; \text{RK}(m,\nu) < \text{RK}(\theta(m,\nu,l),\nu)) \;\wedge\; \text{DEST}(\theta(m,\nu,l),\nu) = \text{DEST}(w,\nu) \;\wedge \bigwedge_{o\in\mathcal{O}} \text{OUT}(\theta(m,\nu,l),\nu,o) \leftrightarrow \text{OUT}(m,\nu,o)\Big)$$

4. Each ranking of a vertex of the run graph bounds the number of visited accepting vertices, not counting the current vertex itself:

$$\bigwedge_{(b,d,X,p)\in M,\, q\in Q} \textsc{rgstate}((b,d,X,p),q) \rightarrow$$
$$\bigwedge_{\nu\in 2^{\mathcal{I}}, q'\in Q} \Delta_{\mathfrak{A}}(q,q')[\nu \mapsto \mathit{true}, (\mathcal{I}\setminus\nu)\mapsto \mathit{false}] \quad [o \mapsto \textsc{out}((\mathit{false},\nwarrow,X,p),\nu,o)] \rightarrow$$
$$\bigwedge_{m'\in M} \textsc{dest}((\mathit{false},\nwarrow,X,p),\nu) = m' \rightarrow$$
$$\textsc{rgstate}(m',q') \wedge \textsc{ranking}((b,d,X,p),q) \prec_q \textsc{ranking}(m',q')$$

**Theorem 20.** *For bounds $n, z \in \mathbb{N}$ and a universal co-Büchi automaton $\mathfrak{A}$, the formula $\Psi_{BP}(\mathfrak{A}, n, z)$ is satisfiable if and only if there is a program $\mathbb{P}\colon \Delta^* \rightharpoonup \mathbb{L}_{\mathcal{I},\mathcal{O},V_{\mathbb{B}}}$ with $|\mathcal{D}_\Delta| = z$ and $|V_{\mathbb{B}}| = n$ such that $\mathcal{L}(\mathbb{P}) \subseteq \mathcal{L}(\mathfrak{A})$.*

*Proof.* "⇒": Assume $\Psi_{BP}(\mathfrak{A}, n, z)$ is satisfiable. Then, according to Lemma 7 the $\textsc{cmd}(p,j)$, $\textsc{hasleft}(p)$, $\textsc{hasright}(p)$, $\textsc{left}(p,j)$, and $\textsc{parent}(p,j)$ variables witness a valid program $\mathbb{P}\colon \Delta^* \rightharpoonup \mathbb{L}_{\mathcal{I},\mathcal{O},V_{\mathbb{B}}}$ with $|\mathcal{D}_\Delta| = z$ and $|V_{\mathbb{B}}| = n$.

The evaluation of the program starts at the root, with the direction coming from the top. Therefore, Constraint 1 guarantees that the corresponding initial state is reachable and that all binary encoded variables respect the induced bounds. Similar to the bounded register encoding, the simulation then distinguishes two cases. If an **inout**-command is evaluated and the evaluation comes from the parent node, *i.e.*, the direction is set to $\downarrow$, then the destination pointer $\textsc{dest}(m,\nu)$ gets locked and the $\textsc{out}(m,\nu,o)$ variables reflect the values of the output variables captured by the state $m$ (Constraint 2). Otherwise, the program is evaluated according to $\theta$ from (Construction 7). At the same time, the destination pointer and the output references are copied backward from the final target while the simulation ranking $\textsc{rk}(m,\nu)$ ensures that such a target indeed must exist (Constraint 3).

Finally, the last Constraint 4 ensures the correct construction of the run graph, in a similar fashion as used in the standard bounded synthesis encoding. In this context, however, the $\textsc{out}(m,\nu,o)$ variables indicate the evaluated output and the destination pointers $\textsc{dest}(m,\nu)$ the corresponding successor state. Note that for both variables the direction is updated to $\nwarrow$ and the evaluation variable $b$ is reset to *false* before the evaluation "gets started". These updates ensure that the evaluation is not immediately trapped at the

current **inout**-command, but continues by moving to the parent first. The only exception is the root, which however does not cause any harm, since moving to the parent from the root immediately brings us back to the root. According to the properties of the run graph it, thus, immediately follows that $\mathcal{L}(\mathbb{P}) \subseteq \mathcal{L}(\mathfrak{A})$.

"$\Leftarrow$": Now assume that there is a program $\mathbb{P}$ of size $z$ utilizing only $n$ additional variables. Then, according to Lemma 7 we can encode the program using the corresponding variables. The selection of the remaining variables is determined deterministically according to the Constraints 1 to 4, except for the simulation ranking, which, however, can be chosen strictly decreasing according to a backward evaluation of every simulation step. It is straightforward to see that the chosen variables then indeed satisfy the formula $\Psi_{BP}(\mathfrak{A}, n, z)$. □

Let $\mathfrak{A} = (2^{\mathcal{I} \cup \mathcal{O}}, Q, q_I, \Delta_{\mathfrak{A}}, \textsc{coBüchi}(R))$ be a universal co-Büchi automaton with $|Q| = m$. Let $t = n + |\mathcal{I}| + |\mathcal{O}|$. Then $|\Psi_{RM}(\mathfrak{A}, n, z)| \in O(z^2 m^2 \cdot t 2^t)$ and $\Psi_{RM}(\mathfrak{A}, n, z)$ consists of $j \in O(zm \log(zm) \cdot t 2^t)$ many variables.

# 4 Experimental Results

To understand how the aforementioned approaches evaluate in terms of their practical performance, we apply them to a set of LTL benchmarks. On the one hand, we consider the synthesis times of the different approaches, including the creation of the SAT instances from a given specification and solving these instances afterwards. On the other hand, we look at the synthesized outputs and how they correlate with the corresponding output parameters. For this purpose, we consider subsets of the following benchmarks, where the atomic propositions $i$, $u$, and $r_j$ for $j \in \mathbb{N}$ describe inputs to the system. The atomic propositions $o$ and $g_j$ for $j \in \mathbb{N}$ describe outputs instead.

**Identity:** $\square(i \leftrightarrow o)$

The benchmark serves as a ground reference for the most simple type of a reactive system that just always passes all content received at the input $i$ to the output $o$.

**Delay:** $\square(i \leftrightarrow \bigcirc o)$

The specified system must pass input data to the output as well, but with a delay of one time step. The value that is output initially can be chosen freely by the synthesizer.

**Initial Test:** $(\square o) \leftrightarrow i$

The system either sets the output $o$ always high or always low, where the chosen value dependents on the first input value, as it is provided by the environment.

**Mode Select:** $(i \rightarrow \square o) \wedge (\neg i \rightarrow \square (i \leftrightarrow o))$

The system either sets the output $o$ always high or always passes the provided input, where the chosen mode depends on the input value that is provided at the initial time step.

**Latch:** $\square (u \rightarrow ((i \leftrightarrow o) \wedge (i \rightarrow \bigcirc (o \,\mathcal{W}\, u)) \wedge ((\neg i) \rightarrow \bigcirc ((\neg o) \,\mathcal{W}\, u))))$

The specification describes a latched input $i$, which is only updated, if a signal at the input $u$ is given. Hence, as long as there is no update signal via $u$, the system must keep providing the last value of $i$ since the previous update.

**Detector:** $\bigwedge_{i=0}^{n} (\square \diamondsuit r_i) \leftrightarrow (\square \diamondsuit o)$

The specification describes a monitor, checking whether $n$ clients regularly send requests $r_j$. Therefore, the output $o$ is enabled regularly if and only if all requests are provided regularly as well. The specification is parameterized in $n \in \mathbb{N}$.

**Simple Arbiter:** $\bigwedge_{j=0}^{n-1} \left( \square (r_j \rightarrow \diamondsuit g_j) \right) \;\wedge\; \bigvee_{j=0}^{n-1} (g_j \rightarrow \bigwedge_{\substack{k=0 \\ k \neq j}}^{n-1} \neg g_k)$

The simple arbiter specification as introduced in Section 6 of Chapter II, which is also parameterized in $n \in \mathbb{N}$.

Beside these smaller system examples we also consider the decomposed version of Arm's *Advanced Microcontroller Bus Architecture* (AMBA) [6], which is well known as one of the first industrial examples that has been fully specified using LTL. However, due to the huge complexity of the monolithic specification, as introduced in [76], we instead focused on a decomposed variant that has been introduced in [73]. In this decomposed version, the architecture is split into eight individual components LOCK, ARBITER, ENCODE, DECODE, SHIFT, TINCR, TBURST4, and TSINGLE, where the components LOCK, ARBITER, and ENCODE are parameterized according to the number of clients $n \in \mathbb{N}$. All components are connected according to the architecture of Figure 33. For their concrete LTL specifications, as well as the way they work together with respect to the original specification, we refer the interested reader to [73].

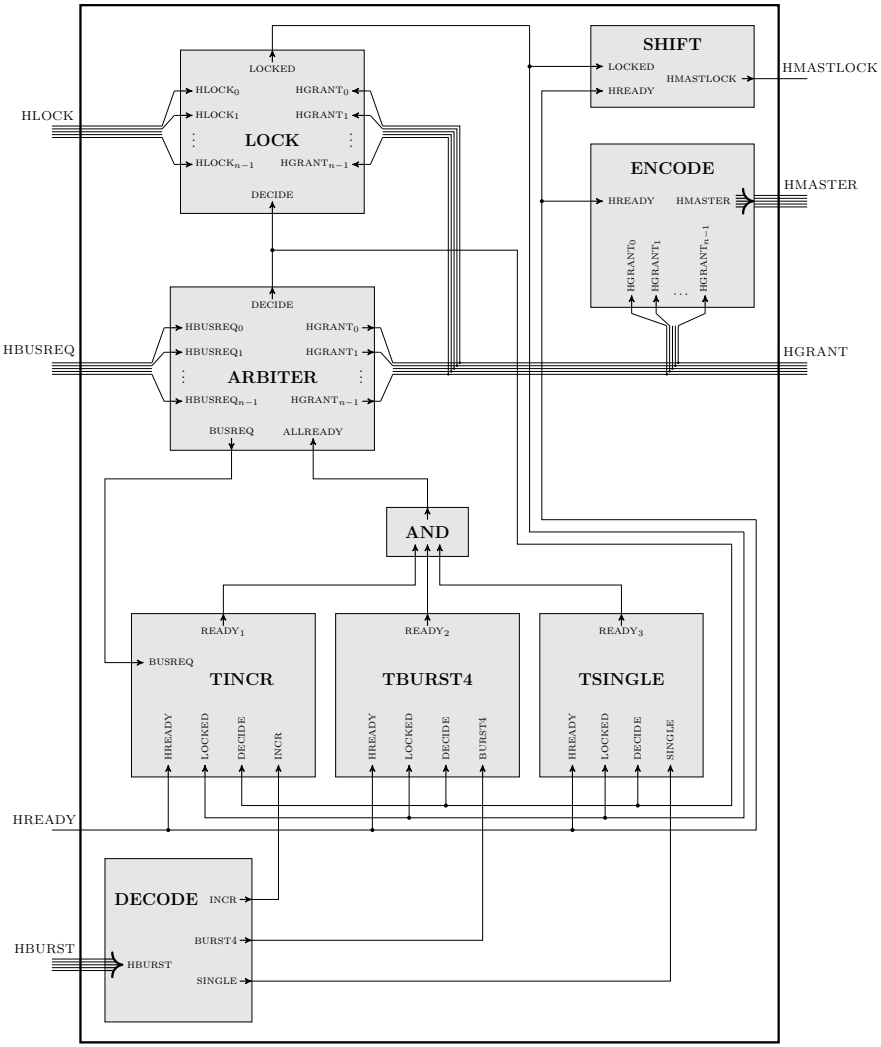



Figure 33: Decomposition of the AMBA AHB arbiter.

All of the aforementioned approaches have been implemented as part of our tool `BoWSer`, the *Bounded Witness Synthesizer*, which allows to create SAT encodings from LTL specifications and the corresponding approach specific parameters. The tool not only creates the encodings, but also automatically

solves the created SAT queries using the SAT solver `maplesat` [93], based on `minisat` (v2.2.0), and allows to extract the witnesses from the corresponding Boolean variable assignments. Due to all encodings relying on the preceding creation of a universal co-Büchi automaton from the given LTL specification, `BoWSer` also utilizes the automaton transformation tool `spot` (v2.5.3) [35] for that purpose.

Furthermore, for the sake of comparison, we also consider the following other synthesis tools:

**Strix [105]:** The tool utilizes a conversion of the specification into a parity game. Therefore, the specification first is pre-processed to syntactically detect LTL fragments that can be turned into deterministic automata with simple winning conditions. From these automata, the solver then creates a parity game that is explored using a forward search. The arena, however, is not constructed completely at first, but generated on the fly out of the cross-product of the specification automata and is only expanded, if necessary [99].

**Ltlsynt [71]:** The tool has been integrated as part of the `spot` library [35]. It first converts an LTL specification to a generalized Büchi automaton with a transition based acceptance condition, which then is turned into a parity game afterwards. The translation utilizes heuristics to obtain efficient intermediate translations. The parity game is solved using an adapted version of Zielonka's algorithm [152], modified to work on games with transition-based winning conditions.

**BoSy [41]:** The tool also builds on the bounded synthesis approach, but focuses more on other constraint systems than SAT, such as QBF, DQBF or SMT. The tool can be parameterized according to different constraint solvers and automata transformation tools, as well as in selecting a linear or exponential search strategy. We use the tool in the default configuration, as provided for the synthesis competition SYNTCOMP 2019 [72].

As all of the aforementioned tools do not create a Mealy machine directly, but instead output AIGER circuits according to the SYNTCOMP LTL synthesis rules [70]. Therefore, we convert the created circuits to Mealy machines first, according to Construction 4, in order to acquire the data for our later comparison.

All experiments have been executed on a quad-core Intel Xeon processor (E3-1271 v3, 3.6GHz, 32 GB RAM, PC1600, ECC), running Ubuntu 64bit LTS 16.04.

We first consider the evaluation of the bounded cycle synthesis approach in comparison to the standard bounded synthesis approach, `Strix`, `Ltlsynt`, and `BoSy`. The corresponding results are listed in Table 5. For `Strix`, `Ltlsynt`, and `BoSy`, we measured the synthesis time in seconds, the size of the Mealy machine $|\mathbb{M}|$, as induced by the created circuit, the number of cycles of the machine $\mathtt{C}(\mathbb{M})$, and the number of latches $l$ as well as the number of gates $g$. For `BoWSer` we directly synthesized the corresponding Mealy machine and thus have no measures for latches or gates.

`BoWSer` uses the following search strategy for finding a minimal solution in both parameters: $|\mathbb{M}|$ and $\mathtt{C}(\mathbb{M})$. The tool first linearly increases the state bound using the standard bounded synthesis encoding until some minimal Mealy machine has been found. Then, the found state bound gets fixed and the cycle bound gets increased in the same linear fashion, but this time using the bounded cycle synthesis encoding. We terminate as soon as the first realizable result is found. The result therefore is minimal in the number of cycles under the previously determined bound. Note that the individual solving times of each of the corresponding SAT queries of `BoWSer` is presented in the table as well. The best result for each instance is highlighted in green according to each measured category.

The results show that `BoWSer` always finds an optimal solution with respect to the number of cycles and Mealy states. The fastest tool on all benchmarks is `Ltlsynt`, but with respect to the size and the number of cycles of the created solutions it is also the worst. In contrast, the tools `Strix` and `BoSy` seem to provide good intermediate tradeoffs between the synthesis times and the solution quality for most of the benchmarks. However, they also fail in some cases. For example, consider the results for the AMBA arbiter specification with four clients, where `Strix` produces a comparably large solution, which is hard to inspect by a human developer, and `BoSy` takes even longer than `BoWSer`, while still finding an non-optimal solution in the end.

The experiments also indicate that finding an optimal solution with respect to the number states and cycles of the underlying Mealy machine is a feasible task in general. As expected, synthesis may take a little bit longer, but it is not that asking for the minimal number of cycles turns the problem to be completely out of scope. Also note that using a non-linear search strategy for the instantiation of solving multiple SAT queries in parallel can speed up synthesis even more, as it would be for example the case for the AMBA TBurst4 specification.

With the first results for `Strix`, `Ltlsynt`, `BoSy`, and the bounded cycle technique at hand, we are ready to proceed to the evaluation of the output sensitive synthesis of circuits, reactive programs and register machines.

| Benchmark | Strix | | | | | Ltlsynt | | | | | BoSy | | | | | BoWSer | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Time (s) | $\|\mathbb{M}\|$ | $\mathbb{C}(\mathbb{M})$ | l | g | Time (s) | $\|\mathbb{M}\|$ | $\mathbb{C}(\mathbb{M})$ | l | g | Time (s) | $\|\mathbb{M}\|$ | $\mathbb{C}(\mathbb{M})$ | l | g | Time (s) | $\|\mathbb{M}\|$ | $\mathbb{C}(\mathbb{M})$ |
| ModeSelect | 0.724 | 3 | 2 | 2 | 6 | 0.004 | 3 | 2 | 2 | 12 | 0.296 | 3 | 2 | 2 | 4 | 0.100 | 3 | 2 |
| Standard | ($n$: 1 ✗ 0.004) ($n$: 2 ✗ 0.008) ($n$: 3 ✓ 0.016) [$\Sigma = 0.028$] | | | | | | | | | | | | | | | | | |
| Cycle ($n$: 3) | ($c$: 1 ✗ 0.028) ($c$: 2 ✓ 0.044) | | | | | | | | | | | | | | | | | |
| Detector1 | 0.428 | 1 | 1 | 0 | 0 | 0.004 | 4 | 4 | 2 | 20 | 0.276 | 1 | 1 | 1 | 1 | 0.024 | 1 | 1 |
| Standard | ($n$: 1 ✓ 0.012) [$\Sigma = 0.012$] | | | | | | | | | | | | | | | | | |
| Cycle ($n$: 1) | ($c$: 1 ✓ 0.012) | | | | | | | | | | | | | | | | | |
| Detector2 | 0.500 | 2 | 3 | 1 | 3 | 0.008 | 12 | 27 | 4 | 160 | 0.348 | 2 | 3 | 1 | 4 | 0.220 | 2 | 3 |
| Standard | ($n$: 1 ✗ 0.016) ($n$: 2 ✓ 0.036) [$\Sigma = 0.052$] | | | | | | | | | | | | | | | | | |
| Cycle ($n$: 2) | ($c$: 1 ✗ 0.072) ($c$: 2 ✗ 0.044) ($c$: 3 ✓ 0.052) | | | | | | | | | | | | | | | | | |
| Detector3 | 0.936 | 3 | 6 | 2 | 8 | 0.012 | 39 | 210 | 6 | 1063 | 0.640 | 3 | 6 | 2 | 15 | 1.552 | 3 | 4 |
| Standard | ($n$: 1 ✗ 0.024) ($n$: 2 ✗ 0.072) ($n$: 3 ✓ 0.184) [$\Sigma = 0.280$] | | | | | | | | | | | | | | | | | |
| Cycle ($n$: 3) | ($c$: 1 ✗ 0.232) ($c$: 2 ✗ 0.268) ($c$: 3 ✗ 0.496) ($c$: 4 ✓ 0.276) | | | | | | | | | | | | | | | | | |
| Detector4 | 1.248 | 4 | 10 | 2 | 16 | 0.052 | 174 | 2043 | 8 | 11620 | 1.024 | 4 | 13 | 2 | 18 | 54.904 | 4 | 5 |
| Standard | ($n$: 1 ✗ 0.040) ($n$: 2 ✗ 0.144) ($n$: 3 ✗ 1.144) ($n$: 4 ✓ 1.404) [$\Sigma = 2.732$] | | | | | | | | | | | | | | | | | |
| Cycle ($n$: 4) | ($c$: 1 ✗ 8.832) ($c$: 2 ✗ 6.004) ($c$: 3 ✗ 13.384) ($c$: 4 ✗ 21.128) ($c$: 5 ✓ 2.824) | | | | | | | | | | | | | | | | | |
| SimpleArbiter2 | 1.324 | 3 | 1 | 2 | 3 | 0.012 | 8 | 18 | 3 | 117 | 0.392 | 3 | 1 | 2 | 3 | 0.276 | 3 | 1 |
| Standard | ($n$: 1 ✗ 0.020) ($n$: 2 ✗ 0.040) ($n$: 3 ✓ 0.092) [$\Sigma = 0.152$] | | | | | | | | | | | | | | | | | |
| Cycle ($n$: 3) | ($c$: 1 ✓ 0.124) | | | | | | | | | | | | | | | | | |
| SimpleArbiter3 | 1.264 | 3 | 1 | 2 | 3 | 0.016 | 8 | 18 | 3 | 117 | 0.332 | 3 | 1 | 2 | 4 | 0.276 | 3 | 1 |
| Standard | ($n$: 1 ✗ 0.020) ($n$: 2 ✗ 0.040) ($n$: 3 ✓ 0.092) [$\Sigma = 0.152$] | | | | | | | | | | | | | | | | | |
| Cycle ($n$: 3) | ($c$: 1 ✓ 0.124) | | | | | | | | | | | | | | | | | |
| SimpleArbiter4 | 1.380 | 4 | 1 | 2 | 5 | 0.024 | 20 | 69 | 5 | 522 | 0.640 | 4 | 1 | 2 | 4 | 1.260 | 4 | 1 |
| Standard | ($n$: 1 ✗ 0.040) ($n$: 2 ✗ 0.088) ($n$: 3 ✗ 0.176) ($n$: 4 ✓ 0.416) [$\Sigma = 0.720$] | | | | | | | | | | | | | | | | | |
| Cycle ($n$: 4) | ($c$: 1 ✓ 0.540) | | | | | | | | | | | | | | | | | |
| AmbaShift2 | 1.008 | 5 | 8 | 3 | 12 | 0.004 | 3 | 3 | 2 | 24 | 0.276 | 2 | 3 | 1 | 3 | 0.108 | 2 | 3 |
| Standard | ($n$: 1 ✗ 0.012) ($n$: 2 ✓ 0.020) [$\Sigma = 0.032$] | | | | | | | | | | | | | | | | | |
| Cycle ($n$: 2) | ($c$: 1 ✗ 0.020) ($c$: 2 ✗ 0.024) ($c$: 3 ✓ 0.032) | | | | | | | | | | | | | | | | | |
| AmbaTSingle | 1.652 | 6 | 11 | 3 | 21 | 0.012 | 14 | 18 | 4 | 271 | 0.888 | 4 | 5 | 2 | 17 | 6.032 | 4 | 4 |
| Standard | ($n$: 1 ✗ 0.040) ($n$: 2 ✗ 0.124) ($n$: 3 ✗ 0.276) ($n$: 4 ✓ 0.864) [$\Sigma = 1.304$] | | | | | | | | | | | | | | | | | |
| Cycle ($n$: 4) | ($c$: 1 ✗ 1.216) ($c$: 2 ✗ 1.168) ($c$: 3 ✗ 1.216) ($c$: 4 ✓ 1.128) | | | | | | | | | | | | | | | | | |
| AmbaTIncr | 1.240 | 5 | 8 | 3 | 19 | 0.020 | 41 | 123 | 6 | 1619 | 1.104 | 4 | 6 | 2 | 21 | 7.748 | 4 | 3 |
| Standard | ($n$: 1 ✗ 0.052) ($n$: 2 ✗ 0.200) ($n$: 3 ✗ 0.600) ($n$: 4 ✓ 1.344) [$\Sigma = 2.196$] | | | | | | | | | | | | | | | | | |
| Cycle ($n$: 4) | ($c$: 1 ✗ 1.800) ($c$: 2 ✗ 1.776) ($c$: 3 ✓ 1.976) | | | | | | | | | | | | | | | | | |
| AmbaTBurst4 | 2.020 | 9 | 16 | 4 | 36 | 0.048 | 16 | 16 | 4 | 205 | 3.216 | 7 | 14 | 3 | 51 | 7699.876 | 7 | 7 |
| Standard | ($n$: 1 ✗ 0.060) ($n$: 2 ✗ 0.172) ($n$: 3 ✗ 0.368) ($n$: 4 ✗ 1.224) ($n$: 5 ✗ 7.344) ($n$: 6 ✗ 49.400) ($n$: 7 ✓ 4.400) [$\Sigma = 62.968$] | | | | | | | | | | | | | | | | | |
| Cycle ($n$: 7) | ($c$: 1 ✗ 144.936) ($c$: 2 ✗ 199.720) ($c$: 3 ✗ 380.508) ($c$: 4 ✗ 249.988) ($c$: 5 ✗ 2265.350) ($c$: 6 ✗ 3969.690) ($c$: 7 ✓ 426.716) | | | | | | | | | | | | | | | | | |
| AmbaEncode2 | 1.384 | 3 | 7 | 2 | 14 | 0.020 | 4 | 4 | 2 | 53 | 0.332 | 2 | 3 | 1 | 8 | 0.144 | 2 | 3 |
| Standard | ($n$: 1 ✗ 0.020) ($n$: 2 ✓ 0.028) [$\Sigma = 0.048$] | | | | | | | | | | | | | | | | | |
| Cycle ($n$: 2) | ($c$: 1 ✗ 0.028) ($c$: 2 ✗ 0.028) ($c$: 3 ✓ 0.040) | | | | | | | | | | | | | | | | | |
| AmbaLock2 | 1.396 | 4 | 10 | 2 | 13 | 0.020 | 5 | 6 | 3 | 93 | 0.524 | 3 | 5 | 2 | 13 | 0.976 | 3 | 5 |
| Standard | ($n$: 1 ✗ 0.020) ($n$: 2 ✗ 0.044) ($n$: 3 ✓ 0.116) [$\Sigma = 0.180$] | | | | | | | | | | | | | | | | | |
| Cycle ($n$: 3) | ($c$: 1 ✗ 0.128) ($c$: 2 ✗ 0.144) ($c$: 3 ✗ 0.144) ($c$: 4 ✗ 0.160) ($c$: 5 ✓ 0.220) | | | | | | | | | | | | | | | | | |
| AmbaArbiter2 | 1.652 | 8 | 21 | 3 | 27 | 0.032 | 7 | 12 | 3 | 196 | 0.692 | 2 | 3 | 1 | 9 | 0.508 | 2 | 3 |
| Standard | ($n$: 1 ✗ 0.052) ($n$: 2 ✓ 0.104) [$\Sigma = 0.156$] | | | | | | | | | | | | | | | | | |
| Cycle ($n$: 2) | ($c$: 1 ✗ 0.108) ($c$: 2 ✗ 0.112) ($c$: 3 ✓ 0.132) | | | | | | | | | | | | | | | | | |
| AmbaArbiter3 | 2.120 | 22 | 173 | 5 | 315 | 0.148 | 27 | 264 | 5 | 1890 | 27.988 | 4 | 9 | 2 | 32 | 8.796 | 3 | 4 |
| Standard | ($n$: 1 ✗ 0.480) ($n$: 2 ✗ 0.680) ($n$: 3 ✓ 1.248) [$\Sigma = 2.408$] | | | | | | | | | | | | | | | | | |
| Cycle ($n$: 3) | ($c$: 1 ✗ 1.500) ($c$: 2 ✗ 1.540) ($c$: 3 ✗ 1.644) ($c$: 4 ✓ 1.704) | | | | | | | | | | | | | | | | | |
| AmbaArbiter4 | 4.880 | 45 | 877 | 6 | 996 | 1.716 | 106 | 2553 | 7 | 15224 | 228.964 | 4 | 10 | 2 | 41 | 104.956 | 4 | 5 |
| Standard | ($n$: 1 ✗ 4.316) ($n$: 2 ✗ 5.192) ($n$: 3 ✗ 6.604) ($n$: 4 ✓ 12.752) [$\Sigma = 28.864$] | | | | | | | | | | | | | | | | | |
| Cycle ($n$: 4) | ($c$: 1 ✗ 13.640) ($c$: 2 ✗ 14.460) ($c$: 3 ✗ 15.420) ($c$: 4 ✗ 16.168) ($c$: 5 ✓ 16.404) | | | | | | | | | | | | | | | | | |

Table 5: Experimental evaluation of the bounded cycle synthesis approach.

Note that all of the aforementioned approaches require multiple parameters to be set correspondingly. Therefore, we use a full spectrum analysis including the search for all possible circuits that range from $l = 0$ to $l = 3$ latches and $g = 0$ to $g = 9$ gates, all possible register machines that range from $r = 0$ to $r = 2$ registers and $i = 1$ to $i = 9$ instructions, and all possible reactive programs that range from $v = 0$ to $v = 1$ variables and $s = 1$ to $s = 14$ program nodes. The corresponding results of all of the analyzed specifications are provided in Figures 34 to 46.

Satisfiable instances witnessing a realizing solution for the corresponding synthesis problems are marked with a ✓ and are highlighted in green. Unsatisfiable instances on the other hand are marked with a ✗ and are highlighted in orange. The corresponding encoding creation plus solving times are given next to these symbols in seconds. For most benchmarks, we used a timeout of an hour (3600 seconds), which however, was increased up to two days for some benchmarks out of curiosity. It only helped in a single case for synthesizing some reactive program for the `delay` specification. Timeouts are highlighted in red and indicate the reached limit. If multiple solutions could be found, the smallest result is highlighted in the table and the corresponding implementation is depicted next to it. If the minimal solution is not unique, both of the corresponding solutions are selected and depicted, correspondingly.

The results indicate that circuits are the easiest to synthesize, followed by register machines and reactive programs. Except for the `TBurst4` component of the AMBA arbiter we found realizing circuits for all of the given specifications. In general, the AMBA components seem to be much harder than the other examples, where we only could find a register machine for the `Shift` component, but no register machines for all other components and no reactive programs at all.

Also note that the complexity of each benchmark depends on the particular implementation type and the selected parameters. Consider for example the results of the simple arbiter specification, where it is hard to find a register machine that realizes the specification, but a circuit and a reactive program still are manageable. Another example is given by the `ModeSelect` specification, which is straightforward to be synthesized as a circuit, but requires a complex register machine and where a reactive program could not be found.

Finally reconsider that a multidimensional search space also implies tradeoffs, as for example indicated by the `delay` specification. Overall, the specification can be implemented very compactly for all of the presented implementation models. However, in the case of reactive programs this requires the introduction of an extra program variable, which is only dispensable at the price of a much more complex program flow.

$$\square(i \leftrightarrow o)$$

| Circuit | $l$: 0 | $l$: 1 | $l$: 2 | $l$: 3 |
|---|---|---|---|---|
| $g$: 0 | ✓ **0.004** | ✓ 0.008 | ✓ 0.024 | ✓ 0.104 |
| $g$: 1 | ✓ 0.008 | ✓ 0.012 | ✓ 0.032 | ✓ 0.124 |
| $g$: 2 | ✓ 0.012 | ✓ 0.020 | ✓ 0.048 | ✓ 0.160 |
| $g$: 3 | ✓ 0.020 | ✓ 0.036 | ✓ 0.072 | ✓ 0.204 |
| $g$: 4 | ✓ 0.032 | ✓ 0.052 | ✓ 0.096 | ✓ 0.280 |
| $g$: 5 | ✓ 0.056 | ✓ 0.076 | ✓ 0.152 | ✓ 0.368 |
| $g$: 6 | ✓ 0.072 | ✓ 0.112 | ✓ 0.212 | ✓ 0.480 |
| $g$: 7 | ✓ 0.120 | ✓ 0.160 | ✓ 0.276 | ✓ 0.592 |
| $g$: 8 | ✓ 0.152 | ✓ 0.204 | ✓ 0.328 | ✓ 0.704 |
| $g$: 9 | ✓ 0.208 | ✓ 0.264 | ✓ 0.412 | ✓ 0.868 |


| Register | $r$: 0 | $r$: 1 | $r$: 2 |
|---|---|---|---|
| $i$: 1 | ✗ 0.020 | ✗ 0.084 | ✗ 0.432 |
| $i$: 2 | ✗ 0.096 | ✗ 0.500 | ✗ 2.692 |
| $i$: 3 | ✓ **0.304** | ✓ 1.480 | ✓ 8.536 |
| $i$: 4 | ✓ 0.656 | ✓ 3.488 | ✓ 25.600 |
| $i$: 5 | ✓ 1.120 | ✓ 5.872 | ✓ 100.720 |
| $i$: 6 | ✓ 1.644 | ✓ 13.064 | ✓ 105.048 |
| $i$: 7 | ✓ 2.728 | ✓ 22.836 | ✓ 163.816 |
| $i$: 8 | ✓ 6.544 | ✓ 28.312 | ✓ 229.116 |
| $i$: 9 | ✓ 7.892 | ✓ 60.616 | ✓ 2101.320 |

| Program | $v$: 0 | $v$: 1 |
|---|---|---|
| $s$: 1 | ✗ 0.028 | ✗ 0.192 |
| $s$: 2 | ✗ 0.264 | ✗ 1.940 |
| $s$: 3 | ✗ 0.980 | ✗ 7.892 |
| $s$: 4 | ✓ **3.340** | ✓ 22.388 |
| $s$: 5 | ✓ 5.828 | ✓ 39.888 |
| $s$: 6 | ✓ 10.572 | ✓ 78.212 |
| $s$: 7 | ✓ 15.004 | ✓ 124.872 |
| $s$: 8 | ✓ 25.172 | ✓ 189.852 |
| $s$: 9 | ✓ 41.364 | ✓ 248.796 |
| $s$: 10 | ✓ 55.008 | ✓ 393.140 |
| $s$: 11 | ✓ 84.288 | ✓ 593.852 |
| $s$: 12 | ✓ 92.796 | ✓ 861.732 |
| $s$: 13 | ✓ 135.336 | ✓ 1237.460 |
| $s$: 14 | ✓ 187.488 | ✓ 1943.030 |

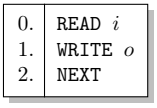
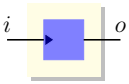




Figure 34: Experimental results: `identity.tlsf`

$$\square(i \leftrightarrow \bigcirc o)$$

| Circuit | $l: 0$ | $l: 1$ | $l: 2$ | $l: 3$ |
|---|---|---|---|---|
| $g: 0$ | ✗ 0.004 | ✓ **0.012** | ✓ 0.048 | ✓ 0.244 |
| $g: 1$ | ✗ 0.008 | ✓ 0.016 | ✓ 0.064 | ✓ 0.264 |
| $g: 2$ | ✗ 0.008 | ✓ 0.024 | ✓ 0.076 | ✓ 0.300 |
| $g: 3$ | ✗ 0.016 | ✓ 0.040 | ✓ 0.096 | ✓ 0.356 |
| $g: 4$ | ✗ 0.024 | ✓ 0.056 | ✓ 0.124 | ✓ 0.444 |
| $g: 5$ | ✗ 0.032 | ✓ 0.084 | ✓ 0.172 | ✓ 0.540 |
| $g: 6$ | ✗ 0.044 | ✓ 0.116 | ✓ 0.252 | ✓ 0.740 |
| $g: 7$ | ✗ 0.072 | ✓ 0.172 | ✓ 0.308 | ✓ 0.792 |
| $g: 8$ | ✗ 0.092 | ✓ 0.216 | ✓ 0.376 | ✓ 0.864 |
| $g: 9$ | ✗ 0.112 | ✓ 0.272 | ✓ 0.456 | ✓ 1.016 |




| Register | $r: 0$ | $r: 1$ | $r: 2$ |
|---|---|---|---|
| $i: 1$ | ✗ 0.040 | ✗ 0.196 | ✗ 1.052 |
| $i: 2$ | ✗ 0.220 | ✗ 1.296 | ✗ 7.604 |
| $i: 3$ | ✓ **0.768** | ✓ 4.208 | ✓ 26.088 |
| $i: 4$ | ✓ 1.720 | ✓ 10.276 | ✓ 57.732 |
| $i: 5$ | ✓ 3.152 | ✓ 20.932 | ✓ 373.852 |
| $i: 6$ | ✓ 5.276 | ✓ 39.596 | > 3600.00 |
| $i: 7$ | ✓ 10.024 | ✓ 56.520 | ✓ 275.632 |
| $i: 8$ | ✓ 15.420 | ✓ 124.916 | ✓ 613.656 |
| $i: 9$ | ✓ 14.708 | ✓ 370.336 | > 3600.00 |

| Program | $v: 0$ | $v: 1$ |
|---|---|---|
| $s: 1$ | ✗ 0.064 | ✗ 0.472 |
| $s: 2$ | ✗ 0.792 | ✗ 5.676 |
| $s: 3$ | ✗ 3.044 | ✗ 25.076 |
| $s: 4$ | ✗ 8.776 | ✗ 64.488 |
| $s: 5$ | ✗ 14.772 | ✗ 110.048 |
| $s: 6$ | ✗ 27.592 | ✗ 218.260 |
| $s: 7$ | ✗ 51.976 | ✓ **357.244** |
| $s: 8$ | ✗ 169.832 | ✓ 664.464 |
| $s: 9$ | ✗ 1029.580 | ✓ 869.296 |
| $s: 10$ | ✗ 11092.500 | ✓ 1198.980 |
| $s: 11$ | ✓ **2556.690** | ✓ 1684.700 |
| $s: 12$ | (✓ 7167.890) | ✓ 2943.270 |
| $s: 13$ | ✓ 2091.470 | > 3600.00 |
| $s: 14$ | ✓ 2742.980 | > 3600.00 |

```
0.  WRITE o
1.  READ i
2.  NEXT
```

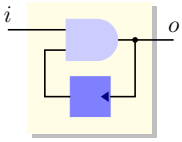
$v_0$ := $i$;
**inout**;
$o$ := $v_0$

**if** ($i$) **then** {
  **inout**;
  $o$ := *true*
} **else** {
  **inout**;
  $o$ := *false*
}

Figure 35: Experimental results: `delay.tlsf`

$$(\square o) \leftrightarrow i$$

| Circuit | $l$: 0 | $l$: 1 | $l$: 2 | $l$: 3 |
|---|---|---|---|---|
| $g$: 0 | ✗ 0.008 | ✗ 0.012 | ✗ 0.040 | ✗ 0.272 |
| $g$: 1 | ✗ 0.008 | ✓ **0.016** | ✓ 0.056 | ✓ 0.328 |
| $g$: 2 | ✗ 0.012 | ✓ 0.024 | ✓ 0.080 | ✓ 0.340 |
| $g$: 3 | ✗ 0.016 | ✓ 0.044 | ✓ 0.100 | ✓ 0.336 |
| $g$: 4 | ✗ 0.024 | ✓ 0.056 | ✓ 0.128 | ✓ 0.436 |
| $g$: 5 | ✗ 0.032 | ✓ 0.084 | ✓ 0.180 | ✓ 0.600 |
| $g$: 6 | ✗ 0.044 | ✓ 0.116 | ✓ 0.240 | ✓ 0.708 |
| $g$: 7 | ✗ 0.068 | ✓ 0.176 | ✓ 0.320 | ✓ 0.776 |
| $g$: 8 | ✗ 0.092 | ✓ 0.216 | ✓ 0.364 | ✓ 0.912 |
| $g$: 9 | ✗ 0.116 | ✓ 0.272 | ✓ 0.448 | ✓ 1.100 |




| Register | $r$: 0 | $r$: 1 | $r$: 2 |
|---|---|---|---|
| $i$: 1 | ✗ 0.036 | ✗ 0.192 | ✗ 1.036 |
| $i$: 2 | ✗ 0.268 | ✗ 1.172 | ✗ 6.796 |
| $i$: 3 | ✗ 0.576 | ✗ 3.372 | ✗ 19.660 |
| $i$: 4 | ✓ **1.752** | ✓ 9.776 | ✓ 48.568 |
| $i$: 5 | ✓ 2.804 | ✓ 17.504 | ✓ 361.704 |
| $i$: 6 | ✓ 6.292 | ✓ 25.836 | > 3600.00 |
| $i$: 7 | ✓ 10.400 | ✓ 44.680 | > 3600.00 |
| $i$: 8 | ✓ 17.176 | ✓ 68.532 | > 3600.00 |
| $i$: 9 | ✓ 17.472 | ✓ 256.060 | > 3600.00 |

| Program | $v$: 0 | $v$: 1 |
|---|---|---|
| $s$: 1 | ✗ 0.084 | ✗ 0.564 |
| $s$: 2 | ✗ 0.820 | ✗ 5.664 |
| $s$: 3 | ✗ 3.212 | ✗ 22.620 |
| $s$: 4 | ✗ 8.392 | ✗ 58.728 |
| $s$: 5 | ✗ 13.968 | ✗ 98.596 |
| $s$: 6 | ✓ **26.704** | ✓ 185.588 |
| $s$: 7 | ✓ 38.924 | ✓ 289.440 |
| $s$: 8 | ✓ 66.972 | ✓ 464.196 |
| $s$: 9 | ✓ 90.952 | ✓ 676.900 |
| $s$: 10 | ✓ 123.536 | ✓ 816.852 |
| $s$: 11 | ✓ 169.440 | ✓ 1306.580 |
| $s$: 12 | ✓ 232.112 | ✓ 1746.150 |
| $s$: 13 | ✓ 337.512 | ✓ 3243.710 |
| $s$: 14 | ✓ 402.968 | ✓ 2280.110 |

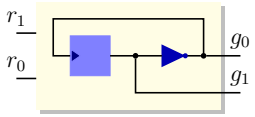
```
0.  READ i
1.  WRITE o
2.  NEXT
3.  JMP 2
```

$o$ := $o$ **or** $i$;
**inout**

Figure 36: Experimental results: `initial-test.tlsf`

$$\square\neg(g_0 \wedge g_1) \;\wedge\; \square(r_0 \rightarrow \diamondsuit g_0) \;\wedge\; \square(r_1 \rightarrow \diamondsuit g_1)$$

| Circuit | $l\colon 0$ | $l\colon 1$ | $l\colon 2$ | $l\colon 3$ |
|---|---|---|---|---|
| $g\colon 0$ | ✗ 0.012 | ✓ **0.028** | ✓ 0.092 | ✓ 0.408 |
| $g\colon 1$ | ✗ 0.016 | ✓ 0.036 | ✓ 0.112 | ✓ 0.536 |
| $g\colon 2$ | ✗ 0.020 | ✓ 0.052 | ✓ 0.148 | ✓ 0.568 |
| $g\colon 3$ | ✗ 0.028 | ✓ 0.072 | ✓ 0.192 | ✓ 0.780 |
| $g\colon 4$ | ✗ 0.040 | ✓ 0.100 | ✓ 0.272 | ✓ 1.108 |
| $g\colon 5$ | ✗ 0.056 | ✓ 0.152 | ✓ 0.364 | ✓ 1.044 |
| $g\colon 6$ | ✗ 0.072 | ✓ 0.216 | ✓ 0.468 | ✓ 1.268 |
| $g\colon 7$ | ✗ 0.108 | ✓ 0.276 | ✓ 0.556 | ✓ 1.496 |
| $g\colon 8$ | ✗ 0.140 | ✓ 0.336 | ✓ 0.696 | ✓ 1.884 |
| $g\colon 9$ | ✗ 0.172 | ✓ 0.424 | ✓ 0.852 | ✓ 2.324 |




| Register | $r\colon 0$ | $r\colon 1$ | $r\colon 2$ |
|---|---|---|---|
| $i\colon 1$ | ✗ 0.336 | ✗ 1.792 | ✗ 9.508 |
| $i\colon 2$ | ✗ 12.048 | > 3600.00 | > 3600.00 |
| $i\colon 3$ | ✗ 61.136 | > 3600.00 | > 3600.00 |
| $i\colon 4$ | > 3600.00 | > 3600.00 | > 3600.00 |
| $i\colon 5$ | > 3600.00 | > 3600.00 | > 3600.00 |
| $i\colon 6$ | > 3600.00 | > 3600.00 | > 3600.00 |
| $i\colon 7$ | > 3600.00 | > 3600.00 | > 3600.00 |
| $i\colon 8$ | > 3600.00 | > 3600.00 | > 3600.00 |
| $i\colon 9$ | > 3600.00 | > 3600.00 | > 3600.00 |

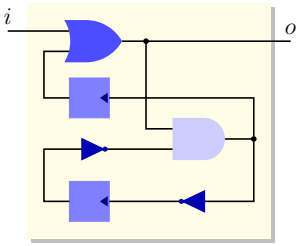
| Program | $v\colon 0$ | $v\colon 1$ |
|---|---|---|
| $s\colon 1$ | ✗ 1.144 | ✗ 7.672 |
| $s\colon 2$ | ✗ 9.784 | ✗ 66.100 |
| $s\colon 3$ | ✗ 42.376 | ✗ 278.096 |
| $s\colon 4$ | ✗ 85.752 | ✗ 619.724 |
| $s\colon 5$ | ✗ 154.004 | ✗ 1030.220 |
| $s\colon 6$ | ✗ 348.228 | ✗ 2514.150 |
| $s\colon 7$ | ✗ 756.956 | > 3600.00 |
| $s\colon 8$ | ✓ **790.016** | > 3600.00 |
| $s\colon 9$ | ✓ 1178.710 | > 3600.00 |
| $s\colon 10$ | ✓ 1889.660 | > 3600.00 |
| $s\colon 11$ | > 3600.00 | > 3600.00 |
| $s\colon 12$ | > 3600.00 | > 3600.00 |
| $s\colon 13$ | > 3600.00 | > 3600.00 |
| $s\colon 14$ | > 3600.00 | > 3600.00 |

**inout**
$g_1$ := $g_0$;
$g_0$ := **not** $g_1$

Figure 37: Experimental results: `simple-arbiter-2.tlsf`

$$(i \rightarrow \square o) \wedge (\neg i \rightarrow \square(i \leftrightarrow o))$$

| Circuit | $l$: 0 | $l$: 1 | $l$: 2 | $l$: 3 |
|---|---|---|---|---|
| $g$: 0 | ✗ 0.008 | ✗ 0.012 | ✗ 0.036 | ✗ 0.224 |
| $g$: 1 | ✗ 0.008 | ✗ 0.016 | ✗ 0.048 | ✗ 1.032 |
| $g$: 2 | ✗ 0.008 | ✗ 0.020 | ✓ **0.080** | ✓ 0.388 |
| $g$: 3 | ✗ 0.016 | ✗ 0.032 | ✓ 0.100 | ✓ 0.368 |
| $g$: 4 | ✗ 0.024 | ✗ 0.044 | ✓ 0.132 | ✓ 0.544 |
| $g$: 5 | ✗ 0.032 | ✗ 0.064 | ✓ 0.184 | ✓ 0.596 |
| $g$: 6 | ✗ 0.044 | ✗ 0.092 | ✓ 0.256 | ✓ 0.912 |
| $g$: 7 | ✗ 0.072 | ✗ 0.140 | ✓ 0.332 | ✓ 1.264 |
| $g$: 8 | ✗ 0.092 | ✗ 0.164 | ✓ 0.376 | ✓ 1.424 |
| $g$: 9 | ✗ 0.116 | ✗ 0.208 | ✓ 0.496 | ✓ 1.304 |




| Register | $r$: 0 | $r$: 1 | $r$: 2 |
|---|---|---|---|
| $i$: 1 | ✗ 0.040 | ✗ 0.192 | ✗ 1.064 |
| $i$: 2 | ✗ 0.220 | ✗ 1.288 | ✗ 7.364 |
| $i$: 3 | ✗ 0.648 | ✗ 3.448 | ✗ 30.400 |
| $i$: 4 | ✗ 2.180 | ✗ 12.200 | ✗ 88.452 |
| $i$: 5 | ✗ 20.660 | ✗ 308.880 | ✗ 1404.490 |
| $i$: 6 | ✗ 770.836 | > 3600.00 | > 3600.00 |
| $i$: 7 | > 3600.00 | > 3600.00 | > 3600.00 |
| $i$: 8 | > 3600.00 | > 3600.00 | > 3600.00 |
| $i$: 9 | > 3600.00 | ✓ **1497.680** | > 3600.00 |

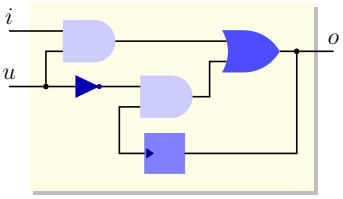
| Program | $v$: 0 | $v$: 1 |
|---|---|---|
| $s$: 1 | ✗ 0.068 | ✗ 0.524 |
| $s$: 2 | ✗ 0.712 | ✗ 6.096 |
| $s$: 3 | ✗ 2.632 | ✗ 22.348 |
| $s$: 4 | ✗ 7.924 | ✗ 60.196 |
| $s$: 5 | ✗ 13.764 | ✗ 103.964 |
| $s$: 6 | ✗ 24.944 | ✗ 192.348 |
| $s$: 7 | ✗ 54.096 | ✗ 369.104 |
| $s$: 8 | ✗ 215.900 | ✗ 1779.830 |
| $s$: 9 | ✗ 1584.330 | > 3600.00 |
| $s$: 10 | > 3600.00 | > 3600.00 |
| $s$: 11 | > 3600.00 | > 3600.00 |
| $s$: 12 | > 3600.00 | > 3600.00 |
| $s$: 13 | > 3600.00 | > 3600.00 |
| $s$: 14 | > 3600.00 | > 3600.00 |

| | |
|---|---|
| 0. | `CJMP` 6 |
| 1. | `READ` $i$ |
| 2. | `CJMP` 4 |
| 3. | `READ` $r_0$ |
| 4. | `CJMP` 5 |
| 5. | `WRITE` $r_0$ |
| 6. | `WRITE` $o$ |
| 7. | `NEXT` |
| 8. | `READ` $i$ |

Figure 38: Experimental results: `mode-select.tlsf`

$$\square\left(u \rightarrow ((i \leftrightarrow o) \wedge (i \rightarrow \bigcirc(o\,\mathcal{W}\,u)) \;\wedge\; ((\neg i) \rightarrow \bigcirc((\neg o)\,\mathcal{W}\,u)))\right)$$

| Circuit | $l\colon 0$ | $l\colon 1$ | $l\colon 2$ | $l\colon 3$ |
|---|---|---|---|---|
| $g\colon 0$ | ✗ 0.008 | ✗ 0.020 | ✗ 0.072 | ✗ 0.420 |
| $g\colon 1$ | ✗ 0.012 | ✗ 0.028 | ✗ 0.092 | ✗ 1.796 |
| $g\colon 2$ | ✗ 0.016 | ✗ 0.040 | ✗ 0.120 | ✗ 1.664 |
| $g\colon 3$ | ✗ 0.024 | ✓ **0.072** | ✓ 0.180 | ✓ 1.156 |
| $g\colon 4$ | ✗ 0.032 | ✓ 0.092 | ✓ 0.272 | ✓ 0.996 |
| $g\colon 5$ | ✗ 0.048 | ✓ 0.144 | ✓ 0.380 | ✓ 1.524 |
| $g\colon 6$ | ✗ 0.068 | ✓ 0.212 | ✓ 0.548 | ✓ 1.564 |
| $g\colon 7$ | ✗ 0.096 | ✓ 0.280 | ✓ 0.584 | ✓ 2.424 |
| $g\colon 8$ | ✗ 0.128 | ✓ 0.324 | ✓ 0.696 | ✓ 2.024 |
| $g\colon 9$ | ✗ 0.160 | ✓ 0.420 | ✓ 0.852 | ✓ 2.724 |




| Register | $r\colon 0$ | $r\colon 1$ | $r\colon 2$ |
|---|---|---|---|
| $i\colon 1$ | ✗ 0.072 | ✗ 0.356 | ✗ 1.952 |
| $i\colon 2$ | ✗ 0.396 | ✗ 2.204 | ✗ 12.596 |
| $i\colon 3$ | ✗ 2.148 | ✗ 43.844 | > 3600.00 |
| $i\colon 4$ | ✗ 13.424 | ✗ 85.236 | > 3600.00 |
| $i\colon 5$ | ✗ 43.264 | > 3600.00 | > 3600.00 |
| $i\colon 6$ | ✓ **98.164** | > 3600.00 | > 3600.00 |
| $i\colon 7$ | > 3600.00 | > 3600.00 | > 3600.00 |
| $i\colon 8$ | > 3600.00 | > 3600.00 | > 3600.00 |
| $i\colon 9$ | > 3600.00 | > 3600.00 | > 3600.00 |

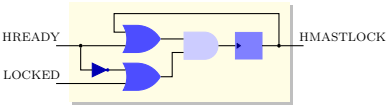
| Program | $v\colon 0$ | $v\colon 1$ |
|---|---|---|
| $s\colon 1$ | ✗ 0.128 | ✗ 1.004 |
| $s\colon 2$ | ✗ 1.372 | ✗ 10.768 |
| $s\colon 3$ | ✗ 5.656 | ✗ 44.164 |
| $s\colon 4$ | ✗ 14.972 | ✗ 109.240 |
| $s\colon 5$ | ✗ 26.020 | ✗ 191.928 |
| $s\colon 6$ | ✗ 51.704 | ✗ 363.316 |
| $s\colon 7$ | ✗ 156.724 | ✗ 913.732 |
| $s\colon 8$ | ✓ **373.256** | ✓ 1892.680 |
| $s\colon 9$ | ✓ 316.508 | > 3600.00 |
| $s\colon 10$ | ✓ 322.672 | > 3600.00 |
| $s\colon 11$ | > 3600.00 | > 3600.00 |
| $s\colon 12$ | > 3600.00 | > 3600.00 |
| $s\colon 13$ | > 3600.00 | > 3600.00 |
| $s\colon 14$ | > 3600.00 | > 3600.00 |

```
0.  READ u
1.  NOT
2.  CJMP 5
3.  READ i
4.  WRITE o
5.  NEXT
```

```
while (u) {
  o := i;
  inout
};
inout
```

Figure 39: Experimental results: `latch.tlsf`

| Circuit | $l: 0$ | $l: 1$ | $l: 2$ | $l: 3$ |
|---|---|---|---|---|
| $g: 0$ | ✗ 0.012 | ✗ 0.020 | ✗ 0.072 | ✗ 0.388 |
| $g: 1$ | ✗ 0.012 | ✗ 0.024 | ✗ 0.088 | ✗ 1.468 |
| $g: 2$ | ✗ 0.016 | ✗ 0.040 | ✗ 0.204 | ✗ 4.164 |
| $g: 3$ | ✗ 0.024 | ✓ **0.072** | ✓ 0.220 | ✓ 0.972 |
| $g: 4$ | ✗ 0.036 | ✓ 0.092 | ✓ 0.260 | ✓ 1.192 |
| $g: 5$ | ✗ 0.044 | ✓ 0.144 | ✓ 0.348 | ✓ 1.252 |
| $g: 6$ | ✗ 0.068 | ✓ 0.204 | ✓ 0.508 | ✓ 1.200 |
| $g: 7$ | ✗ 0.096 | ✓ 0.272 | ✓ 0.604 | ✓ 1.964 |
| $g: 8$ | ✗ 0.124 | ✓ 0.316 | ✓ 0.668 | ✓ 2.444 |
| $g: 9$ | ✗ 0.152 | ✓ 0.412 | ✓ 0.816 | ✓ 4.608 |



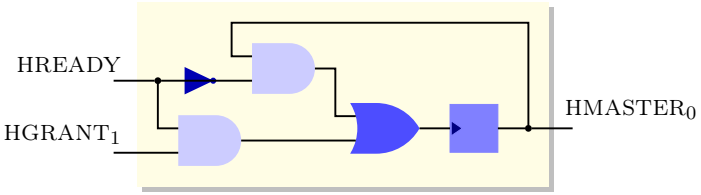

| Register | $r: 0$ | $r: 1$ | $r: 2$ |
|---|---|---|---|
| $i: 1$ | ✗ 0.068 | ✗ 0.340 | ✗ 1.804 |
| $i: 2$ | ✗ 0.388 | ✗ 6.588 | ✗ 469.296 |
| $i: 3$ | ✗ 1.128 | ✗ 28.072 | > 3600.00 |
| $i: 4$ | ✗ 5.920 | ✗ 128.112 | > 3600.00 |
| $i: 5$ | ✗ 72.108 | ✗ 818.700 | > 3600.00 |
| $i: 6$ | ✗ 510.632 | > 3600.00 | > 3600.00 |
| $i: 7$ | ✓ **3522.250** | > 3600.00 | > 3600.00 |
| $i: 8$ | > 3600.00 | > 3600.00 | > 3600.00 |
| $i: 9$ | > 3600.00 | > 3600.00 | > 3600.00 |

| Program | $v: 0$ | $v: 1$ |
|---|---|---|
| $s: 1$ | ✗ 0.124 | ✗ 1.000 |
| $s: 2$ | ✗ 1.300 | ✗ 10.188 |
| $s: 3$ | ✗ 4.932 | ✗ 34.384 |
| $s: 4$ | ✗ 13.548 | ✗ 100.020 |
| $s: 5$ | ✗ 22.940 | ✗ 163.292 |
| $s: 6$ | ✗ 41.216 | ✗ 282.980 |
| $s: 7$ | ✗ 123.576 | ✗ 817.008 |
| $s: 8$ | > 3600.00 | > 3600.00 |
| $s: 9$ | > 3600.00 | > 3600.00 |
| $s: 10$ | > 3600.00 | > 3600.00 |
| $s: 11$ | > 3600.00 | > 3600.00 |
| $s: 12$ | > 3600.00 | > 3600.00 |
| $s: 13$ | > 3600.00 | > 3600.00 |
| $s: 14$ | > 3600.00 | > 3600.00 |

```
0. JMP 4
1. AND LOCKED
2. NEXT
3. WRITE HMASTLOCK
4. READ HREADY
5. CJMP 1
6. NEXT
```

Figure 40: Experimental results: `amba-decomposed-shift.tlsf`

| Circuit | $l$: 0 | $l$: 1 | $l$: 2 | $l$: 3 |
|---|---|---|---|---|
| $g$: 0 | ✗ 0.032 | ✗ 0.028 | ✗ 0.100 | ✗ 0.484 |
| $g$: 1 | ✗ 0.020 | ✗ 0.048 | ✗ 0.136 | ✗ 2.156 |
| $g$: 2 | ✗ 0.032 | ✗ 0.068 | ✗ 0.280 | ✗ 5.464 |
| $g$: 3 | ✗ 0.040 | ✓ **0.116** | ✓ 0.356 | ✓ 1.484 |
| $g$: 4 | ✗ 0.100 | ✓ 0.208 | ✓ 0.536 | ✓ 1.648 |
| $g$: 5 | ✗ 0.112 | ✓ 0.260 | ✓ 0.792 | ✓ 4.640 |
| $g$: 6 | ✗ 0.120 | ✓ 0.400 | ✓ 1.008 | ✓ 2.672 |
| $g$: 7 | ✗ 0.156 | ✓ 0.472 | ✓ 1.128 | ✓ 3.868 |
| $g$: 8 | ✗ 0.200 | ✓ 0.592 | ✓ 1.352 | ✓ 4.076 |
| $g$: 9 | ✗ 0.248 | ✓ 0.724 | ✓ 1.656 | ✓ 5.492 |

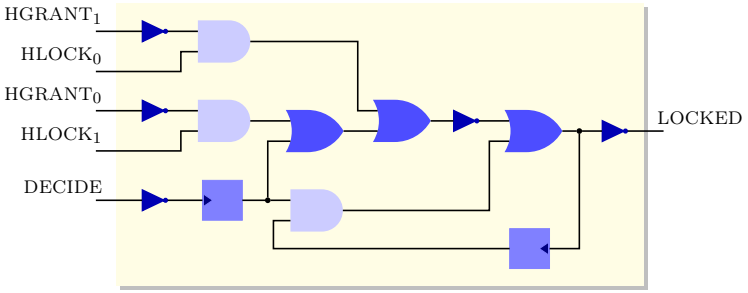



| Register | $r$: 0 | $r$: 1 | $r$: 2 |
|---|---|---|---|
| $i$: 1 | ✗ 0.088 | ✗ 0.400 | ✗ 2.044 |
| $i$: 2 | ✗ 0.492 | ✗ 2.368 | ✗ 12.936 |
| $i$: 3 | ✗ 1.284 | ✗ 25.700 | > 3600.00 |
| $i$: 4 | ✗ 12.712 | ✗ 76.216 | > 3600.00 |
| $i$: 5 | ✗ 54.464 | > 3600.00 | > 3600.00 |
| $i$: 6 | ✗ 1530.820 | > 3600.00 | > 3600.00 |
| $i$: 7 | > 3600.00 | > 3600.00 | > 3600.00 |
| $i$: 8 | > 3600.00 | > 3600.00 | > 3600.00 |
| $i$: 9 | > 3600.00 | > 3600.00 | > 3600.00 |

| Program | $v$: 0 | $v$: 1 |
|---|---|---|
| $s$: 1 | ✗ 0.140 | ✗ 1.096 |
| $s$: 2 | ✗ 1.376 | ✗ 10.484 |
| $s$: 3 | ✗ 5.256 | ✗ 35.688 |
| $s$: 4 | ✗ 13.920 | ✗ 98.784 |
| $s$: 5 | ✗ 24.656 | ✗ 177.564 |
| $s$: 6 | ✗ 104.340 | ✗ 322.668 |
| $s$: 7 | ✗ 117.836 | ✗ 819.648 |
| $s$: 8 | ✗ 820.184 | > 3600.00 |
| $s$: 9 | > 3600.00 | > 3600.00 |
| $s$: 10 | > 3600.00 | > 3600.00 |
| $s$: 11 | > 3600.00 | > 3600.00 |
| $s$: 12 | > 3600.00 | > 3600.00 |
| $s$: 13 | > 3600.00 | > 3600.00 |
| $s$: 14 | > 3600.00 | > 3600.00 |

Figure 41: Experimental results: `amba-decomposed-encode-2.tlsf`

| Circuit | $l$: 0 | $l$: 1 | $l$: 2 | $l$: 3 |
|---|---|---|---|---|
| $g$: 0 | ✗ 0.028 | ✗ 0.100 | ✗ 0.444 | ✗ 2.584 |
| $g$: 1 | ✗ 0.068 | ✗ 0.176 | ✗ 0.796 | ✗ 5.296 |
| $g$: 2 | ✗ 0.084 | ✗ 0.348 | ✗ 1.364 | ✗ 9.744 |
| $g$: 3 | ✗ 0.156 | ✗ 0.544 | ✗ 14.140 | ✗ 62.916 |
| $g$: 4 | ✗ 0.228 | ✗ 0.824 | ✗ 736.000 | ✗ 529.624 |
| $g$: 5 | ✗ 0.324 | ✗ 1.136 | > 3600.00 | > 3600.00 |
| $g$: 6 | ✗ 0.484 | ✗ 1.568 | ✓ **3326.420** | > 3600.00 |
| $g$: 7 | ✗ 0.564 | ✗ 2.080 | ✓ 2913.820 | > 3600.00 |
| $g$: 8 | ✗ 0.692 | ✗ 2.628 | ✓ 13.940 | ✓ 72.376 |
| $g$: 9 | ✗ 0.900 | ✗ 3.404 | ✓ 631.904 | ✓ 89.176 |

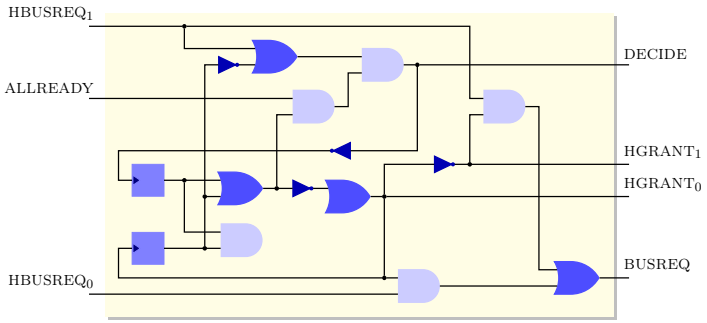



| Register | $r$: 0 | $r$: 1 | $r$: 2 |
|---|---|---|---|
| $i$: 1 | ✗ 0.408 | ✗ 1.968 | ✗ 9.920 |
| $i$: 2 | ✗ 2.100 | ✗ 14.560 | ✗ 165.052 |
| $i$: 3 | ✗ 8.368 | ✗ 193.892 | > 3600.00 |
| $i$: 4 | ✗ 26.724 | > 3600.00 | > 3600.00 |
| $i$: 5 | ✗ 542.252 | > 3600.00 | > 3600.00 |
| $i$: 6 | > 3600.00 | > 3600.00 | > 3600.00 |
| $i$: 7 | > 3600.00 | > 3600.00 | > 3600.00 |
| $i$: 8 | > 3600.00 | > 3600.00 | > 3600.00 |
| $i$: 9 | > 3600.00 | > 3600.00 | > 3600.00 |

| Program | $v$: 0 | $v$: 1 |
|---|---|---|
| $s$: 1 | ✗ 0.704 | ✗ 5.520 |
| $s$: 2 | ✗ 6.936 | ✗ 49.812 |
| $s$: 3 | ✗ 22.876 | ✗ 170.192 |
| $s$: 4 | ✗ 65.268 | ✗ 450.872 |
| $s$: 5 | ✗ 117.288 | ✗ 875.352 |
| $s$: 6 | ✗ 224.064 | ✗ 1842.850 |
| $s$: 7 | ✗ 401.484 | ✗ 3469.140 |
| $s$: 8 | > 3600.00 | > 3600.00 |
| $s$: 9 | > 3600.00 | > 3600.00 |
| $s$: 10 | > 3600.00 | > 3600.00 |
| $s$: 11 | > 3600.00 | > 3600.00 |
| $s$: 12 | > 3600.00 | > 3600.00 |
| $s$: 13 | > 3600.00 | > 3600.00 |
| $s$: 14 | > 3600.00 | > 3600.00 |

Figure 42: Experimental results: `amba-decomposed-lock-2.tlsf`

| Circuit | $l$: 0 | $l$: 1 | $l$: 2 | $l$: 3 |
|---|---|---|---|---|
| $g$: 0 | ✗ 0.064 | ✗ 0.128 | ✗ 0.404 | ✗ 4.632 |
| $g$: 1 | ✗ 0.072 | ✗ 0.144 | ✗ 0.472 | ✗ 10.880 |
| $g$: 2 | ✗ 0.084 | ✗ 0.176 | ✗ 0.568 | ✗ 28.088 |
| $g$: 3 | ✗ 0.096 | ✗ 0.212 | ✗ 0.896 | ✗ 667.596 |
| $g$: 4 | ✗ 0.116 | ✗ 0.288 | ✗ 6.168 | > 3600.00 |
| $g$: 5 | ✗ 0.152 | ✗ 0.372 | ✗ 84.656 | > 3600.00 |
| $g$: 6 | ✗ 0.192 | ✗ 0.468 | > 3600.00 | > 3600.00 |
| $g$: 7 | ✗ 0.232 | ✗ 0.548 | > 3600.00 | > 3600.00 |
| $g$: 8 | ✗ 0.280 | ✗ 0.640 | > 3600.00 | > 3600.00 |
| $g$: 9 | ✗ 0.328 | ✗ 0.800 | ✓ **208.080** | ✓ 2568.810 |

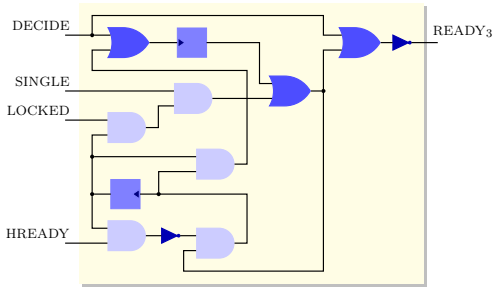



| Register | $r$: 0 | $r$: 1 | $r$: 2 |
|---|---|---|---|
| $i$: 1 | ✗ 45.924 | ✗ 284.016 | ✗ 1772.960 |
| $i$: 2 | > 3600.00 | > 3600.00 | > 3600.00 |
| $i$: 3 | > 3600.00 | > 3600.00 | > 3600.00 |
| $i$: 4 | > 3600.00 | > 3600.00 | > 3600.00 |
| $i$: 5 | > 3600.00 | > 3600.00 | > 3600.00 |
| $i$: 6 | > 3600.00 | > 3600.00 | > 3600.00 |
| $i$: 7 | > 3600.00 | > 3600.00 | > 3600.00 |
| $i$: 8 | > 3600.00 | > 3600.00 | > 3600.00 |
| $i$: 9 | > 3600.00 | > 3600.00 | > 3600.00 |

| Program | $v$: 0 | $v$: 1 |
|---|---|---|
| $s$: 1 | ✗ 324.860 | ✗ 2470.350 |
| $s$: 2 | ✗ 2739.730 | > 3600.00 |
| $s$: 3 | > 3600.00 | > 3600.00 |
| $s$: 4 | > 3600.00 | > 3600.00 |
| $s$: 5 | > 3600.00 | > 3600.00 |
| $s$: 6 | > 3600.00 | > 3600.00 |
| $s$: 7 | > 3600.00 | > 3600.00 |
| $s$: 8 | > 3600.00 | > 3600.00 |
| $s$: 9 | > 3600.00 | > 3600.00 |
| $s$: 10 | > 3600.00 | > 3600.00 |
| $s$: 11 | > 3600.00 | > 3600.00 |
| $s$: 12 | > 3600.00 | > 3600.00 |
| $s$: 13 | > 3600.00 | > 3600.00 |
| $s$: 14 | > 3600.00 | > 3600.00 |

Figure 43: Experimental results: `amba-decomposed-arbiter-2.tlsf`

| Circuit | $l$: 0 | $l$: 1 | $l$: 2 | $l$: 3 |
|---|---|---|---|---|
| $g$: 0 | ✗ 0.040 | ✗ 0.144 | ✗ 0.768 | ✗ 6.908 |
| $g$: 1 | ✗ 0.056 | ✗ 0.176 | ✗ 1.400 | ✗ 200.972 |
| $g$: 2 | ✗ 0.076 | ✗ 0.232 | ✗ 1.584 | ✗ 3312.120 |
| $g$: 3 | ✗ 0.108 | ✗ 0.332 | ✗ 2.144 | ✗ 2687.370 |
| $g$: 4 | ✗ 0.192 | ✗ 0.444 | ✗ 6.792 | > 3600.00 |
| $g$: 5 | ✗ 0.228 | ✗ 0.600 | ✗ 273.868 | > 3600.00 |
| $g$: 6 | ✗ 0.312 | ✗ 0.748 | > 3600.00 | > 3600.00 |
| $g$: 7 | ✗ 0.392 | ✗ 0.968 | > 3600.00 | > 3600.00 |
| $g$: 8 | ✗ 0.488 | ✗ 1.176 | ✓ **298.532** | ✓ 3134.000 |
| $g$: 9 | ✗ 0.624 | ✗ 1.452 | ✓ 658.608 | > 3600.00 |

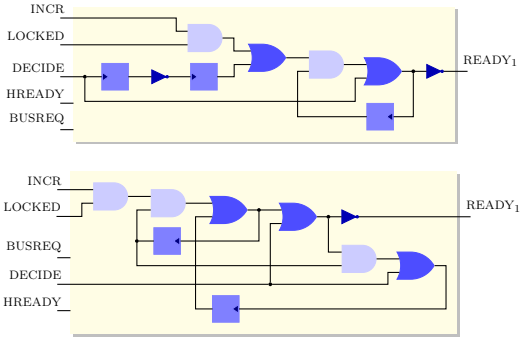



| Register | $r$: 0 | $r$: 1 | $r$: 2 |
|---|---|---|---|
| $i$: 1 | ✗ 0.604 | ✗ 3.384 | ✗ 17.548 |
| $i$: 2 | ✗ 3.912 | ✗ 19.912 | ✗ 118.316 |
| $i$: 3 | ✗ 14.544 | ✗ 269.344 | > 3600.00 |
| $i$: 4 | > 3600.00 | ✗ 508.012 | ✗ 1135.320 |
| $i$: 5 | > 3600.00 | > 3600.00 | > 3600.00 |
| $i$: 6 | > 3600.00 | > 3600.00 | > 3600.00 |
| $i$: 7 | > 3600.00 | > 3600.00 | > 3600.00 |
| $i$: 8 | > 3600.00 | > 3600.00 | > 3600.00 |
| $i$: 9 | > 3600.00 | > 3600.00 | > 3600.00 |

| Program | $v$: 0 | $v$: 1 |
|---|---|---|
| $s$: 1 | ✗ 1.948 | ✗ 14.364 |
| $s$: 2 | ✗ 17.652 | ✗ 131.992 |
| $s$: 3 | ✗ 72.656 | ✗ 545.160 |
| $s$: 4 | ✗ 162.400 | ✗ 1282.370 |
| $s$: 5 | ✗ 275.156 | ✗ 2320.480 |
| $s$: 6 | ✗ 645.356 | > 3600.00 |
| $s$: 7 | ✗ 965.856 | > 3600.00 |
| $s$: 8 | > 3600.00 | > 3600.00 |
| $s$: 9 | > 3600.00 | > 3600.00 |
| $s$: 10 | > 3600.00 | > 3600.00 |
| $s$: 11 | > 3600.00 | > 3600.00 |
| $s$: 12 | > 3600.00 | > 3600.00 |
| $s$: 13 | > 3600.00 | > 3600.00 |
| $s$: 14 | > 3600.00 | > 3600.00 |

Figure 44: Experimental results: `amba-decomposed-tsingle.tlsf`

| Circuit | $l$: 0 | $l$: 1 | $l$: 2 | $l$: 3 |
|---|---|---|---|---|
| $g$: 0 | ✗ 0.076 | ✗ 0.312 | ✗ 1.944 | ✗ 63.300 |
| $g$: 1 | ✗ 0.104 | ✗ 0.396 | ✗ 2.656 | ✗ 78.748 |
| $g$: 2 | ✗ 0.148 | ✗ 0.584 | ✗ 3.324 | ✗ 1563.120 |
| $g$: 3 | ✗ 0.244 | ✗ 0.812 | ✗ 4.032 | ✗ 1803.360 |
| $g$: 4 | ✗ 0.344 | ✗ 1.068 | ✗ 20.596 | ✓ **1485.750** |
| $g$: 5 | ✗ 0.488 | ✗ 1.384 | ✗ 3321.820 | ✓ 251.960 |
| $g$: 6 | ✗ 0.632 | ✗ 1.836 | ✓ **68.796** | ✓ 1025.180 |
| $g$: 7 | ✗ 0.812 | ✗ 2.340 | ✓ 13.944 | ✓ 165.168 |
| $g$: 8 | ✗ 1.048 | ✗ 2.908 | ✓ 23.984 | ✓ 440.676 |
| $g$: 9 | ✗ 1.328 | ✗ 3.744 | ✓ 28.540 | ✓ 1036.250 |




| Register | $r$: 0 | $r$: 1 | $r$: 2 |
|---|---|---|---|
| $i$: 1 | ✗ 1.384 | ✗ 7.448 | ✗ 39.852 |
| $i$: 2 | ✗ 9.080 | ✗ 47.636 | ✗ 253.580 |
| $i$: 3 | > 3600.00 | > 3600.00 | > 3600.00 |
| $i$: 4 | ✗ 589.084 | > 3600.00 | > 3600.00 |
| $i$: 5 | > 3600.00 | > 3600.00 | > 3600.00 |
| $i$: 6 | > 3600.00 | > 3600.00 | > 3600.00 |
| $i$: 7 | > 3600.00 | > 3600.00 | > 3600.00 |
| $i$: 8 | > 3600.00 | > 3600.00 | > 3600.00 |
| $i$: 9 | > 3600.00 | > 3600.00 | > 3600.00 |

| Program | $v$: 0 | $v$: 1 |
|---|---|---|
| $s$: 1 | ✗ 4.572 | ✗ 32.312 |
| $s$: 2 | ✗ 38.500 | ✗ 297.596 |
| $s$: 3 | ✗ 121.776 | ✗ 1052.820 |
| $s$: 4 | ✗ 346.228 | ✗ 2775.920 |
| $s$: 5 | ✗ 684.756 | > 3600.00 |
| $s$: 6 | ✗ 1193.500 | > 3600.00 |
| $s$: 7 | ✗ 2630.240 | > 3600.00 |
| $s$: 8 | > 3600.00 | > 3600.00 |
| $s$: 9 | > 3600.00 | > 3600.00 |
| $s$: 10 | > 3600.00 | > 3600.00 |
| $s$: 11 | > 3600.00 | > 3600.00 |
| $s$: 12 | > 3600.00 | > 3600.00 |
| $s$: 13 | > 3600.00 | > 3600.00 |
| $s$: 14 | > 3600.00 | > 3600.00 |

Figure 45: Experimental results: `amba-decomposed-tincr.tlsf`

| Circuit | $l$: 0 | $l$: 1 | $l$: 2 | $l$: 3 |
|---|---|---|---|---|
| $g$: 0 | ✗ 0.060 | ✗ 0.192 | ✗ 1.204 | ✗ 21.220 |
| $g$: 1 | ✗ 0.076 | ✗ 0.224 | ✗ 1.708 | ✗ 68.492 |
| $g$: 2 | ✗ 0.092 | ✗ 0.280 | ✗ 2.228 | ✗ 3460.200 |
| $g$: 3 | ✗ 0.128 | ✗ 0.384 | ✗ 1.956 | ✗ 3134.680 |
| $g$: 4 | ✗ 0.184 | ✗ 0.520 | ✗ 2.748 | ✗ 7983.400 |
| $g$: 5 | ✗ 0.244 | ✗ 0.676 | ✗ 3.028 | > 172800.000 |
| $g$: 6 | ✗ 0.380 | ✗ 0.820 | ✗ 3.344 | > 172800.000 |
| $g$: 7 | ✗ 0.472 | ✗ 1.076 | ✗ 3.464 | > 172800.000 |
| $g$: 8 | ✗ 0.508 | ✗ 1.224 | ✗ 4.908 | > 172800.000 |
| $g$: 9 | ✗ 0.636 | ✗ 1.552 | ✗ 5.080 | > 172800.000 |

| Register | $r$: 0 | $r$: 1 | $r$: 2 |
|---|---|---|---|
| $i$: 1 | ✗ 0.752 | ✗ 4.088 | ✗ 22.536 |
| $i$: 2 | ✗ 27.048 | ✗ 2575.550 | > 3600.00 |
| $i$: 3 | ✗ 748.368 | > 3600.00 | > 3600.00 |
| $i$: 4 | > 3600.00 | > 3600.00 | > 3600.00 |
| $i$: 5 | > 3600.00 | > 3600.00 | > 3600.00 |
| $i$: 6 | > 3600.00 | > 3600.00 | > 3600.00 |
| $i$: 7 | > 3600.00 | > 3600.00 | > 3600.00 |
| $i$: 8 | > 3600.00 | > 3600.00 | > 3600.00 |
| $i$: 9 | > 3600.00 | > 3600.00 | > 3600.00 |

| Program | $v$: 0 | $v$: 1 |
|---|---|---|
| $s$: 1 | ✗ 2.492 | ✗ 19.324 |
| $s$: 2 | ✗ 23.164 | ✗ 184.428 |
| $s$: 3 | ✗ 99.180 | ✗ 779.950 |
| $s$: 4 | ✗ 218.288 | ✗ 1869.600 |
| $s$: 5 | ✗ 373.068 | ✗ 2842.880 |
| $s$: 6 | ✗ 787.368 | > 3600.00 |
| $s$: 7 | ✗ 1475.710 | > 3600.00 |
| $s$: 8 | > 3600.00 | > 3600.00 |
| $s$: 9 | > 3600.00 | > 3600.00 |
| $s$: 10 | > 3600.00 | > 3600.00 |
| $s$: 11 | > 3600.00 | > 3600.00 |
| $s$: 12 | > 3600.00 | > 3600.00 |
| $s$: 13 | > 3600.00 | > 3600.00 |
| $s$: 14 | > 3600.00 | > 3600.00 |

Figure 46: Experimental results: `amba-decomposed-tburst4.tlsf`

# 5 Discussion

As our analysis reveals, the presented output sensitive synthesis methods indeed enable the creation of better quality solutions from a logic specification with respect to the given quality metrics. The produced solutions are well structured and easy to inspect by a human being. Accordingly, they also help developers in order to verify, whether the created solutions indeed meet their design intents or, whether the specification still misses some important functional quality constraints. On the contrary, the results also reveal that producing high quality solutions comes at the price of higher synthesis times, which, however, is not that much of a big surprise.

The more intriguing question is: What kind of quality metrics are preferable to be useful and adequate for the system design? Clearly, the question cannot be answered in general, since the corresponding answer always depends on the respective development environment and the final application requirements. Hence, whether a circuit, a register program or a reactive program is the preferred model of choice always depends on the developers expertise and how easy these models can be translated into the final application context.

To this end, we also introduced a more general structural measure that works independently of the final evaluation model, as given by the number of simple cycles of the underlying Mealy machine. This metric is not biased towards a specific implementation type, but directly targets the structural complexity of the solution instead. That the number of simple cycles is linked with the structural complexity is underlined by the explosive nature of the metric, as proven theoretically, as well as through our practical analyses, which shows that a bound on the number of cycles does neither introduce much overhead with respect to the synthesis times, nor is a large number of cycles required for most systems in general.

How relevant the collected insights for practical aplications are in the end still needs to be determined through further experiments. This however requires more systems that are designed based on a logic description in the first place, which only happens if the corresponding development tools get strong enough to provide all of the required insights fast and in a straightforward fashion. We hope that with the previously presented quality enforcing synthesis tools we are able to get a step closer towards this goal.

# Chapter V

# Delay Games

So far, we only considered techniques that turn realizable specifications into system implementations. Therefore, we learned that synthesis concerns the control, while the way data is represented must be kept abstract, and that output sensitive methods provide advantages against game based synthesis, since they allow to choose from multiple realizations according to the preferred output requirements. However, during the development of such specifications it also happens that designers introduce inconsistencies causing an unrealizability result. The developers then first need to refine the specification by correcting the introduced inconsistency. Beforehand, however, the accidentally introduced inconsistency must be found and understood by it's creator. According to these concerns, our introduced synthesis engine is not sufficient yet to provide the developer with the required feedback for such situations.

First insights into the cause of unrealizability may be revealed by the counter-strategy of the environment, which takes advantage of the developer's fault in order to win against every possible system implementation. By determinacy of the underlying game such a counter-strategy always must exist [102]. Nevertheless, especially for simple inconsistencies, the counter-strategy often just guides the system to the point in time, where the fault of the developer takes place and then behaves arbitrarily. Hence, for getting a better understanding of the introduced fault, the strategy is only of a little help at this point. In order to identify such inconsistencies directly, we instead need a better understanding of how it could even happen that the developer has introduced the error in the first place.

It turns out, that most of the *overseen* causes for unrealizability are introduced by miss-understandings of temporal dependencies between different specified tasks, which cannot be realized simultaneously. As an example, consider an update that needs to be executed whenever a button gets pressed, which is indicated by a Boolean input signal switching from low (`false`) to high (`true`). A first specification attempt of the designer may look as follows:

$$\square\big(\neg\texttt{button} \wedge \bigcirc\texttt{button} \leftrightarrow [\,\texttt{o} \leftarrowtail \texttt{f o}\,]\big)$$

The specification is unrealizable. The problem is that checking the button for

being pressed requires the system to compare the current input value with the previous one. The update then must be triggered after this comparison has taken place. Thus, the fault in the specification is that the update is required *too early*. The system has yet not observed whether the input has changed at this point in time. The solution is to delay the update by a single time step. Then the system is able to gather all the required information in time:

$$\square\big(\neg\texttt{button} \wedge \bigcirc\texttt{button} \;\leftrightarrow\; \bigcirc[\,\texttt{o} \leftarrowtail \texttt{f o}\,]\big)$$

The requirement for such simple delay operations is a regular cause of unrealizability. However, with the techniques presented so far they can only be detected from a faulty specification after an exhausting in depth inspection through the developer. Instead, souldn't it be much more convenient, if we could check a faulty specification against this type of erroneous behavior automatically?

In this last chapter of the thesis we considers a solution to this problem using *delay games*, which offer an extension to the classical synthesis framework for reactive systems. In delay games, the requirement of a strict alternation between both players is relaxed, implying that one of the players can postpone her moves to obtain a lookahead on her opponent's moves. With lookahead at hand, the system player is able to win games that she would loose otherwise. Delay games provide a perfect solution for finding causes of unrealizability in faulty specifications that are introduced through too strict timing requirements imposed by the developer. Delay games automatically relax such requirements leading to realizability under the assumption of delay. Therefore, developers not only get insights into the cause of the initial error, but with a realizing system strategy at hand, which wins under the assumption of delay, they also get feedback for the required adaptions needed to resolve the issue in the original specification.

Considering the problem from a more theoretical position, delay games also can be expressed as uniformization of relations by continuous functions [137, 140, 62, 136]. The games are played between the system and the environment, each picking letters from alphabets $\Sigma_I$ and $\Sigma_O$, respectively. As a result, they produce two infinite sequences $\alpha$ and $\beta$, leading to a strategy for the system player realized as a mapping $\tau\colon \Sigma_I^\omega \to \Sigma_O^\omega$. The strategy is winning for the system if $\alpha$ and $\tau(\alpha)$ are related by the winning condition Win $\subseteq \Sigma_I^\omega \times \Sigma_O^\omega$ for every $\alpha \in \Sigma_I^\omega$. In this case we say that $\tau$ uniformizes $L$. In the classical setting, where the system and the environment pick letters in alternation, the $n$-th letter of $\tau(\alpha)$ depends only on the first $n$ letters of $\alpha$. Therefore, strategies with bounded lookahead, *i.e.*, only finitely many moves

are postponed, induce Lipschitz-continuous functions $\tau$ in the Cantor topology on $\Sigma^\omega$. In contrast, strategies with unbounded lookahead induce continuous functions, or uniformly continuous functions, due to $\Sigma^\omega$ being compact [62].

**Related Work.** In 1972, Hosch and Landweber first proved decidability of winning delay games with $\omega$-regular winning conditions, where one of the players is able to utilize the lookahead [64]. Much later, in 2012 the problem then was revisited again by Holtmann, Kaiser, and Thomas, which showed that there is also a doubly-exponential upper bound on the lookahead, *i.e.*, if the system player wins with bounded lookahead, then double-exponential lookahead is already sufficient [62]. Furthermore, their decidability proof provided the first algorithm for delay games of doubly-exponential running time. Their results show that the delaying player does not take any advantages from having unbounded lookahead, bounded lookahead is always sufficient.

Considering winning conditions beyond $\omega$-regularity, as in delay games with context-free winning conditions, leads to undecidability in general and non-elementary lower bounds on the required lookahead, even if only very weak fragments are considered [47]. The analogue of the Hosch-Landweber theorem can be proven, however, for another class of winning conditions beyond $\omega$-regularity: if the winning condition is definable in weak monadic second order logic with the unbounding quantifier (WMSO+U), then determining whether a player wins a delay game with bounded lookahead is decidable [155]. Doubly-exponential lookahead is also sufficient for WMSO+U conditions, under the assumption that the delaying player wins with bounded lookahead in the first place. In general, however, bounded lookahead does not suffice, *i.e.*, the analogue of the Holtmann-Kaiser-Thomas theorem does not hold for delay games with WMSO+U conditions. There exists a corresponding game that the system player only wins with unbounded lookahead, no matter how slowly the lookahead grows [154].

Another variant are delay games with winning conditions in Prompt-LTL [89]: a logic that extends LTL [116] with temporal operators whose scope is bounded in time. Solving delay games that use Prompt-LTL winning conditions is 3ExpTime-complete. Furthermore, triply-exponentially bounded lookahead is necessary and always sufficient [81]. Moreover, all lower bounds already hold for the special case of LTL.

Furthermore, it has been shown that every delay game with Borel winning condition is determined. Delay games with Borel winning conditions can be reduced to delay-free games with Borel winning conditions, while preserving the winning strategies of the players [79].

In terms of uniformization, decidability of the uniformization problem

for $\omega$-regular relations by Lipschitz-continuous functions has been proven by Hosch and Landweber. On the other hand, the equivalence of the existence of continuous uniformization functions and the existence of Lipschitz-continuous uniformization functions for $\omega$-regular relations has been proven by Holtmann et al., hence, the uniformization of context-free relations has been proven to be undecidable, even in the context of Lipschitz-continuous functions. However, the uniformization of WMSO+U relations by Lipschitz-continuous functions is decidable.

In another line of work, the case of finite words has been considered by Carayol and Löding [22], while Löding and Winter considered the case of finite trees [98]. Moreover, the uniformization on infinite binary trees [21, 57] fails due to the non-existence of MSO-definable choice functions.

Considering the representation of strategies in delay games, they are also more involved than in the classical setting, since strategies need to take the utilized delay into account. As it turns out, finding a suitable and compact model for such strategies is a non-trivial task. Accordingly, different notions for encoding delay as part of the strategy and the resulting differences in their expressivity have been analyzed [79]. Furthermore, by setting the focus on a general representation of finite-strategies for delay games, some of the known solving techniques for delay games have been revisited [156].

Finally, first experimental evaluations of an on-the-fly algorithm for the construction of strategies in delay games with safety objectives has been presented in [26].

# 5 Games with Delay

Delay games are two-player games of infinite duration in which the system player may delay her moves to obtain a lookahead on the environment players' moves. The amount of delay that is introduced into the system is given by a delay function $f$ determining at each point in time the lookahead of the system player on the environment players' moves.

**Definition 26.** A *delay function* is a mapping $f\colon \mathbb{N} \to \mathbb{N}^+$, which is said to be *constant*, if $f(i) = 1$ for every $i > 0$. The delay that is accumulated over time is captured by the *accumulating delay function* with $f_\Sigma(n) = \sum_{i=0}^{n} f(i)$ for all $n \in \mathbb{N}$.

Note that constant delay functions only introduce some initial delay once. However, by the again strict alternation of the players after this initial delay,

the system player always has a constant lookahead on her opponent's moves. Furthermore, consider that by the introduced delay, the resulting strategies for the players are assymmetric according to their moves. The environment player now needs to pick words of the length of the assigned delay.

**Definition 27.** In a delay game, a strategy for Player $O$ is a function $\sigma_O \colon (\Sigma_I)^* \to \Sigma_o$.

In contrast, the system still responses with single letters. The input and output history, thus, does not grow simultaneously. However, in the infinite both still produce infinite words.

**Definition 28.** In a delay game, a strategy for Player $I$ is a function $\sigma_I \colon (\Sigma_O)^* \to (\Sigma_I)^*$.

The standard notions of plays and consistency then are lifted to the adapted strategy definitions in a natural way.

**Definition 29.** Let $\rho$ be a play in an arena $\mathcal{A}$ with $\rho_m = (v_m^I, i_m, v_m^O, o_m)$ for all $m \in \mathbb{N}$, then $\rho$ is *consistent* with a strategy $\sigma_O$ of Player $O$ and a delay function $f$ iff for all $n \in \mathbb{N}$

$$\sigma_O(i_0 i_1 \dots i_{f_\Sigma(n)-1}) = o_n$$

A play $\rho$ is *consistent* with a strategy $\sigma_I$ of Player $I$ and a delay function $f$ iff for all $n \in \mathbb{N}^+$

- $\sigma_I(\varepsilon) = i_0 i_1 \dots i_{f(0)-1}$, and
- $\sigma_I(o_0 o_1 \dots o_{n-1}) = i_{f_\Sigma(n-1)} i_{f_\Sigma(n-1)+1} \dots i_{f_\Sigma(n)-1}$.

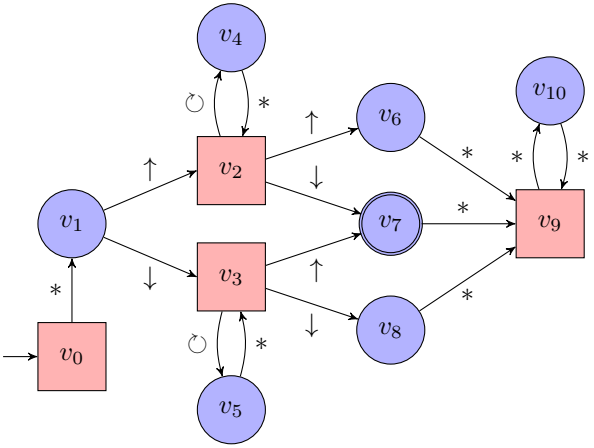
The set of all plays that are consistent with a strategy $\sigma_P$ and a delay function $f$ is denoted by $\mathrm{Plays}(A, f, \sigma_P)$.

Delay games extend standard games with the delay function $f$, which determines the lookahead of the system in every step.

**Definition 30.** A *delay game* $\Gamma_f(\mathcal{G})$ is a game $\mathcal{G} = (\mathcal{A}, \mathrm{Win})$ that is additionally equipped with a delay function $f$. A strategy $\sigma_O$ is *winning* for Player $O$ iff all plays consistent with $\sigma_O$ and $f$ are winning, *i.e.*, iff $\mathrm{Plays}(\mathcal{A}, f, \sigma_O) \subseteq \mathrm{Win}$. A strategy $\sigma_I$ is *winning* for Player $I$ iff $\mathrm{Plays}(\mathcal{A}, f, \sigma_O) \cap \mathrm{Win} = \emptyset$. A delay game is won by Player $P$ if there is a winning strategy $\sigma_P$ for Player $P$.

**Examples.**

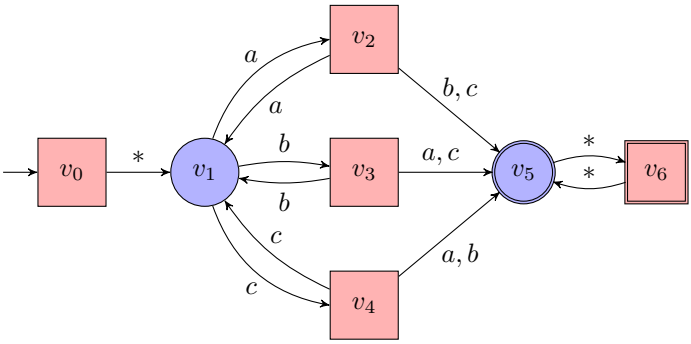
1.




Consider the safety game $\mathcal{G}_1$ depicted above over $\Sigma_I = \{\circlearrowleft, \downarrow, \uparrow\}$ and $\Sigma_O = \{\downarrow, \uparrow\}$ with the only unsafe vertex $v_7$ marked by double borders. Intuitively, Player $O$ wins, if the letter she picks in the first round is equal to the first letter other than $\circlearrowleft$ that Player $I$ picks. Also, Player $O$ wins, if there is no such letter.

We claim that Player $I$ wins $\Gamma_f(\mathcal{G}_1)$ for every delay function $f$: Player $I$ picks $\circlearrowleft^{f(0)}$ in the first round and assume Player $O$ picks $\downarrow$ afterwards (the case where she picks $\uparrow$ is dual). Then, Player $I$ picks a word starting with $\uparrow$ in the second round. The resulting play is winning for Player $I$ no matter how it is continued. Thus, Player $I$ has a winning strategy in $\Gamma_f(\mathcal{G}_1)$.

2.




Next, consider the safety $\mathcal{G}_2$ depicted on top of this page with $\Sigma_I = \{a, b, c\}$ and $\Sigma_O = \{a, b, c\}$, where Player $O$ wins if the input is shifted two positions to the left on the corresponding outcome of $\mathcal{G}_2$.

Player $O$ has a winning strategy for $\Gamma_f(\mathcal{G}_2)$ for every $f$ with $f(0) \geq 3$. In this case, Player $O$ has at least three letters lookahead in each round, which suffices to shift the input of Player $I$ two positions to the left. On the other hand, if $f(0) < 3$, then Player $I$ has a winning strategy, since Player $O$ has to pick the first response before the first three letters have been picked by Player $I$.

# 6 Computational Complexity

## 6.1 Parity Games

We start by presenting a method for solving delay games with parity winning conditions, which runs in exponential time in the size of the game arena. Our results also imply upper bounds on the solving time for delay games with safety or reachability conditions. A similar result already has been established by [62], however their upper bounds on the necessary lookahead are double-exponential, while our construction yields a single exponential upper bound.

**Theorem 21.** *The following problem is in* ExpTime*: Given a parity game $\mathcal{G}_\Omega = (\mathcal{A}, \text{Parity}(\Omega))$, does Player $O$ win $\Gamma_f(\mathcal{G}_\Omega)$ for some delay function $f$?*

We proceed by constructing an exponentially-sized, delay-free parity game with the same number of colors as $\mathcal{G}_\Omega$, which is won by Player $O$ if and only if she wins $\Gamma_f(\mathcal{G}_\Omega)$ for some delay function $f$. Intuitively, we assign to each potential lookahead $w \in \Sigma_I^*$ the behavior it induces on $\mathcal{G}_\Omega$, which is given by a function:

$$r\colon V_I \to 2^{V_I \times \Omega(V_I \cup V_O)}$$

If $(v', c) \in r(v)$, then there is a path from $v$ to $v'$ with maximal color $c$ on a word over $\Sigma_I \times \Sigma_O$ whose projection to $\Sigma_I$ is $w$. Having the same behavior gives rise to an equivalence relation over $\Sigma_I^*$ of exponential index. In the parity game that we construct, Player $I$ picks equivalence classes of this relation and Player $O$ constructs a play on representatives. As a consequence, Player $O$ wins the game, if the constructed play is accepting. To account for the delay in the original game, Player $I$ is always one move ahead. This gives Player $O$ a lookahead of one equivalence class, which can be stored in the state space of the parity game. First, we adapt $\mathcal{G}_\Omega$ to keep track of the maximal color visited during a play.

**Construction 8.** Let $\mathcal{G}_\Omega = (\mathcal{A}, \textsc{Parity}(\Omega))$ be a parity game with arena $\mathcal{A} = (\Sigma_I, \Sigma_O, V_I, V_O, v_I, \delta_I, \delta_O)$, and coloring $\Omega\colon V_I \cup V_O \to [k]$. We construct the *color-tracking game* $\mathcal{G}^\circledcirc_{\Omega,v} = (\mathcal{A}^\circledcirc_v, \textsc{Parity}(\Omega))$ that is parameterized in $v \in V_I$ using the arena $\mathcal{A}^\circledcirc_v = (\Sigma_I, \Sigma_O, V_I^\circledcirc, V_O^\circledcirc, v_I^\circledcirc, \delta_I^\circledcirc, \delta_P^\circledcirc)$, where

- $V_I^\circledcirc = V_I \times [k]$,
- $V_O^\circledcirc = V_O \times [k]$,
- $v_I^\circledcirc = (v, \Omega(v))$,
- $\delta_I^\circledcirc((v', c), \sigma) = (\delta_I(v', \sigma), \max\{c, \Omega(v')\})$,
- $\delta_O^\circledcirc((v', c), \sigma) = (\delta_O(v', \sigma), \max\{c, \Omega(v')\})$, and
- $\Omega(v', c) = \Omega(v)$.

Note that $\mathcal{G}^\circledcirc_{\Omega,v}$ only differs against $\mathcal{G}_\Omega$ by saving the maximal color visited in the state space. Although, this saved maximal color finally is ignored by the winning condition.

**Remark 1.** *Let* $\mathcal{G}^{\circledcirc}_{\Omega,v_I} = (\mathcal{A}^{\circledcirc}_{v_I}, \textsc{Parity}(\Omega))$ *be the color-tracking game for some parity game* $\mathcal{G}_\Omega = (\mathcal{A}, \textsc{Parity}(\Omega))$ *with initial vertex* $v_I$ *and*

$$\rho^{\circledcirc} = ((v_0^I, c_0^I), i_0, (v_0^O, c_0^O), o_0)\,((v_1^I, c_1^I), i_1, (v_1^O, c_1^O), o_1)\dots$$

*be a play in* $\text{Plays}(\mathcal{A}^{\circledcirc}_{v_I})$. *Then*

$$\rho = (v_0^I, i_0, v_0^O, o_0)(v_1^I, i_1, v_1^O, o_1)\dots \in \text{Plays}(\mathcal{A})$$

*and* $\forall m \in \mathbb{N}.\ c_m^I = \max\{\Omega(v_j^P) \mid 0 \le j < m, P \in \{I, O\}\}$.

In the following, we work with partial functions $r\colon V_I \rightharpoonup 2^{V_I^{\circledcirc}}$, where we denote the domain of each such function $r$ by $\mathcal{D}_r$. Intuitively, we use $r$ to capture the information encoded in the lookahead provided by Player $I$. Hence, the function $r$ determines the remaining combinations of vertices of the game that still can be chosen by Player $O$ through picking a corresponding completion, after Player $I$ has provided his lookahead in the preceding turn. Assume Player $I$ has picked $\alpha_0 \dots \alpha_j$ and Player $O$ has picked $\beta_0 \dots \beta_i$ for $i < j$ such that the lookahead is $w = \alpha_{i+1} \dots \alpha_j$. Then, we can determine the vertex $v$ that $\mathcal{G}_\Omega$ reaches after processing $\binom{\alpha_0}{\beta_0} \dots \binom{\alpha_i}{\beta_i}$, but we cannot process $w$, since Player $O$ has not picked $\beta_{i+1} \dots \beta_j$ yet. However, we can determine the vertices Player $O$ can enforce by picking an appropriate completion, which will be the ones contained in $r(v)$.

To formalize the functions $r$, capturing the lookahead picked by Player $I$, we define the transition function $\delta_{\circledcirc}\colon 2^{V_I^{\circledcirc}} \times \Sigma_I \to 2^{V_I^{\circledcirc}}$ via

$$\delta_{\circledcirc}(S, a) = \bigcup_{v \in S} \bigcup_{b \in \Sigma_O} \delta^{\circledcirc}_O(\delta^{\circledcirc}_I(v, a), b),$$

*i.e.*, $\delta_{\circledcirc}$ is a transition function over powersets of vertices of $\mathcal{G}^{\circledcirc}_{\Omega,v}$, projected to $\Sigma_I$. As usual, we extend $\delta_{\circledcirc}$ to $\delta^*_{\circledcirc}\colon 2^{V_I^{\circledcirc}} \times \Sigma_I^* \to 2^{V_I^{\circledcirc}}$ by $\delta^*_{\circledcirc}(S, \varepsilon) = S$ and $\delta^*_{\circledcirc}(S, wa) = \delta_{\circledcirc}(\delta^*_{\circledcirc}(S, w), a)$.

Let $D \subseteq V_I$ be non-empty and let $w \in \Sigma_I^*$. We define the function $r_w^D$ with domain $D$ as follows: for every $v \in D$, we have

$$r_w^D(v) = \delta^*_{\circledcirc}(\{(v, \Omega(v))\}, w).$$

If $(v', c) \in r_w^D(v)$, then there is a word $w' \in (\Sigma_I \times \Sigma_O)^*$ whose projection to the input is $w \in \Sigma_I^*$ and every play infix of $\mathcal{G}_\Omega$ following $w'$ leads from $v$ to $v'$ and has maximal color $c$. Thus, if Player $I$ has picked the lookahead $w$, then

Player $O$ could pick an answer such that the combined word leads from $v$ to $v'$ with the maximal color visited on this path being $c$. We call $w$ a witness for a partial function

$$r\colon V_I \rightharpoonup 2^{V_I \times [k]}$$

if and only if $r = r_w^{\mathcal{D}_r}$. We obtain a language $W_r \subseteq \Sigma_I^*$ of witnesses for each such function $r$. We define

$$\mathfrak{R} = \{r\colon V_I \rightharpoonup 2^{V_I^\circledcirc} \mid \mathcal{D}_r \neq \emptyset \text{ and } W_r \text{ is infinite}\}.$$

**Lemma 8.** *Let $\mathcal{G}_\Omega = (\mathcal{A}, \textsc{Parity}(\Omega))$ be a parity game with arena $\mathcal{A} = (\Sigma_I, \Sigma_O, V_I, V_O, v_I, \delta_I, \delta_O)$, and coloring $\Omega\colon V_I \cup V_O \to [k]$ and let $\mathfrak{R}$ be defined as above.*

1. *Let $r \in \mathfrak{R}$. Then, $r(v) \neq \emptyset$ for every $v \in \mathcal{D}_r$.*
2. *Let $r \neq r' \in \mathfrak{R}$ such that $\mathcal{D}_r = \mathcal{D}_{r'}$. Then, $W_r \cap W_{r'} = \emptyset$.*
3. *Let $r\colon V_I \rightharpoonup 2^{V_I \times [k]}$ be a partial function with non-empty domain. Then, $W_r$ is recognized by a deterministic finite automaton with at most $2^{|\mathcal{A}|^2 k}$ states.*
4. *Let $D \subseteq V_I$ be non-empty and let $w \in \Sigma_I^*$ be such that $|w| \geq 2^{|\mathcal{A}|^2 k}$. Then, there exists some $r \in \mathfrak{R}$ with $\mathcal{D}_r = D$ and $w \in W_r$.*

*Proof.* The first statement directly follows from the totality of $\delta_I$ and $\delta_O$, while the second one follows from the definition of $r_w^D$, which is uniquely determined by $w$ and $D$. Hence, a fixed $w$ cannot witness two different functions $r$ and $r'$ with the same domain.

To prove the third statement, fix some partial function $r$ from $V_I$ to $2^{V_I^\circledcirc}$ with domain $D = \{v_1, \ldots, v_{|D|}\}$. The product of $|D|$ copies of the finite automaton induced by $\delta_\circledcirc$ with initial state

$$(\{(v_1, \Omega(v_1))\}, \ldots, \{(v_{|D|}, \Omega(v_{|D|}))\})$$

and the unique accepting state $(r(v_1), \ldots, r(v_{|D|}))$ recognizes the witness language $W_r$. As $|D| \leq |\mathcal{A}|$, the automaton has at most $2^{|\mathcal{A}|^2 k}$ states.

For proving the last statement, fix some non-empty $D$ and $w$ of length at least $2^{|\mathcal{A}|^2 k}$. Define $r = r_w^D$, which implies $w \in W_r$ by definition. As just shown, there exists a finite automaton recognizing $W_r$ with at most

$2^{|\mathcal{A}|^2 k} \leq |w|$ many states. Hence, by the pidgin whole principle, the accepting run of the automaton on $w$ must contain a state-repetition, which is why $W_r$ must be infinite and $r \in \mathfrak{R}$. □

Now, we are able to define the equivalent delay-free parity game. In this game, Player $I$ picks elements from $\mathfrak{R}$ while Player $O$ produces a run on witnesses, which corresponds to picking suitable completions to witnesses of the functions picked by Player $I$. She wins, if the constructed run is accepting. Furthermore, to account for the lookahead, as introduced by the delay function $f$, Player $I$ is always one move ahead.

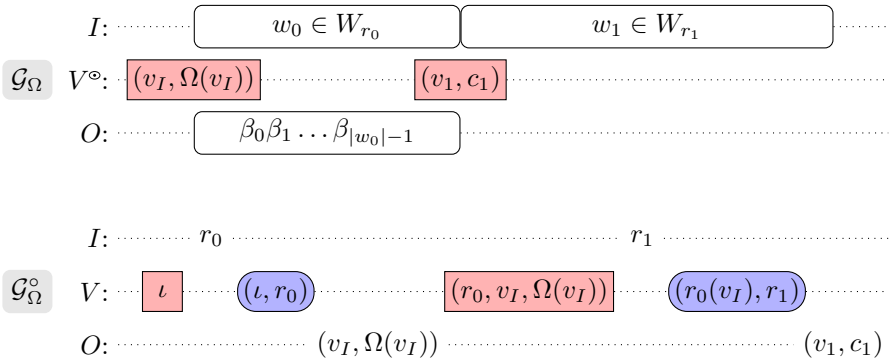
**Construction 9.** Let $\mathcal{G}_\Omega = (\mathcal{A}, \textsc{Parity}(\Omega))$ be a parity game with $\Omega\colon V_I \cup V_O \to [k]$ that is played in $\mathcal{A} = (\Sigma_I, \Sigma_O, V_I, V_O, v_I, \delta_I, \delta_O)$. To check whether there is some delay function $f$ such that Player $O$ wins $\Gamma_f(\mathcal{G}_\Omega)$ we construct the *delay-free* parity game $\mathcal{G}^\circ_\Omega = (\mathcal{A}^\circ, \textsc{Parity}(\Omega^\circ))$ with $\mathcal{A}^\circ = (\Sigma^\circ_I, \Sigma^\circ_O, V^\circ_I, V^\circ_O, v^\circ_I, \delta^\circ_I, \delta^\circ_O)$ and fresh vertices $\iota, \checkmark_I, \checkmark_O, \text{✗}_I, \text{✗}_O$, where:

- $\Sigma^\circ_I = \mathfrak{R}$
- $\Sigma^\circ_O = V_I \times [k]$
- $V^\circ_I = \{\iota, \checkmark_I, \text{✗}_I\} \cup (\mathfrak{R} \times V_I \times [k])$,
- $V^\circ_O = \{\checkmark_O, \text{✗}_O\} \cup ((2^{V_I \times [k]} \cup \{\iota\}) \times \mathfrak{R})$,
- $v^\circ_I = \iota$
- $\delta^\circ_I(x, r) = \begin{cases} (\iota, r) & \text{if } x = \iota \wedge \mathcal{D}_r = \{v_I\} \\ (r'(v), r) & \text{if } x = (r', v, c) \ \wedge \ \mathcal{D}_r = pr_0(r'(v)) \\ \text{✗}_O & \text{if } x = \text{✗}_I \\ \checkmark_O & \text{otherwise} \end{cases}$
- $\delta^\circ_O(x, (v, c)) = \begin{cases} (r, v, c) & \text{if } x = (\iota, r) \wedge v = v_I \wedge c = \Omega(v_I) \\ (r, v, c) & \text{if } x = (X, r) \wedge (v, c) \in X \\ \checkmark_I & \text{if } x = \checkmark_O \\ \text{✗}_I & \text{otherwise} \end{cases}$
- $\Omega^\circ(x) = \begin{cases} c & \text{if } x = (r, v, c) \in \mathfrak{R} \times V_I \times [k] \\ 1 & \text{if } x = \text{✗}_I \\ 0 & \text{otherwise} \end{cases}$

The correctness proof of Construction 9 is split according to the two players of the game.

**Lemma 9.** *If Player $I$ wins $\mathcal{G}^\circ_\Omega$, then Player $I$ wins $\Gamma_f(\mathcal{G}_\Omega)$ for all delay functions $f$.*

*Proof.* Let $\sigma^\circ_I$ be a winning strategy for Player $I$ in $\Gamma_f(\mathcal{G}^\circ_\Omega)$. We construct a winning strategy $\sigma_I$ for Player $I$ in $\Gamma_f(\mathcal{G}_\Omega)$ for some arbitrary delay function $f$ by simulating a play of $\Gamma_f(\mathcal{G}_\Omega)$ with a play of $\mathcal{G}^\circ_\Omega$. Let $r_0 = \sigma^\circ_I(\varepsilon) \in \mathfrak{R}$ be the first move of Player $I$ in $\mathcal{G}^\circ_\Omega$, where Player $I$ picks a valid $r_0 \in \mathfrak{R}$, as otherwise he has lost immediately. Also note that due to Lemma 8 (Items 1 and 2) picking an element of $\mathfrak{R}$ is always possible for Player $I$. Player $O$ then answers by picking $v_I$, which is the only reasonable choice, as otherwise she looses $\mathcal{G}^\circ_\Omega$ immediately. Let $(v_0, c_0) = (v_I, \Omega(v_I))$ and let $r_1 = \sigma^\circ_I((v_0, v_1))$ be Player $I$'s next response. Due to $W_{r_0}$ and $W_{r_1}$ being infinite, according to the membership in $\mathfrak{R}$, we can choose witnesses $w_0 \in W_{r_0}$ and $w_1 \in W_{r_1}$ such that $f(0) \leq |w_0|$ and $f_\Sigma(|w_0|) \leq |w_1|$.




We are ready to simulate a play prefix in $\Gamma_f(\mathcal{G}_\Omega)$: Player $I$ picks the prefix $\alpha_0\alpha_1 \dots \alpha_{f_\Sigma(|w_0|)-1}$ of $w_0w_1$ of length $f_\Sigma(|w_0|)$ in his first moves, which is long enough to play $|w_0|$ many rounds: $w_0$ is long enough for the first round and $w_1$ is long enough for the next $|w_0|$ rounds. Formally, we fix $\sigma_I$ for the first $f_\Sigma(|w_0|)$ responses, independently of the responses $\beta_0 \dots \beta_{|w_0|-1}$ of Player $O$, *i.e.*, we define $\sigma_I(\varepsilon) = \alpha_0 \dots \alpha_{f(0)-1}$ and for every $0 \leq j < |w_0|$ let $\sigma_I(\beta_0 \dots \beta_{j-1}) = \alpha_{f_\Sigma(j)} \dots \alpha_{f_\Sigma(j+1)-1}$. As we have played $|w_0|$ many rounds, the response of Player $O$ is long enough to determine the vertex $(v_1, c_1)$ that

is reached by the play $\rho^{\circledcirc} \in \mathrm{Plays}(\mathcal{A}^{\circledcirc}_{v_I})$ after playing $\binom{\alpha_0}{\beta_0}\binom{\alpha_1}{\beta_1}\dots\binom{\alpha_{|w_0|-1}}{\beta_{|w_0|-1}}$ and used as Player $O$'s response in $\mathcal{G}^{\circ}_{\Omega}$. We are in the following situation for $i = 1$:

- In $\mathcal{G}^{\circ}_{\Omega}$, we have constructed a play prefix:

$$(\iota, r_0, (\iota, r_0), (v_0, c_0)) \dots ((r_{i-1}, (v_{i-1}, c_{i-1})), r_i, (X_i, r_i), (v_i, c_i))$$

- In $\Gamma_f(\mathcal{G}_{\Omega})$, Player $I$ has committed to the input

$$w_0 \dots w_i = \alpha_0\alpha_1 \dots \alpha_{|w_0 \dots w_i|-1}$$

  from which we have fixed the first $f_{\Sigma}(|w_0 \dots w_{i-1}|)$ letters to be picked by $\sigma_I$ and Player $O$ has picked $\beta_0 \dots \beta_{|w_0 \dots w_{i-1}|-1}$.

- The word $w_j$ is a witness for $r_j$ for every $j \leq i$.

Now, let $i > 0$ be arbitrary and let $r_{i+1} = \sigma^{\circ}_I((v_0, v_1) \dots (v_i, c_i)) \in \mathfrak{R}$ be the next move of Player $I$ in $\mathcal{G}^{\circ}_{\Omega}$ and let $w_{i+1} \in W_{r_{i+1}}$ be a witness, large enough such that $|w_{i+1}| \geq f_{\Sigma}(|w_0 \dots w_i|)$. Then, in $\Gamma_f(\mathcal{G}_{\Omega})$ Player $I$ commits to $w_{i+1}$ as his next moves and picks $\sigma_I(\beta_0 \dots \beta_{j-1}) = \alpha_{f_{\Sigma}(j)} \dots \alpha_{f_{\Sigma}(j+1)-1}$ for all $|w_0 \dots w_{i-1}| \leq j < |w_0 \dots w_i|$. Player $O$ responds by $\beta_{|w_0 \dots w_{i-1}|} \dots \beta_{|w_0 \dots w_i|-1}$, which is again long enough to determine the vertex $(v_{i+1}, c_{i+1})$ that is reached by the play $\rho^{\circledcirc} \in \mathrm{Plays}(\mathcal{A}^{\circledcirc}_{v_i})$ after playing

$$\binom{\alpha_{|w_0 \dots w_{i-1}|}}{\beta_{|w_0 \dots w_{i-1}|}}\binom{\alpha_{|w_0 \dots w_{i-1}|+1}}{\beta_{|w_0 \dots w_{i-1}|+1}} \dots \binom{\alpha_{|w_0 \dots w_i|-1}}{\beta_{|w_0 \dots w_i|-1}}.$$

As a result, we are again in the aforementioned situation for $i + 1$. Thus, repeating the simulation ad infinitum concludes the definition of $\sigma_I$.

It remains to show that $\sigma_I$ is winning for Player $I$. Consider a play $\rho \in \mathrm{Plays}(\mathcal{A}, f, \sigma_I)$ that is consistent with $\sigma_I$ and let $\gamma = \binom{\alpha_0}{\beta_0}\binom{\alpha_1}{\beta_1}\binom{\alpha_2}{\beta_2}\dots$ be the outcome of $\rho$. Note that $\alpha_0\alpha_1\alpha_2 \dots$ is equal to $w_0w_1w_2 \dots$, where each $w_i$ is a witness of $r_i$. A straightforward inductive application of Remark 1 shows that $v_{i+1}$ is the vertex that $\rho$ reaches after processing $w_i$ and the corresponding moves of Player $O$ starting in $v_i$ and that $c_{i+1}$ is the maximal color seen in the play. Thus, the maximal color visited infinitely often by $\rho$ after processing $w$ is the same as the maximal color of the sequence $c_0c_1c_2 \dots$ of the play $\rho^{\circ} \in \mathrm{Plays}(\mathcal{A}^{\circ}, \sigma^{\circ}_I)$ constructed by our simulation and consistent with the winning strategy $\sigma^{\circ}_I$ of Player $I$ in $\mathcal{G}^{\circ}_{\Omega}$. Hence, the play $\rho$ is winning for Player $I$, since it shares the same maximal color seen infinitely often as $\rho^{\circ}$. Thus, $\sigma_I$ is a winning strategy for Player $I$ in $\Gamma_f(\mathcal{G}_{\Omega})$. □

**Lemma 10.** *If Player O wins the parity game* $\mathcal{G}_\Omega^\circ = (\mathcal{A}, \textsc{Parity}(\Omega))$ *with maximal color* $k$*, then Player O wins* $\Gamma_f(\mathcal{G}_\Omega)$ *for some constant delay function* $f$ *with* $f(0) \leq 2^{|\mathcal{A}|^2 k+1}$.

*Proof.* Let $\sigma_O^\circ$ be a wining strategy for Player $O$ in $\mathcal{G}_\Omega^\circ$. We construct a winning strategy $\sigma_O$ for Player $O$ in $\Gamma_f(\mathcal{G}_\Omega)$ for the constant delay function $f$ with $f(0) = 2d$, where $d = 2^{|\mathcal{A}|^2 k}$. The strategy $\sigma_O$ is constructed by simulating a play of $\Gamma_f(\mathcal{G}_\Omega)$ by a play of $\mathcal{G}_\Omega^\circ$.

In the following, both players pick their moves in $\Gamma_f(\mathcal{G}_\Omega)$ in blocks of length $d$. We denote Player $I$'s blocks by $\overline{a_i}$ and Player $O$'s blocks by $\overline{b_i}$, *i.e.*, every $\overline{a_i}$ is in $\Sigma_I^d$ and every $\overline{b_i}$ is in $\Sigma_O^d$.

Let $\overline{a_0 a_1}$ be the first move of Player $I$ in $\Gamma_f(\mathcal{G}_\Omega)$, fix $(v_0, c_0) = (v_I, \Omega(v_I))$, and let $r_0 = r_{\overline{a_0}}^{\{v_0\}}$ and $r_1 = r_{\overline{a_1}}^{r_0(v_0)}$ be the first two functions Player $I$ picks in $\mathcal{G}_\Omega^\circ$, which are well-defined according to Lemma 8 (Item 4). Then,

$$(\iota, r_0, (\iota, r_0), (v_0, c_0))\,((r_0, v_0, c_0), r_1, (X_1, r_1), (v_1, c_1))$$

is a play prefix in $\mathcal{G}_\Omega^\circ$ for $\sigma_O^\circ(r_0 r_1) = (v_1, c_1)$ that is consistent with $\sigma_O^\circ$. Thus, we are in the following situation for $i = 1$:

- in $\Gamma_f(\mathcal{G}_\Omega)$, Player $I$ has picked blocks $\overline{a_0} \dots \overline{a_i}$ and Player $O$ has picked $\overline{b_0} \dots \overline{b_{i-2}}$,
- in $\mathcal{G}_\Omega^\circ$ we have constructed a play prefix

  $$(\iota, r_0, (\iota, r_0), (v_0, c_0)) \dots ((r_{i-1}, v_{i-1}, c_{i-1}), r_i, (X_i, r_i), (v_i, c_i))$$

  that is consistent with $\sigma_O^\circ$, and
- $\overline{a_j}$ is a witness for $r_j$ for every $j \leq i$.

Let $i > 0$ be arbitrary. The rules of $\mathcal{G}_\Omega^\circ$ imply that $(v_i, c_i) = r_{i-1}(v_{i-1}, c_{i-1})$ and $v_i \in \mathcal{D}_{r_i}$. Furthermore, as $\overline{a_{i-1}}$ is a witness for $r_{i-1}$, there is some $\overline{b_{i-1}}$ such that a play $\rho^\circledcirc \in \text{Plays}(\mathcal{A}_{v_{i-1}}^\circledcirc)$ reaches $v_i$ after processing $\binom{\overline{a_{i-1}}}{\overline{b_{i-1}}}$ from $(v_{i-1}, \Omega(v_{i-1}))$. Player $O$'s strategy for $\Gamma_f(\mathcal{G}_\Omega)$ is to pick the letters of $\overline{b_{i-1}}$ in the next $d$ rounds. These are answered by Player $I$ by $d$ letters forming $\overline{a_{i+1}}$. This way, we obtain $r_{i+1} = r_{\overline{a_{i+1}}}^{r_i(v_i)}$ from Lemma 8 (Item 4) bringing us back to the aforementioned situation for $i + 1$, concluding the definition of $\sigma_O$.

It remains to show that $\sigma_O$ is a winning strategy for Player $O$. Let $w = \binom{\overline{a_0}}{\overline{b_0}}\binom{\overline{a_1}}{\overline{b_1}}\binom{\overline{a_2}}{\overline{b_2}} \cdots$ be the outcome of a play $\rho \in \text{Plays}(\mathcal{A}, f, \sigma_O)$ of $\Gamma_f(\mathcal{G}_\Omega)$ that is consistent with $\sigma_O$. Also, let $\rho^\circ \in \text{Plays}(\mathcal{A}, \sigma_O^\circ)$ be the corresponding

play of $\mathcal{G}_\Omega$ constructed as described in the simulation above, where each $\overline{a_i}$ is a witness for $r_i$. A straightforward inductive application of Remark 1 resolves that $(v_{i+1}, c_{i+1})$ is the vertex reached in $\mathcal{G}^{\circledast}_{\Omega,v_I}$ after processing $\left(\frac{\overline{a_i}}{b_i}\right)$ starting in $(v_i, \Omega(v_i))$ and that $c_{i+1}$ is the largest color seen on this prefix. As $\rho^\circ$ is consistent with $\sigma^\circ_O$, the sequence $c_0c_1c_2\cdots$ satisfies the parity condition, i.e., the maximal color occurring infinitely often is even. Thus, the maximal color occurring infinitely often in $\rho$ is even as well, i.e., $\rho$ is winning for Player $O$. Hence, $\sigma_O$ is a winning strategy for Player $O$ and she wins $\Gamma_f(\mathcal{G}_\Omega)$. □

**Corollary 2.** *Delay games with parity conditions are determined.*

Note that the results of Corollary 2 had been previously known [62], but independently they also follow from the correctness of Construction 9.

**Theorem 22.** *Let $\mathcal{G}_\Omega = (\mathcal{A}, \textsc{Parity}(\Omega))$ be a parity game with $k$ colors. Then the following are equivalent:*

1. *Player $O$ wins $\Gamma_f(\mathcal{G}_\Omega)$ for some delay function $f$.*
2. *Player $O$ wins $\Gamma_f(\mathcal{G}_\Omega)$ for some constant delay function $f$ with $f(0) \leq 2^{|\mathcal{A}|^2k+1}$.*

With Lemma 9 and Lemma 10 at hand, we can finally prove the main theorem of this section: determining whether Player $O$ wins a delay game $\Gamma_f(\mathcal{G}_\Omega)$ for some delay function $f$ is in ExpTime. To this end, it suffices to construct $\mathcal{G}^\circ_\Omega$ as a classical parity game and to show that it can be constructed and solved in exponential time.

*Proof of Theorem 21.* First, we argue that $\mathfrak{R}$ can be constructed in exponential time: to this end, one constructs for every partial function $r\colon V_I \rightharpoonup 2^{V_I\times[k]}$ the automaton of Lemma 8 (Item 3) recognizing $W_r$ and tests it for recognizing an infinite language. There are exponentially many functions and each automaton is of exponential size, which yields the desired result.

Having constructed $\mathfrak{R}$ in exponential time also allows us to construct $\mathcal{G}^\circ_\Omega$ in exponential time, which is won by Player $O$ if and only if she wins $\Gamma_f(\mathcal{G}_\Omega)$ for some delay function $f$. A parity game with $n$ vertices, $m$ edges, and $k$ colors can be solved in time $O(mn^{\frac{k}{3}})$ [128]. The parity game $\mathcal{G}^\circ_\Omega$ has at most $O(2^{|\mathcal{G}_\Omega|^2k})$ many vertices and at most $|\mathcal{G}_\Omega|$ many colors. Hence, it can be solved in exponential time in the size of $\mathcal{G}_\Omega$. □

## 6.2 Safety Games

We complement the already proven upper bounds on the complexity of solving delay games with parity winning conditions by matching lower bounds for safety games. Consequently, as safety games can be easily encoded with the parity winning condition, they also imply matching lower bounds for delay games with parity winning conditions. Delay games with reachability winning conditions are considered in the later Section 6.3, as it turns out that they are easier to solve than delay games with safety winning conditions in terms of complexity.

**Theorem 23.** *The following problem is* ExpTime*-hard: Given a safety game $\mathcal{G}_\forall = (\mathcal{A}, \textsc{Safety}(S))$, does Player $O$ win $\Gamma_f(\mathcal{G}_\forall)$ for some delay function $f$?*

*Proof.* Let $\mathcal{TM} = (\Sigma, Q, Q_\exists, Q_\forall, q_I, \Delta, q_A, q_R)$ be an alternating polynomial space Turing machine, where $\Delta \subseteq Q \times \Sigma \times Q \times \Sigma \times \{-1, 0, 1\}$ is the transition relation, and let $x \in \Sigma^*$ be an input. For technical reasons, we assume the accepting state $q_A$ and the rejecting state $q_R$ to be equipped with a self-loop. Furthermore, let $p$ be a polynomial that bounds $\mathcal{TM}$'s space consumption. We construct a safety game $\mathcal{G}_\forall = (\mathcal{A}, \textsc{Safety}(S))$ of polynomial size in $|\Delta|$ and $p(|x|)$ such that $\mathcal{TM}$ accepts $x$ if and only if Player $I$ wins $\Gamma_f(\mathcal{G}_\forall)$ for all delay functions $f$. This is sufficient, since APSpace = ExpTime [23] is closed under complement. Thus, we give Player $I$ control over the existential states while Player $O$ controls the universal ones. Additionally, Player $I$ is in charge of producing all configurations with his moves. He can copy configurations in order to wait for Player $O$'s choices for the universal transitions, which are delayed due to the lookahead.

Formally, the input alphabet $\Sigma_I = \Sigma \cup Q \cup \{N, C\}$ contains $\Sigma \cup Q$ and two separators $N$ and $C$, while the output alphabet $\Sigma_O = \Delta \cup \{✗, ✓\}$ contains $\Delta$, and two signals ✗ and ✓. Intuitively, Player $I$ produces configurations of $\mathcal{TM}$ of length $p(|x|)$ preceded by either $C$ or $N$ to denote whether the configuration is a *copy* of the previous one or a *new* one. Copying configurations is necessary to bridge the lookahead while waiting for Player $O$ to determine the transition that is applied to a universal configuration. Player $I$ could copy a configuration ad infinitum, but this will be losing for him, unless it is an accepting one. Player $O$ chooses universal transitions at every separator (if the successor configuration is existential or the separator is a $C$, then her choice is ignored) by picking a letter from $\Delta$. At every other position, she has to pick a signal: ✗ allows her to claim an error in the configurations picked

by Player $I$ while ✓ means that she does not claim an error at the current position.

The arena of the game $\mathcal{G}_\forall$ challenges Player $I$ to satisfy the following properties of the outcome $\binom{\alpha_0}{\beta_0}\binom{\alpha_1}{\beta_1}\binom{\alpha_2}{\beta_2}\ldots \in (\Sigma_I \times \Sigma_O)^\omega$ in order to win:

1. $\alpha \in (\{N, C\}\cdot\mathrm{Conf})^\omega$, where Conf is the set of encodings of configurations of length $p(|x|)$, *i.e.*, words of length $p(|x|)+1$ over $\Sigma \cup Q$ that contain exactly one letter from $Q$. If this is not the case, then the corresponding play is lead to an accepting sink, *i.e.*, in order to win, Player $I$ has to produce some input $\alpha$ that satisfies the requirement.

2. $\beta \in (\Delta \cdot \{✓, ✗\}^{p(|x|)+1})^\omega$. If this is not the case, then the play is lead to a rejecting sink, *i.e.*, in order to win, Player $O$ has to produce some output $\beta$ that satisfies the requirement.

3. The first configuration picked by Player $I$ is the initial one of $\mathcal{TM}$ on $x$. If this is not the case, the play is lead to an accepting sink.

4. If $\beta$ contains a ✗, then we check, whether there is indeed an error by doing the following at the first occurrence of ✗: we store the previous, the current, and the next input letter, the transition picked by Player $O$ at the last separator $N$, and whether the current configuration is existential or universal as part of the state space of the arena. Some of this information has to be stored continuously, since these letters appear before the first ✗. This is possible with a set of vertices, whose size is polynomial in $|\Sigma| + |Q| + |\Delta|$.

   Then, we processes $p(|x|)+1$ letters (and remember whether we traversed the separator $N$ or $C$), and then check whether the letter just reached is updated correctly or not:

   - If the separator is $C$, then the current letter is updated correctly, if it is equal to the marked one.
   - If the separator is $N$ and the configuration in which the error was marked is existential, then the letter is updated correctly, if there is a transition of $\mathcal{TM}$ that is compatible with the current letter and the marked one.
   - If the separator is $N$ and the configuration in which the error was marked is universal, then the letter is updated correctly, if it is compatible with the transition picked by Player $O$ at the last separator $N$ before the ✗, which is stored by the automaton. If she has picked a transition that is not applicable to the current configuration, the play is lead to a rejecting sink.

If the update is not correct, *i.e.*, Player $O$ has correctly claimed an error, then the play is lead to an accepting sink. Otherwise, it is lead to a rejecting sink, *i.e.*, in order to win, Player $O$ can only claim an error at an incorrect update of a configuration, but she wins if she correctly claims the error. All subsequent claims by Player $O$ are ignored, *i.e.*, after the first claim is evaluated, the play is either accepted or rejected, no matter how it is continued.

5. Finally, if $\alpha$ contains the accepting state of $\mathcal{TM}$, then the play is lead to a rejecting sink, unless Player $O$ correctly claimed an error in a preceding configuration.

All of the aforementioned properties can be checked using safety games whose sizes are polynomial in the size of $\mathcal{TM}$ and $p(|x|)$. Thus, they also can be checked using a single arena $\mathcal{A}$, build using a cross-product construction for example. All non-sink vertices of the arena $\mathcal{A}$ are accepting, *i.e.*, as long as both players stick to their requirements on the format: Player $I$ starts with the initial configuration, Player $O$ does not incorrectly claim an error, and the accepting state of $\mathcal{TM}$ is not reached, the resulting play will satisfy the safety condition.

It remains to prove that $\mathcal{TM}$ accepts $x$ if and only if Player $I$ wins $\Gamma_f(\mathcal{G}_\forall)$ all delay functions $f$.

"$\Rightarrow$": Assume that $\mathcal{TM}$ accepts $x$ and let $f$ be an arbitrary delay function. We show that Player $I$ wins $\Gamma_f(\mathcal{G}_\forall)$. Player $I$ starts with the initial configuration and picks the successor configuration of an existential one according to the accepting run, and copies universal configurations as often as necessary to obtain a play prefix in which Player $O$ has to determine the transition she wants to apply in this configuration. Thus, he will eventually produce an accepting configuration of $\mathcal{TM}$ without ever introducing an error. Hence, either Player $O$ incorrectly claims an error or the play reaches an accepting sink. In either case, Player $I$ wins the resulting play, *i.e.*, he has a winning strategy for $\Gamma_f(\mathcal{G}_\forall)$.

"$\Leftarrow$": We show the contrapositive, *i.e.*, assume that $\mathcal{TM}$ rejects $x$ and let $f$ be the constant delay function with $f(0) = p(|x|) + 3$. We show that Player $O$ wins $\Gamma_f(\mathcal{G}_\forall)$, which is sufficient due to safety games with delay being determined (Corollary 2). At every time, Player $O$ has enough lookahead to correctly claim the first error introduced by Player $I$, if he introduces one. Furthermore, she has access to the whole encoding of each universal configuration whose successor she has to determine. This allows her to simulate the rejecting run of $\mathcal{TM}$ on $x$, which does not reach the accepting state $q_A$, no matter which transitions Player $I$ picks. Thus, he has to introduce an error

in order to win, which Player $O$ can detect using the lookahead. If Player $I$ does not introduce an error, the play proceeds ad infinitum by repeating a rejecting configuration forever. In every case, the resulting play is winning for Player $O$. Thus, Player $O$ has a winning strategy to win $\Gamma_f(\mathcal{G}_\forall)$. □

It is noteworthy that the lower bound just proven does not require the full exponential lookahead that is necessary in general to win delay games with safety conditions: Player $O$ wins the game constructed above with sublinear lookahead, as $p(|x|) + 3$ is smaller than the size of $\mathcal{A}$. Thus, determining the winner of a delay game with a safety condition with respect to linearly bounded delay is already ExpTime-hard.

## 6.3 Reachability Games

Recall that universality of the projection to the first component of the winning condition is a necessary condition of Player $O$ for having a winning strategy in a delay game. Our first result states that universality is also sufficient in the case of reachability winning conditions. Thus, solving delay games with reachability conditions is equivalent, via linear time reductions, to the universality problem for non-deterministic reachability automata, which is PSpace-complete [80]. Therefore, solving delay games with reachability conditions is PSpace-complete as well. Moreover, our proof yields an exponential upper bound on the necessary lookahead.

**Theorem 24.** *The following problem is in* PSpace*: Given a reachability game $\mathcal{G}_\exists = (\mathcal{A}, \text{Reach}(R))$, does Player $O$ win $\Gamma_f(\mathcal{G}_\exists)$ for some delay function $f$?*

*Proof.* We show membership by a reduction to the LTL satisfiability problem, which is PSpace-complete [134]. Let $\mathcal{A} = (\Sigma_I, \Sigma_O, V_I, V_O, v_I, \delta_I, \delta_O)$ and $\mathcal{O} = \Sigma_I \cup V_I \cup V_O \cup \Sigma_I$. We construct the LTL formula $\varphi$ over the alphabet $\Sigma = 2^{\mathcal{O}}$ as follows:

$$\varphi = v_I \wedge \square\Big( \bigwedge_{v \in V_I} \bigwedge_{i \in \Sigma_I} (v \wedge i \rightarrow \delta_I(v,i)) \wedge \bigwedge_{v \in V_O} \bigwedge_{o \in \Sigma_O} (v \rightarrow \bigcirc \delta_O(v,o)) \wedge \bigwedge_{v \in R} \neg v \Big)$$

We claim that $\varphi$ is satisfiable if and only if Player $I$ wins $\Gamma_f(\mathcal{G}_\exists)$ for all delay functions $f$. A proof of this claim suffices, since PSpace is closed under complement and delay games with $\omega$-regular winning conditions are determined (Corollary 2).

"⇒": Assume $\varphi$ is satisfiable and let $\alpha \in \Sigma^\omega$ be the infinite word such that $\alpha \vDash \varphi$. Furthermore, let $\beta = \alpha \cap \Sigma_I$ be the projection of $\alpha$ to $\Sigma_I$. Then $\beta$ defines a winning strategy $\sigma_I$ for Player $I$ in $\Gamma_f(\mathcal{G}_\exists)$ via $\sigma(\varepsilon) = \beta_0\beta_1 \dots \beta_{f(0)-1}$ and $\sigma(w) = \beta_{f_\Sigma(|w|-1)}\beta_{f_\Sigma(|w|-1)+1} \dots \beta_{f_\Sigma(|w|)}$ for all $w \in (\Sigma_I)^+$ and any delay function $f$. A simple induction shows that for all $n \in \mathbb{N}$ and every play $\rho \in \text{Plays}(\mathcal{A}, f, \sigma_I)$ we have that $\rho_n = (v_n^I, i_n, v_n^O, o_n)$ implies that $v_n^I \in \alpha_i$ and $v_n^O \in \alpha_i$. Thus, the strategy $\sigma_I$ is winning, because by the satisfaction of $\varphi$ we have that $\alpha \cap R = \emptyset^\omega$. Hence, Player $I$ wins $\Gamma_f(\mathcal{G}_\exists)$.

"⇐": Assume that Player $I$ wins the game $\Gamma_f(\mathcal{G}_\exists)$ for the constant delay function $f$ with $f(0) = 2^{|A|}$ using some winning strategy $\sigma_I$. We define the infinite sequence $\alpha \in \Sigma^\omega$ for all $n \in \mathbb{N}$ using

$$\alpha_n = \bigcup \{\{v_n^I, v_n^O, i_n\} \mid \exists \rho \in \text{Plays}(\mathcal{A}, f, \sigma_I).\ \rho_n = (v_n^I, i_n, v_n^O, o_n)\}.$$

A simple induction shows that $\alpha \vDash \varphi$. □

**Theorem 25.** *The following problem is* PSPACE*-hard: Given a reachability game $\mathcal{G}_\exists = (\mathcal{A}, \text{REACH}(R))$, does Player $O$ win $\Gamma_f(\mathcal{G}_\exists)$ for some delay function $f$?*

*Proof.* We show hardness by a direct reduction from the acceptance problem of polynomial space Turing machines. Let $\mathcal{TM} = (\Sigma, Q, q_I, \delta, q_A, q_R)$ be such a machine, where $\delta\colon Q \times \Sigma \to Q \times \Sigma \times \{-1, 0, 1\}$ is the transition function. We assume w.l.o.g. that the accepting state $q_A$ and the rejecting state $q_R$ are sink states. Furthermore, let $x \in \Sigma^*$ be an input for $\mathcal{TM}$.

Let $p$ be a polynomial that bounds the space-consumption of $\mathcal{TM}$. From $p$ we can compute a polynomial $p'$ such that $2^{p'}$ bounds the time-consumption of $\mathcal{TM}$. Thus, $s = p(|x|)$ and $t = 2^{p'(|x|)}$ are upper bounds on the space- and time-consumption of $\mathcal{TM}$ on $x$, respectively. We construct a reachability game $\mathcal{G}_\exists = (\mathcal{A}, \text{REACH}(R))$ such that $x$ is accepted by $\mathcal{TM}$ if and only if Player $I$ wins $\mathcal{G}_\exists$. This suffices to prove our claim, since PSPACE is closed under complement and reachability games with delay are determined.

A configuration of $\mathcal{TM}$'s run on $x$ is encoded by a word $c \in (Q \cup \Sigma)^{s+1}$ as usual. In the following, we do not distinguish between a configuration and its encoding. In the construction, we need to count the configurations of the run. To this end, let $\langle n \rangle_2 \in \{0,1\}^*$ denote the binary encoding of $n$ in the range $0 \leq n \leq t-1$ using $\log_2(t)$ many bits. Let $\Sigma_I = Q \cup \Sigma \cup \{0, 1, \$, \#\}$, where we assume $(Q \cup \Sigma) \cap \{0, 1, \$, \#\} = \emptyset$, and $\Sigma_O = \{0, 1, 2\}$. Consider the following three reachability games $\mathcal{G}_\exists^i = (\mathcal{A}_n, \text{REACH}(R_i))$ with arenas $\mathcal{A}_i = (\Sigma_I, \Sigma_O, V_I^i, V_O^i, v_I^i, \delta_I^i, \delta_O^i)$ for $i \in \Sigma_O$.

1. $\mathcal{G}_\exists^0$ is a reachability game, where Player $O$ wins iff Player $I$ produces an input sequence $\alpha$ not starting with $\langle 0 \rangle_2 \,\$\, c_0 \,\#$, where $c_0$ is the initial configuration of $\mathcal{TM}$ on $x$.

2. $\mathcal{G}_\exists^1$ is a reachability game, where Player $O$ wins iff Player $I$ produces an input sequence $\alpha$ that contains the infix $\langle t-1 \rangle_2$, but the first occurrence of $\langle t-1 \rangle_2$ is not followed by $\$\, c \,\#$, where c is some accepting configuration.

3. $\mathcal{G}_\exists^2$ is a reachability game, where Player $O$ wins iff Player $I$ produces an input sequence $\alpha$ that contains an infix $\langle n \rangle_2 \,\$\, c \,\#$ with $n < t-1$, not followed by $\langle n+1 \rangle_2 \,\$\, c' \,\#$, where $c'$ is the successor configuration of $c$.

We claim that all three games can be constructed in size polynomial in $|\mathcal{TM}| + |x|$. This is straightforward for $\mathcal{G}_\exists^0$ and $\mathcal{G}_\exists^1$. For $\mathcal{G}_\exists^2$ we need to use the fact that it suffices to find a single bit or tape-cell that is not updated correctly by Player $I$. This cell can be identified by Player $O$ using lookahead and then be marked by using one of the markers 1 or 2, where we assume that 0 is played by Player $O$ as the default.

We build the reachability game $\mathcal{G}_\exists = (\mathcal{A}, \textsc{Reach}(R_0 \cup R_1 \cup R_2))$ from the arenas $\mathcal{A}_0$, $\mathcal{A}_1$, and $\mathcal{A}_2$ by adding a fresh initial vertex for Player $I$ leading to a fresh vertex for Player $O$ for any possible input. In this second fresh vertex, Player $O$ then can choose between moving to one of the arenas $\mathcal{A}_i$ by playing a letter $i \in [3]$, respectively.

"$\Rightarrow$": Assume $\mathcal{TM}$ accepts $x$ and let $c_0, c_1, \ldots, c_k$ for $k \leq t-1$ be the accepting run. Then, Player $I$ wins $\Gamma_f(\mathcal{G}_\exists)$ for any delay function $f$ by playing the word

$$\alpha_{acc} = \langle 0 \rangle_2 \,\$\, c_0 \,\# \, \langle 1 \rangle_2 \,\$\, c_1 \,\# \, \ldots \langle k \rangle_2 \,\$\, c_k \,\# \, \langle k+1 \rangle_2 \,\$\, c_k \,\# \, \ldots (\langle t-1 \rangle_2 \,\$\, c_k \,\#)^\omega$$

in subsequent pieces as given by $f$, while ignoring the responses of Player $O$.

"$\Leftarrow$": Next, assume Player $I$ has a winning strategy $\sigma_I$ to win $\Gamma_f(\mathcal{G}_\exists)$ for the constant delay function $f$ with $f(0) = t(\log_2(t) + s + 3)$. Let $w = \sigma_I(\epsilon)$, then $w$ starts with $\langle 0 \rangle_2 \,\$\, c_0 \,\#$, where $c_0$ is the initial configuration of $\mathcal{TM}$ on $x$, since otherwise Player $O$ would have a winning strategy by choosing $\mathcal{A}_0$ initially. This is followed by $\langle 1 \rangle_2 \,\$\, c_1 \,\#$, where $c_1$ is the successor configuration of $c_0$. Otherwise, Player $O$ could choose $\mathcal{A}_2$. We can continue this argument until we reach an infix $\langle t-1 \rangle_2 \,\$\, c_{t-1} \,\#$, which is the first occurrence of the infix $\langle t-1 \rangle_2$. Due to Player $O$ not having a winning strategy by choosing $\mathcal{A}_2$ initially, it follows that $c_{t-1}$ is an accepting configuration. Therefore, $\mathcal{TM}$ accepts $x$, because we have constructed an accepting run. □

Another consequence of the proof of Theorem 22 concerns the strategy complexity of delay games with reachability conditions: if Player $O$ wins for some

delay function, then she has a winning strategy that receives exponentially many input letters and answers by also giving exponentially many output letters and thereby already guarantees a winning play, *i.e.*, all later moves are irrelevant. Thus, the situation is similar to classical reachability games, in which positional attractor strategies allow a player to guarantee a win after a bounded number of moves. The strategy described above can be implemented by a lookup table mapping all minimal plays of $\mathcal{G}_\exists$ projected to $\Sigma_I$ to a word in $\Sigma_O^*$ of the same length such that the combination results in a winning play.

# 7 Lower Bounds on the Delay

In this section, we prove lower bounds on the necessary lookahead for Player $O$ to win delay games. We first give an exponential lower bound for reachability conditions, then we extend this idea to provide an exponential lower bound for safety conditions. Consequently, the same bounds hold for more expressive acceptance conditions like Büchi, co-Büchi, and parity. They are complemented by an exponential upper bound for parity conditions in the next section. Note that both lower bounds already hold for deterministic automata.

**Definition 31.** Let $\Sigma = \{1, \ldots, n\}$. We say that $w$ in $\Sigma^*$ contains a *bad $j$-pair*, for $j \in \Sigma$, if there are two occurrences of $j$ in $w$ such that no $j' > j$ occurs in between, *i.e.*, $\exists n, m \in [|w|].\ w_n = w_m = j \wedge \forall n < i < m.\ w_i \leq j$.

Consider the language $L_{j\text{-pair}}$ over $\Sigma_I$ defined by

$$L_{\overline{bad}} = \bigcap_{1 \leq j \leq n} \{w \in \Sigma_I^* \mid w \text{ contains no bad } j\text{-pair}\}.$$

**Lemma 11.** *Every $w \in L_{\overline{bad}}$ satisfies $|w| < 2^n$.*

*Proof.* We prove the stronger statement $|w| < 2^m$, where $m$ is the maximal letter occurring in $w$, by induction over $m$. The induction base $m = 1$ is trivial, so let $m > 1$. There cannot be two occurrences of $m$ in $w$, as they would constitute a bad $m$-pair. Accordingly, there is exactly one $m$ in $w$, i.e., we can decompose $w$ into $w = w_\triangleleft\, m\, w_\triangleright$ such that $w_\triangleleft$ and $w_\triangleright$ contain no occurrence of $m$. Thus, the induction hypothesis is applicable and shows $|w_\triangleleft|, |w_\triangleright| < 2^{m-1}$, which implies $|w| < 2^m$. □

**Lemma 12.** *There is a word $w_n \in L_{\overline{bad}}$ with $|w_n| = 2^n - 1$.*

*Proof.* The word $w_n$ is defined inductively via $w_1 = 1$ and $w_m = w_{m-1}\, m\, w_{m-1}$ for $m > 1$. A simple induction shows $w_n \in L_{\overline{bad}}$ and $|w_n| = 2^n - 1$. □

**Theorem 26.** *For all $n > 1$ there is a reachability game $\mathcal{G}_\exists^n$ such that*

- *$|\mathcal{G}_\exists^n| \in O(n)$,*
- *Player $O$ wins $\Gamma_f(\mathcal{G}_\exists^n)$ for some constant delay function $f$, but*
- *Player $I$ wins $\Gamma_f(\mathcal{G}_\exists^n)$ for every delay function $f$ with $f(0) \leq 2^n$.*

*Proof.* We fix input and output alphabets with $\Sigma_I = \Sigma_O = \{1, 2, \dots n\}$, respectively. Then, the reachability game $\mathcal{G}_\exists^n = (\mathcal{A}_\exists^n, \textsc{Reach}(R))$ is played in the arena $\mathcal{A}_\exists^n$, as depicted in Figure 47, with the goal of Player $O$ to reach a vertex in $R = \bigcup_{j=1}^n \{v_j^6, v_j^7\}$. The characteristics of the game work as follows: $\binom{\alpha_0}{\beta_0}\binom{\alpha_1}{\beta_1}\binom{\alpha_2}{\beta_2} \ldots \in \text{Plays}(\mathcal{A}_\exists^n)$ if $\alpha_1\alpha_2\alpha_3 \ldots$ contains a bad $\beta_0$-pair, i.e., with the first move, Player $O$ must pick a $j \in [n]$ such that Player $I$ has produced a bad $j$-pair. Clearly, we have $\mathfrak{A}_n \in O(n)$.

Player $O$ wins $\Gamma_f(\mathcal{G}_\exists^n)$ for every delay function $f$ with $f(0) > 2^n$. In the first round, Player $I$ picks a word $u_0$ such that $u_0$ without its first letter is not in $L_{\overline{bad}}$, as it is too long according to Lemma 11. This allows Player $O$ to pick a bad $j$-pair for some $j$, i.e., Player $O$ wins the play no matter how it is continued.

However, for $f$ with $f(0) \leq 2^n$, Player $I$ has a winning strategy by picking the prefix of the word $1w_n$ from Lemma 12 of length $f(0)$ in the first round. Player $O$ has to answer with some $j \in \Sigma_O$ such that Player $I$ continues by playing some $j' \neq j$ ad infinitum, which ensures that the resulting sequence does not contain a bad $j$-pair. Thus, the play is winning for Player $I$. □

For safety games, we use the same idea as in the reachability case. However, we additionally need to introduce a new letter #. The letter is used to give Player $I$ the possibility to reach a non-accepting vertex. Without the letter, Player $O$ could always stay in the safe region using the same strategy as for reachability games.

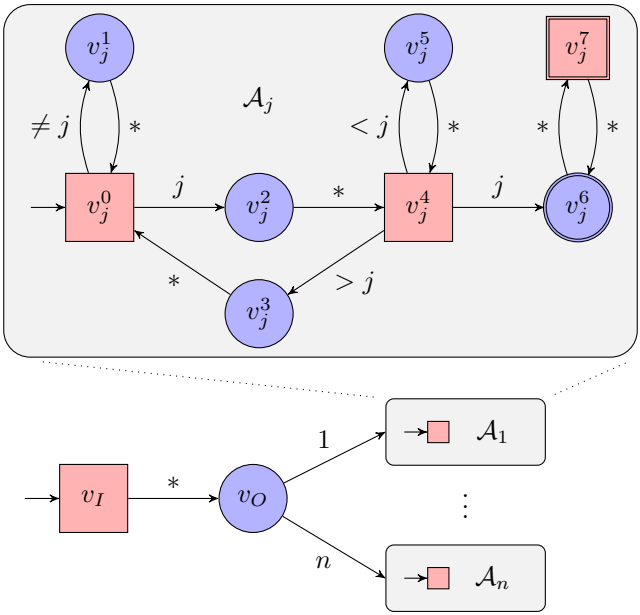



Figure 47: The arena $\mathcal{A}^n_\exists$ of the game $\mathcal{G}^n_\exists$ using the sub-arenas $\mathcal{A}_j$ for $j \in [n]$.

**Theorem 27.** *For all $n > 1$ there is a safety game $G^n_\forall$ such that*

- $|\mathcal{G}^n_\forall| \in O(n)$,
- *Player $O$ wins $\Gamma_f(\mathcal{G}^n_\forall)$ for some constant delay function $f$, but*
- *Player $I$ wins $\Gamma_f(\mathcal{G}^n_\forall)$ for every delay function $f$ with $f(0) \leq 2^n$.*

*Proof.* Let $\Sigma_I = \Sigma_O = \{1, 2, \ldots, n, \#\}$ and let $w_n \in L_{\overline{bad}}$ be defined as given by Lemma 12. Now, consider the extended arena $\mathcal{A}^n_\forall$ using extended sub-arenas $\mathcal{A}_j$, as depicted in Figure 48. The game $\mathcal{G}^n_\forall$ then is defined by $(\mathcal{A}^n_\forall, \textsc{Safety}(S))$ with $S = \{v_I, v_O\} \cup \{v_x^m \mid x \in \Sigma_I, m \in [10], (x, m) \neq (\#, 3)\}$.

For $f(0) > 2^n$, Player $O$ wins the game: assume Player $I$ picks $u_0$ in the first round and let $u_0'$ be $u_0$ without the first letter. If $u_0'$ contains a # preceded by at most one $n$, then Player $O$ answers with # in the first round. If there is more than one $n$ before the first # in $u_0'$, then Player $O$ answers with $n$. Finally, if there is no # in $u_0'$, then by Lemma 11 Player $O$ can pick a $j$ such that $u_0'$ contains a bad $j$-pair. All outcomes are winning for Player $O$.

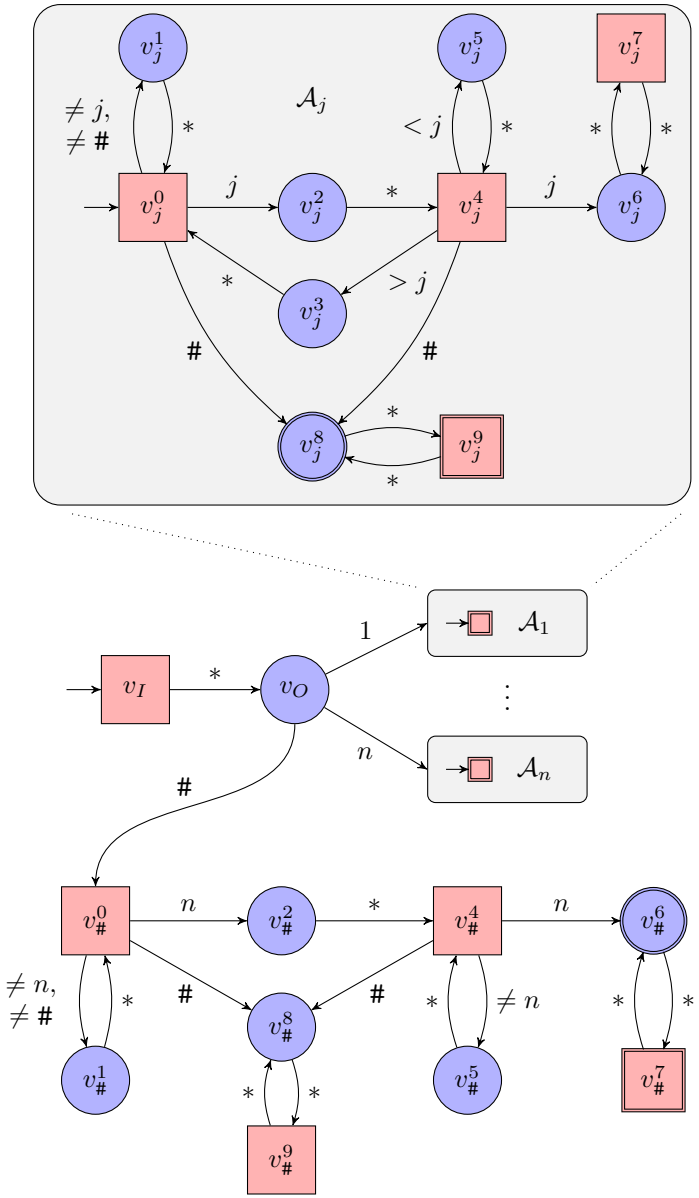



Figure 48: The arena $\mathcal{A}_\forall^n$ of the game $\mathcal{G}_\exists^n$ using the sub-arenas $\mathcal{A}_j$ for $j \in [n]$.

Player $I$ still wins the game for a constant delay function $f$ with $f(0) \leq 2^n$ by picking the prefix of $1w_n$ of length $f(0)$ in the first round: if Player $O$ picks some $j \in \Sigma_O \setminus \{\#\}$ in the first round, then Player $I$ just has to answer with $\#$. Otherwise, if Player $O$ picks $\#$ in the first round, then Player $I$ continues with $n^\omega$. Player $I$ wins in both situations. □

The aforementioned constructions also work for constant-size alphabets, if we encode every $j \in \{1, \ldots, n\}$ in binary with the most significant bit in the first position. Then, the natural ordering on $\{1, \ldots, n\}$ is exactly the lexicographical ordering on the corresponding bit-string representation. Accordingly, we can encode both arenas $\mathcal{A}^n_\forall$ and $\mathcal{A}^n_\exists$ in logarithmic size in $n$, as deciding whether the input represents $j$, is larger than $j$, or smaller than $j$ can be checked bit-wise. Together with a binary decision tree of size $O(n)$ for the initial choice of Player $O$ we obtain games $\mathcal{G}^n_\exists$ and $\mathcal{G}^n_\forall$ whose sizes are in $O(n \log n)$. It is open whether linear-sized games and a constant-sized alphabet can be achieved simultaneously.

# 8 LTL Synthesis with Delay

We showed in the previous sections that parity games with delay can be solved by reducing them to delay free games, as provided by Construction 9. Unfortunately, this reduction comes at the price of an exponential blowup in the size of the game arena, which we showed to be unavoidable in general, even for simpler winning conditions such as reachability or safety. Nevertheless, with the reduction of Construction 9 at hand, we are able to solve delay games in the first place.

As a consequence of our findings, we also obtain immediate upper bounds on the complexity of solving the LTL synthesis problem with delay. Moreover, we obtain an upper bound on the lookahead that may be required by Player $O$. Using the known transformations via automata over infinite words, we first translate the given LTL formula into a parity game, which causes a doubly-exponential blowup in the size of the formula in the worst case [124, 143]. Afterwards, the resulting parity game is considered as a delay game, which then can be solved via the aforementioned reduction. Overall, the complete procedure leads to a triply-exponential upper bound in the worst case. Both, in terms of the complexity of the realizability question and in terms of the required lookahead.

The question that remains is: Is this solution indeed optimal? Or is there a better method that provides improved upper bounds on the complexity and lookahead by utilizing a smarter construction? Unfortunately, we show that

this is not the case, *i.e.*, the corresponding triply-exponential upper bounds are tight.

To this end, we first establish a triply-exponential lower bound on the necessary lookahead in LTL delay games, which matches the aforementioned upper bound. Our proof shares some similarities with the previous proofs of the lower bounds for safety, reachability, and parity games. However, in order to reach a triply-exponential complexity class, it also relies on standard encodings of doubly-exponentially large numbers using *small* LTL formulas and the interaction between the players.

Consider an alphabet $\Sigma$ that contains the propositions $b_0, \ldots, b_{n-1}, b_I, b_O$ and let $w \in (2^\Sigma)^\omega$ and $i \in \mathbb{N}$. We interpret $w(i) \cap \{b_0, \ldots, b_{n-1}\}$ as the binary encoding of a number in $[0, 2^n - 1]$, which we refer to as the address of position $i$. Then there is a formula $\psi_{\text{inc}}$ of quadratic size in $n$ such that $(w, i) \vDash \psi_{\text{inc}}$ if and only if $m + 1 \bmod 2^n = m'$, where $m$ is the address of position $i$ and $m'$ is the address of position $i + 1$. A respective counter that increases with every time step then is specified by the following LTL formula:

$$\psi_0 \;=\; \bigwedge_{j=0}^{n-1} \neg b_j \;\wedge\; \square \psi_{\text{inc}}$$

If we have that $w \vDash \psi_0$, then the $b_j$ form a cyclic addressing of the positions starting at zero, *i.e.*, the address of position $i$ is $i \bmod 2^n$. In this case, we define a block of $w$ as an infix that starts at a position with address zero and ends at the next position with address $2^n - 1$. We interpret the $2^n$ bits $b_I$ of a block as a number $x$ in $R = [0, 2^n - 1]$ and the $2^n$ bits $b_O$ of a block as a number $y$ from the same range $R$. Furthermore, there are *small* formulas that are satisfied at the start of the $i$-th block if and only if $x_i = y_i$ ($x_i < y_i$, respectively). However, we cannot compare numbers from different blocks for equality with *small* formulas. Nevertheless, if $x_i$ is unequal to $x_{i'}$, then there is a single bit that witnesses this, *i.e.*, the bit is one in $x_i$ if and only if it is zero in $x_{i'}$. We check this by letting one of the players specify the address of such a witness (but not the witness itself). The correctness of this claim is then verifiable by a *small* LTL formula.

Also remember that the exponential lower bound on the necessary lookahead for $\omega$-regular delay games is witnessed by avoiding bad $j$-pairs over the the alphabet $1, \ldots, n$ (cf. Definition 31). We adapt this winning condition to the alphabet $R$, which yields a triply-exponential lower bound $2^{|R|}$. The main difficulty of the proof then is the inability of small LTL formulas to compare letters from $R$. We overcome this by exploiting the interaction between the players of the game.

**Theorem 28.** *For every $n > 0$, there is an* LTL *formula $\phi_n$ of size $O(n^2)$ such that*

- *Player $O$ wins $\Gamma_f(\phi_n)$ for some delay function $f$, but*
- *Player $I$ wins $\Gamma_f(\phi_n)$ for every delay function $f$ with $f(0) \leq 2^{2^{2^n}}$.*

*Proof.* We fix some $n > 0$, where in the following we measure all formula sizes in $n$. Furthermore, let $I = \{b_0, \ldots, b_{n-1}, b_I, \#\}$ and $O = \{b_O, \rightarrow, \leftarrow\}$. We assume that $\binom{\alpha}{\beta} \in (\Sigma_I \times \Sigma_O)^\omega$ satisfies $\psi_0$ from above. Then, $\alpha$ induces a sequence $x_0x_1x_2\cdots \in R^\omega$ of numbers encoded by the bits $b_I$ in each block. Similarly, $\beta$ induces a sequence $y_0y_1y_2\cdots \in R^\omega$.

The winning condition is intuitively described as follows: $x_i$ and $x_{i'}$ with $i < i'$ constitute a bad $j$-pair, if $x_i = x_{i'} = j$ and $x_{i''} < j$ for all $i < i'' < i'$. Every sequence $x_0x_1x_2\cdots$ contains a bad $j$-pair, *e.g.*, pick $j$ to be the maximal number occurring infinitely often. In order to win, Player $O$ has to pick $y_0$ such that $x_0x_1x_2\cdots$ contains a bad $y_0$-pair. It is known that this winning condition requires lookahead of length $2^m$ for Player $O$ to win, where $m$ is the largest number.

To be able to specify this condition with a small LTL formula, we have to require Player $O$ to copy $y_0$ ad infinitum, *i.e.*, to pick $y_i = y_0$ for all $i$, and to mark the two positions constituting the bad $y_0$-pair. Furthermore, the winning condition allows Player $I$ to mark one copy error introduced by Player $O$ through specifying its address by a # (which may appear anywhere in $\alpha$). This forces Player $O$ to implement the copying correctly and thus allows a small formula to check that she indeed marks a bad $y_0$-pair. Consider the following properties:

1. # holds at most once. Player $I$ uses # to specify the address where he claims an error.

2. ➜ holds exactly at one position, which has to be the start of a block. Furthermore, we require the two numbers encoded by the propositions $b_I$ and $b_O$ within this block to be equal. Player $O$ uses ➜ to denote the first component of a claimed bad $j$-pair.

3. ← holds exactly at one position, which has to be the start of a block and has to appear at a later position than ➜. Again, we require the two numbers encoded by this block to be equal. Player $O$ uses ← to denote the second component of the claimed bad $j$-pair.

4. For every block between the two marked blocks, we require the number encoded by the $b_I$ to be strictly smaller than the number encoded by the $b_O$.

5. If there is a position $i_{\#}$ marked by #, then there are no two different positions $i \neq i'$ such that the following two conditions are satisfied: the addresses of $i$, $i'$, and $i_{\#}$ are equal and $b_O$ holds at $i$ iff $b_O$ does not hold at $i'$. Such positions witness an error in the copying process by Player $O$, which manifests itself in a single bit, whose address is marked by Player $I$ at any time in the future.

Each of these properties $i \in \{1,2,3,4,5\}$ can be specified by an LTL formula $\psi_i$ of at most quadratic size. Let $\phi_n = \psi_0 \wedge \psi_1 \rightarrow \psi_2 \wedge \psi_3 \wedge \psi_4 \wedge \psi_5$ and fix $n' = 2^{2^n}$ to simplify the notation. We show that Player $O$ wins $\Gamma_f(\phi_n)$ for some triply-exponential constant delay function, but not for any smaller one.

We first show that Player $O$ wins $\Gamma_f(\phi_n)$ for the constant delay function with $f(0) = 2^n \cdot 2^{n'}$. A simple induction shows that every word $w \in R^*$ of length $2^{n'}$ contains a bad $j$-pair for some $j \in R$. Thus, a move $\Sigma_I^{f(0)}$ made by Player $I$ in round 0 interpreted as sequence $x_0 x_1 \cdots x_{2^{n'}-1} \in R^*$ contains a bad $j$-pair for some fixed $j$. Hence, Player $O$'s strategy $\sigma_O$ produces the sequence $j^\omega$ and additionally marks the corresponding bad $j$-pair with ➜ and ➜. Every outcome of a play that is consistent with $\sigma_O$ and satisfies $\psi_0$ also satisfies $\psi_2 \wedge \psi_3 \wedge \psi_4 \wedge \psi_5$, as Player $O$ correctly marks a bad $j$-pair and never introduces a copy-error. Hence, $\sigma_O$ is a winning strategy for Player $O$.

It remains to show that Player $I$ wins $\Gamma_f(\phi_n)$, if

$$f(0) \;\leq\; 2^n \cdot (2^{n'} - 2) \;\geq\; 2^{n'} \;=\; 2^{2^{2^n}}.$$

Let $w_{n'} \in R^*$ be recursively defined via $w_0 = 0$ and $w_j = w_{j-1}\, j\, w_{j-1}$. A simple induction shows that $w_{n'}$ does not contain a bad $j$-pair, for every $j \in R$, and that $|w_{n'}| = 2^{n'} - 1$.

Consider the following strategy $\sigma_I$ for Player $I$ in $\Gamma_f(\phi_n)$: $\sigma_I$ ensures that $\psi_0$ is satisfied by the $b_j$, which fixes them uniquely to implement a cyclic addressing starting at zero. Furthermore, he picks the $b_I$'s so that the sequence of numbers $x_0 x_1 \cdots x_\ell$ he generates during the first $2^n$ rounds is a prefix of $w_{n'}$. This is possible, as each $x_i$ is encoded by $2^n$ bits and by the choice of $f(0)$. As a response during the first $2^n$ rounds, Player $O$ determines some number $y \in R$. During the next rounds, Player $I$ finishes $w_{n'}$ and then picks some fixed $x \neq y$ ad infinitum (while still implementing the cyclic addressing). In case Player $O$ picks both markings ➜ and ➜ in way that is consistent with properties 2, 3, and 4 as above, let $y_0, y_1 \cdots, y_i$ be the sequence of numbers

picked by her up to and including the number marked by ←. If they are not all equal, then there is an address that witnesses the difference between two of these numbers. Player $I$ then marks exactly one position with the same address using #. If this is not the case, he never marks a position with #.

Now, consider an outcome of a play that is consistent with $\sigma_I$ and let $x_0x_1x_2\cdots \in R^\omega$ and $y_0y_1y_2\cdots \in R^\omega$ be the sequences of numbers induced by the outcome. By definition of $\sigma_I$, the premise $\psi_0 \wedge \psi_1$ of $\phi_n$ is satisfied and $x_0x_1x_2\cdots = w_{n'} \cdot x^\omega$ for some $x \neq y_0$.

If Player $O$ never uses her markers → and ← in a way that satisfies $\psi_2 \wedge \psi_3 \wedge \psi_4$, then Player $I$ wins the play, as it satisfies the premise of $\phi_n$, but not the consequence. Thus, it remains to consider the case where the outcome satisfies $\psi_2 \wedge \psi_3 \wedge \psi_4$. Let $y_0y_1\cdots y_i$ be the sequence of numbers picked by her up to and including the number marked by ←. Assume we have $y_0 = y_1 = \cdots = y_i$. Then, → and ← specify a bad $y_0$-pair, as implied by $\psi_2 \wedge \psi_3 \wedge \psi_4$ and the equality of the $y_j$. As $w_{n'}$ does not contain a bad $y_0$-pair, we conclude $y_0 = x$. However, $\sigma_I$ ensures $y_0 \neq x$. Hence, our assumption is false, *i.e.*, the $y_j$ are not all equal. In this situation, $\sigma_I$ marks a position whose address witnesses this difference. This implies that $\psi_5$ is not satisfied, *i.e.*, the play is winning for Player $I$. Hence, $\sigma_I$ is winning for him. □

We conclude with the proof of 3ExpTime-completeness of solving the LTL realizability problem under the assumption of delay. The proof consists of a combination of techniques developed for the lower bound on the lookahead, as presented above, and of the techniques from the proof of Theorem 23.

**Theorem 29.** *The following problem is* 3ExpTime*-complete: given an* LTL *formula $\varphi$, does Player $O$ win $\Gamma_f(\varphi)$ for some delay function $f$?*

*Proof.* Membership directly follows from the aforementioned reduction of a given LTL formula to a parity game combined with Construction 9. Hence, it only remains to prove hardness. To this end, let

$$\mathcal{TM} = (Q, Q_\exists, Q_\forall, \Sigma, q_I, \Delta, q_A, q_R)$$

be an alternating doubly-exponential space Turning machine with transition relation $\Delta \subseteq Q \times \Sigma \times Q \times \Sigma \times \{-1, +1\}$ and accepting and rejecting states $q_A$ and $q_R$, which we assume w.l.o.g. to have self-loops. Furthermore, let $p$ be a polynomial such that $2^{2^p}$ bounds the space-consumption of $\mathcal{TM}$ and let $x \in \Sigma^*$ be an input. Fix $n = p(|x|)$. We construct an LTL formula $\varphi$ (of

polynomial size in $n + |\Delta|$) such that Player $O$ wins $\Gamma_f(\varphi)$ for some $f$ if and only if $\mathcal{TM}$ rejects $x$. This suffices, as A2ExpSpace = 3ExpTime is closed under complement.

We pick $I = \{b_0, \ldots, b_{n-1}, b_I, \#, C, N\} \cup \Sigma \cup Q \cup \Delta$ and $O = \{✗, ➜, ➜\} \cup \Delta$. Let $\psi_0$ be the formula that requires Player $I$ to implement the cyclic addressing of length $2^n$ starting at zero using the bits $b_j$, as defined above. In the following, we only consider outcomes that satisfy $\psi_0$. Also, blocks are defined as before and we again interpret the bits $b_I$ of a block as a number in $R = [0, 2^{2^n} - 1]$. Then, there is an LTL formula $\theta_0$ of polynomial size that allows Player $O$ to use the error mark ✗ to force Player $I$ to implement a cyclic addressing of the blocks of length $2^{2^n}$ starting at zero [153]. In particular, the formula $\theta_0$ holds if and only if the first occurrence of ✗ marks a position witnessing that the addressing is not implemented correctly. Hence, Player $O$ can satisfy $\theta_0$ if and only if Player $I$ incorrectly implements the cyclic addressing of the blocks of length $2^{2^n}$.

Assume Player $I$ implements both addressings correctly: then, a superblock is an infix starting with a block encoding $0 \in R$ and that ends one position before the next block that encodes $0 \in R$, *i.e.*, each superblock consists of $2^{2^n}$ blocks. We use such superblocks to encode configurations of $\mathcal{TM}$ on $x$ by placing the cell contents at the starts of the blocks.

Intuitively, Player $I$ produces configurations of $\mathcal{TM}$ and is in charge of existential states, while Player $O$ controls the universal ones and checks the configurations for correctness using the marks ➜ and ➜ to indicate cells, where the configurations are not updated correctly. To account for the lookahead in the game, which means that Player $O$ picks her transitions to apply asynchronously, Player $I$ is able to copy configurations in order to wait for Player $O$'s choice. Player $O$ checks this copying process for correctness.

To this end, we use the two propositions $C$ and $N$ to denote whether a copy or a successor configuration follows. If Player $I$ produces a new universal configuration, then Player $O$ has to pick some transition from $\Delta \subseteq O$, which should be applied to this configuration, possibly after some copies. If Player $I$ copies a configuration ad infinitum, then he loses.

Consider the following assumptions on Player $I$'s behavior beside $\psi_0$:

1. At every start of a block, exactly one proposition from $\Sigma$ holds and at most one from $Q$. Also, in each superblock, there is exactly one start of a block where a proposition from $Q$ holds. If this holds, then each superblock encodes a configuration of $\mathcal{TM}$ of length $2^{2^{p(|x|)}}$.

2. The configuration encoded by the first superblock is the initial one of $\mathcal{TM}$ on $x$.

3. At each start of a superblock, either $C$ or $N$ holds, starting with $N$ at the first superblock. Furthermore, we require $N$ to hold infinitely often at such positions.

4. At each start of a superblock that encodes an existential configuration, exactly one proposition from $\Delta$ holds, which has to be applicable to the configuration.

5. There is at most one position where # holds. This is used by Player $I$ to check Player $O$'s error claim and is implemented as in the proof of Theorem 28.

Each of these properties $1 \leq i \leq 5$ can be captured by an LTL formula $\psi_i$ of polynomial size. Furthermore, let $\theta_1$ be an LTL formula that expresses the following, which has to be guaranteed by Player $O$: at each start of a superblock that encodes a universal configuration, exactly one proposition from $\Delta$ holds, which has to be applicable to the configuration.

Now, we define what it means for Player $O$ to mark an incorrectly updated configuration: the conjunction of the following properties has to be satisfied, where we assume that $\bigwedge_{i=0}^{5} \psi_i \wedge \neg\theta_0 \wedge \theta_1$ holds, as this is the only case where the formula to be defined is relevant.

1. ➜ and ➜ hold both exactly once, each at the start of a block. Furthermore, ➜ appears one superblock after the superblock in which ➜ appears.

2. If there is a position $i_{\#}$ marked by #, then there are no two different positions $i$ being in the superblock of ➜ and $i'$ being in the superblock of ➜ such that the following two conditions are satisfied: the addresses of $i$, $i'$, and $i_{\#}$ are equal and $b_I$ holds at $i$ if and only if $b_I$ does not hold at $i'$. Such positions witness that Player $O$ has not marked the same cell of the two subsequent configurations. This manifests itself in a single bit $b_I$, whose address is marked by Player $I$.

3. If $C$ holds at the start of ➜'s superblock, then there has to be a proposition from $\Sigma \cup Q$ that holds at the position marked by ➜ if and only if it does not hold at the position marked by ➜.

4. If $N$ holds at the start of ➜'s superblock, then let $\delta \in \Delta$ be the unique transition holding at the last occurrence of $N$ before the start of this superblock. Furthermore, let $c_m$ be the tape content (a letter from $\Sigma$ and possibly a state from $Q$) encoded at the position marked with ➜ and let $c_\ell$ ($c_r$) be the cell content encoded in the start of the previous

(next) block. Then, $c = (c_\ell, c_m, c_r)$ uniquely determines the cell content of the middle cell after applying the transition function $\delta$. We require that the cell content encoded at the position marked by ← is different from $c$.

Let $\theta_2$ be an LTL formula capturing the conjunction of these properties, which can be constructed such that it has polynomial size.

Now, consider the LTL delay game $\Gamma_f(\varphi)$ with:

$$\varphi \;:=\; \bigwedge_{i=0}^{5} \psi_i \;\rightarrow\; \theta_1 \wedge \big(\theta_0 \vee \theta_2 \vee \Diamond q_R\big)$$

We show that $\mathcal{TM}$ rejects $x$ if and only if Player $O$ wins $\Gamma_f(\varphi)$ for some $f$.

First, assume that $\mathcal{TM}$ rejects $x$ and let $f$ be the constant delay function with $f(0) = 2 \cdot 2^{2^n}$. We show that Player $O$ wins $\Gamma_f(\varphi)$. As long as Player $I$ implements both addressings correctly and produces legal configurations as required by the premise of $\varphi$, Player $O$ has enough lookahead to correctly claim the first error introduced by Player $I$, no matter whether he increments the superblock addressing incorrectly or updates a configuration incorrectly. Her strategy is to place the markers ✗, ➔, ← at appropriate positions. Furthermore, she has access to the whole encoding of each universal configuration whose successor she has to determine. This allows her to simulate the rejecting run of $\mathcal{TM}$ on $x$, which reaches the rejecting state $q_R$, no matter which existential transitions Player $I$ picks. Thus, he has to introduce an error in order to win, which Player $O$ can detect using the lookahead. If Player $I$ does not introduce an error, the play reaches a rejecting configuration. In every case Player $O$ wins.

For the converse direction, we show the contrapositive. Assume that $\mathcal{TM}$ accepts $x$ and let $f$ be an arbitrary delay function. We show that Player $I$ wins $\Gamma_f(\varphi)$. Player $I$ implements both addressings correctly, starts with the initial configuration, and picks the successor configuration of an existential one according to the accepting run. Also, he copies universal configurations to obtain a play prefix in which Player $O$ has to determine the transition she wants to apply in this configuration. Thus, he will eventually produce an accepting configuration without ever introducing an error. In particular, a rejecting state is never reached and Player $O$ cannot successfully claim an error: the superblock addressing is correctly implemented and if she claims an erroneous update of the configurations, she has to mark different cells, as there are no such incorrect updates. This can be detected by Player $I$ by placing the # at a witnessing address. In either case, Player $I$ wins the resulting play. □

# 9 Discussion

Our results show that it is possible to identify delay requirements that cause faulty temporal logic specification to be unrealizable. This identification comes, however, at the cost of an exponential blowup being unavoidable in general. According to the utilization of synthesis for the development of reactive systems, delay games, thus, provide an effective support in helping the developers. In case of an unrealizable specification, the delay-free game that is won by the environment player can be automatically *upgraded* to a delay game, which instead may return a realizability result. In this case, the developer gets immediate feedback towards the cause for unrealizability. Moreover, the winning strategy of the delay game can be utilized to present the developer with a first resolution strategy. However, if the specification still remains unrealizable the developer still gets the feedback that the problem is not caused due to delay. In any case, delay games provide us with an effective tool for debugging faulty specifications.

With our last result of this chapter, we settled the complexity of solving delay games with LTL winning conditions. These results also can directly be leveraged towards TSL. Reconsidering the CEGAR refinement loop for TSL, it turns out that the utilized approximation is easy to lift towards delay, while still preserving the soundness of the overall approach. If the approximated LTL specification turns out to be unrealizable in the delay free case, but is realizable under the assumption of delay, then the corresponding winning strategy still lifts towards TSL. The argument is the same as for the proof of Theorem 11, since TSL only weakens the abilities of the environment in contrast to LTL. Hence, if a winning strategy is found for the *stronger* LTL environment, it also wins in the *weaker* TSL case. Note that delay games also weaken the environment, which is why both approaches counteract against each other at this point. However, if the approximated specification is unrealizable, we only swap to delay games for testing realizability under the assumption of delay once. If this fails, we again enter the refinement as before. Remember that solving delay games is a decidable problem and, thus, can be easily integrated with our CEGAR approach.

While the complexity classes of solving games with delay have been settled by our results, open questions still rise regarding the representation of winning strategies for games with delay. While different possibilities already have been explored [156], and most of the theoretical problems have been identified [79], there still is no decent solution. The consequences also affect respective debugging techniques for TSL, since the resulting strategy still must be translated to an FRP framework. FRP, also utilizing delay as part

of cells or more advanced `hold` operations, thus, may give a good indication for the requirements of the strategy representation. These considerations are, however, out of scope of this thesis and are considered as future work.

# Chapter VI

# Conclusions

In this thesis, we considered the automatic creation of reactive systems from temporal logic specifications solely describing the behavior of the system to be satisfied by any possible infinite system execution. This process is known as reactive synthesis and introduces new development concepts, but also new challenges in comparison to manual system development. We considered three major challenges: data abstraction during specification creation, intend verification of realizable specifications and the debugging of unrealizable specifications. We showed how exactly these challenges are connected with each other and presented new approaches for tackling each of them.

Regarding the required abstraction techniques, we introduced Temporal Stream Logic, a logic that leverages a clean separation between the description of control and data manipulations. Therefore, the logic focuses on the reactive control, while keeping data descriptions and transformations abstract. For the verification of the designer's intend, in case of a realizability result, we considered output sensitive synthesis approaches. The presented methods support the introduction of additional output related quality metrics, which allow to specifically target the synthesizer towards certainly selecting the considerated solution in the end. Finally, in the case of an unrealizable specification, we considered the introduction of lookahead, which weakens the strict alternation between the system and environment players in the underlying infinite two-player game. With lookahead being utilized, specifications that are unrealizable at first can become realizable at a second glance. Therefore, the introduced extension not only offers an automatic process for resolving the unrealiziablity result, but also provides direct insights for the developer in order to identify the original cause of unrealizability.

Our results not only strictly improve on the currently available state-of-the-art approaches, but also highlight the variety of challenges that need to be overcome for making synthesis based reactive system development a standardized method, which can be efficiently applied in practice in the end.

# Bibliography


[1] R. Alur and S. La Torre. Deterministic generators and games for ltl fragments. ACM Trans. Comput. Log., 5(1):1–25, 2004.

[2] R. Alur, S. Moarref, and U. Topcu. Counter-strategy guided refinement of GR(1) temporal logic specifications. In Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013, pages 26–33. IEEE, 2013.

[3] R. Alur, S. Moarref, and U. Topcu. Pattern-based refinement of assume-guarantee specifications in reactive synthesis. In C. Baier and C. Tinelli, editors, Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings, volume 9035 of Lecture Notes in Computer Science, pages 501–516. Springer, 2015.

[4] H. Apfelmus. Reactive-banana. Haskell library available at `http://www.haskell.org/haskellwiki/Reactive-banana`, 2012.

[5] H. Apfelmus. Threepenny-gui. `https://wiki.haskell.org/Threepenny-gui`, 2013.

[6] ARM Ltd. AMBA Specification (rev. 2). available at `www.arm.com`, 1999.

[7] C. Baaij. Digital circuit in CλaSH: functional specifications and type-directed synthesis. PhD thesis, University of Twente, 1 2015. eemcs-eprint-23939.

[8] C. Baier and J. Katoen. Principles of model checking. MIT Press, 2008.

[9] M. Bärenz and I. Perez. Rhine: FRP with type-level clocks. In N. Wu, editor, Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2018, St. Louis, MO, USA, September 27-17, 2018, pages 145–157. ACM, 2018.

[10] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL - a tool suite for automatic verification of real-time systems. In R. Alur, T. A. Henzinger, and E. D. Sontag, editors, Hybrid Systems

III: Verification and Control, Proceedings of the DIMACS/SYCON Workshop on Verification and Control of Hybrid Systems, October 22-25, 1995, Ruttgers University, New Brunswick, NJ, USA, volume 1066 of Lecture Notes in Computer Science, pages 232–243. Springer, 1995.

[11] M. Bezem, editor. Computer Science Logic, 25th International Workshop / 20th Annual Conference of the EACSL, CSL 2011, September 12-15, 2011, Bergen, Norway, Proceedings, volume 12 of LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

[12] A. Biere. The AIGER And-Inverter Graph (AIG) format version 20071012. Technical Report Report 07/1, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, 2007.

[13] S. Blackheath and A. Jones. Functional reactive programming. Manning Publications, 2016.

[14] R. Bloem, S. J. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Automatic hardware synthesis from specifications: a case study. In R. Lauwereins and J. Madsen, editors, 2007 Design, Automation and Test in Europe Conference and Exposition, DATE 2007, Nice, France, April 16-20, 2007, pages 1188–1193. EDA Consortium, San Jose, CA, USA, 2007.

[15] R. Bloem, S. Jacobs, and A. Khalimov. Parameterized synthesis case study: AMBA AHB. In K. Chatterjee, R. Ehlers, and S. Jha, editors, Proceedings 3rd Workshop on Synthesis, SYNT 2014, Vienna, Austria, July 23-24, 2014., volume 157 of EPTCS, pages 68–83, 2014.

[16] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. J. Comput. Syst. Sci., 78(3):911–938, 2012.

[17] A. Bohy, V. Bruyère, E. Filiot, N. Jin, and J. Raskin. Acacia+, a tool for LTL synthesis. In P. Madhusudan and S. A. Seshia, editors, Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings, volume 7358 of Lecture Notes in Computer Science, pages 652–657. Springer, 2012.

[18] G. P. Brat, D. Drusinsky, D. Giannakopoulou, A. Goldberg, K. Havelund, M. R. Lowry, C. S. Pasareanu, A. Venet, W. Visser, and R. Washington. Experimental evaluation of verification and validation

tools on martian rover software. Formal Methods in System Design, 25(2-3):167–198, 2004.

[19] T. Brázdil, K. Chatterjee, J. Kretínský, and V. Toman. Strategy representation by decision trees in reactive synthesis. In D. Beyer and M. Huisman, editors, Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I, volume 10805 of Lecture Notes in Computer Science, pages 385–407. Springer, 2018.

[20] G. Cabodi, C. Loiacono, M. Palena, P. Pasini, D. Patti, S. Quer, D. Vendraminetto, A. Biere, and K. Heljanko. Hardware Model Checking Competition 2014: An Analysis and Comparison of Solvers and Benchmarks. volume 9, pages 135–172, 2016.

[21] A. Carayol and C. Löding. MSO on the infinite binary tree: Choice and order. In J. Duparc and T. A. Henzinger, editors, Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings, volume 4646 of Lecture Notes in Computer Science, pages 161–176. Springer, 2007.

[22] A. Carayol and C. Löding. Uniformization in automata theory. In P. Schroeder-Heister, G. Heinzmann, W. Hodges, and P. E. Bour, editors, CLMPS 2012, pages 153–178. College Publications, London, 2015.

[23] A. K. Chandra, D. Kozen, and L. J. Stockmeyer. Alternation. J. ACM, 28(1):114–133, 1981.

[24] S. Chandra, P. Godefroid, and C. Palm. Software model checking in practice: an industrial case study. In Tracz et al. [139], pages 431–441.

[25] K. Chatterjee, T. A. Henzinger, and B. Jobstmann. Environment assumptions for synthesis. In F. van Breugel and M. Chechik, editors, CONCUR 2008 - Concurrency Theory, 19th International Conference, CONCUR 2008, Toronto, Canada, August 19-22, 2008. Proceedings, volume 5201 of Lecture Notes in Computer Science, pages 147–161. Springer, 2008.

[26] M. Chen, M. Fränzle, Y. Li, P. N. Mosaad, and N. Zhan. What's to come is still unsure - synthesizing controllers resilient to delayed interaction. In Lahiri and Wang [91], pages 56–74.

[27] C. Cheng, C. Huang, H. Ruess, and S. Stattelmann. G4LTL-ST: automatic generation of PLC programs. In A. Biere and R. Bloem, editors, Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings, volume 8559 of Lecture Notes in Computer Science, pages 541–549. Springer, 2014.

[28] A. Cimatti, M. Roveri, V. Schuppan, and A. Tchaltsev. Diagnostic information for realizability. In F. Logozzo, D. A. Peled, and L. D. Zuck, editors, Verification, Model Checking, and Abstract Interpretation, 9th International Conference, VMCAI 2008, San Francisco, USA, January 7-9, 2008, Proceedings, volume 4905 of Lecture Notes in Computer Science, pages 52–67. Springer, 2008.

[29] E. M. Clarke, E. A. Emerson, and J. Sifakis. Model checking: algorithmic verification and debugging. Commun. ACM, 52(11):74–84, 2009.

[30] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the futurebus+ cache coherence protocol. In D. Agnew, L. J. M. Claesen, and R. Camposano, editors, Computer Hardware Description Languages and their Applications, Proceedings of the 11th IFIP WG10.2 International Conference on Computer Hardware Description Languages and their Applications - CHDL '93, sponsored by IFIP WG10.2 and in cooperation with IEEE COMPSOC, Ottawa, Ontario, Canada, 26-28 April, 1993, volume A-32 of IFIP Transactions, pages 15–30. North-Holland, 1993.

[31] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. J. ACM, 50(5):752–794, 2003.

[32] A. Courtney, H. Nilsson, and J. Peterson. The yampa arcade. In Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2003, Uppsala, Sweden, August 28, 2003, pages 7–18. ACM, 2003.

[33] E. Czaplicki and S. Chong. Asynchronous functional reactive programming for guis. In H. Boehm and C. Flanagan, editors, ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013, pages 411–422. ACM, 2013.

[34] R. Dimitrova, B. Finkbeiner, and H. Torfah. Synthesizing approximate implementations for unrealizable specifications. In I. Dillig and

S. Tasiran, editors, Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I, volume 11561 of Lecture Notes in Computer Science, pages 241–258. Springer, 2019.

[35] A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu. Spot 2.0 - A framework for LTL and \omega - automata manipulation. In C. Artho, A. Legay, and D. Peled, editors, Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings, volume 9938 of Lecture Notes in Computer Science, pages 122–129, 2016.

[36] C. Elliott and P. Hudak. Functional reactive animation. In S. L. P. Jones, M. Tofte, and A. M. Berman, editors, Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997., pages 263–273. ACM, 1997.

[37] E. A. Emerson and J. Y. Halpern. "sometimes" and "not never" revisited: on branching versus linear time temporal logic. J. ACM, 33(1):151–178, 1986.

[38] E. A. Emerson and C. S. Jutla. Tree automata, mu-calculus and determinacy (extended abstract). In 32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991, pages 368–377. IEEE Computer Society, 1991.

[39] L. Erkök. Value recursion in monadic computations. PhD thesis, OGI School of Science & Engineering at OHSU, 2002.

[40] P. Faymonville, B. Finkbeiner, M. N. Rabe, and L. Tentrup. Encodings of bounded synthesis. In A. Legay and T. Margaria, editors, Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I, volume 10205 of Lecture Notes in Computer Science, pages 354–370, 2017.

[41] P. Faymonville, B. Finkbeiner, and L. Tentrup. Bosy: An experimentation framework for bounded synthesis. In R. Majumdar and V. Kuncak, editors, Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part

II, volume 10427 of Lecture Notes in Computer Science, pages 325–332. Springer, 2017.

[42] E. Filiot, N. Jin, and J. Raskin. Antichains and compositional algorithms for LTL synthesis. Formal Methods in System Design, 39(3):261–296, 2011.

[43] B. Finkbeiner and S. Jacobs. Lazy synthesis. In V. Kuncak and A. Rybalchenko, editors, Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings, volume 7148 of Lecture Notes in Computer Science, pages 219–234. Springer, 2012.

[44] B. Finkbeiner, F. Klein, R. Piskac, and M. Santolucito. Vehicle platooning simulations with functional reactive programming. In Proceedings of the 1st International Workshop on Safe Control of Connected and Autonomous Vehicles, SCAV@CPSWeek 2017, Pittsburgh, PA, USA, April 21, 2017, pages 43–47. ACM, 2017.

[45] B. Finkbeiner and S. Schewe. Bounded synthesis. STTT, 15(5-6):519–539, 2013.

[46] B. Finkbeiner and L. Tentrup. Detecting unrealizability of distributed fault-tolerant systems. Logical Methods in Computer Science, 11(3), 2015.

[47] W. Fridman, C. Löding, and M. Zimmermann. Degrees of lookahead in context-free infinite games. In Bezem [11], pages 264–276.

[48] J. Friedman. Alonzo church. application of recursive arithmetic to the problem of circuit synthesissummaries of talks presented at the summer institute for symbolic logic cornell university, 1957, 2nd edn., communications research division, institute for defense analyses, princeton, nj, 1960, pp. 3–50. 3a-45a. The Journal of Symbolic Logic, 28(4):289–290, 1963.

[49] Y. Fukaya and N. Yoshiura. Extracting environmental constraints in reactive system specifications. In O. Gervasi, B. Murgante, S. Misra, M. L. Gavrilova, A. M. A. C. Rocha, C. M. Torre, D. Taniar, and B. O. Apduhan, editors, Computational Science and Its Applications - ICCSA 2015 - 15th International Conference, Banff, AB, Canada, June 22-25, 2015, Proceedings, Part IV, volume 9158 of Lecture Notes in Computer Science, pages 671–685. Springer, 2015.

[50] G. Geier, P. Heim, F. Klein, and B. Finkbeiner. Syntroids: Synthesizing a game for fpgas using temporal logic specifications. In C. W. Barrett and J. Yang, editors, 2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019, pages 138–146. IEEE, 2019.

[51] S. Gélineau. FRPzoo - comparing many FRP implementations by reimplementing the same toy app in each. `https://github.com/gelisam/frp-zoo`, 2016.

[52] C. Gerstacker. Bounded Synthesis of Reactive Programs, 2017. Bachelor's Thesis.

[53] C. Gerstacker, F. Klein, and B. Finkbeiner. Bounded synthesis of reactive programs. In Lahiri and Wang [91], pages 441–457.

[54] P. Godefroid. Model checking for programming languages using verisoft. In P. Lee, F. Henglein, and N. D. Jones, editors, Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997, pages 174–186. ACM Press, 1997.

[55] P. Godefroid. Verisoft: A tool for the automatic analysis of concurrent reactive software. In O. Grumberg, editor, Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings, volume 1254 of Lecture Notes in Computer Science, pages 476–479. Springer, 1997.

[56] E. Grädel, W. Thomas, and T. Wilke, editors. Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001], volume 2500 of Lecture Notes in Computer Science. Springer, 2002.

[57] Y. Gurevich and S. Shelah. Rabin's uniformization problem. J. Symb. Log., 48(4):1105–1119, 1983.

[58] D. Harel and A. Pnueli. On the Development of Reactive Systems. In K. R. Apt, editor, Logics and Models of Concurrent Systems, pages 477–498. Springer Berlin Heidelberg, 1985.

[59] K. Havelund and T. Pressburger. Model checking JAVA programs using JAVA pathfinder. STTT, 2(4):366–381, 2000.

[60] K. Havelund, A. Skou, K. G. Larsen, and K. Lund. Formal modeling and analysis of an audio/video protocol: an industrial case study using UPPAAL. In Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97), December 3-5, 1997, San Francisco, CA, USA, pages 2–13. IEEE Computer Society, 1997.

[61] C. Helbling and S. Z. Guyer. Juniper: a functional reactive programming language for the arduino. In Janin and Sperber [74], pages 8–16.

[62] M. Holtmann, L. Kaiser, and W. Thomas. Degrees of lookahead in regular infinite games. Logical Methods in Computer Science, 8(3), 2012.

[63] G. J. Holzmann. The model checker SPIN. IEEE Trans. Software Eng., 23(5):279–295, 1997.

[64] F. A. Hosch and L. H. Landweber. Finite delay solutions for sequential conditions. In ICALP, pages 45–60, 1972.

[65] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming. In J. Jeuring and S. L. Peyton Jones, editors, Advanced Functional Programming, 4th International School, AFP 2002, Oxford, UK, August 19-24, 2002, Revised Lectures, volume 2638 of Lecture Notes in Computer Science, pages 159–187. Springer, 2002.

[66] P. Hudak, S. L. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. H. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, R. B. Kieburtz, R. S. Nikhil, W. Partain, and J. Peterson. Report on the programming language haskell, A non-strict, purely functional language. SIGPLAN Notices, 27(5):1, 1992.

[67] J. Hughes. Generalising monads to arrows. Sci. Comput. Program., 37(1-3):67–111, 2000.

[68] S. Jacobs, N. Basset, R. Bloem, R. Brenguier, M. Colange, P. Faymonville, B. Finkbeiner, A. Khalimov, F. Klein, T. Michaud, G. A. Pérez, J. Raskin, O. Sankur, and L. Tentrup. The 4th reactive synthesis competition (SYNTCOMP 2017): Benchmarks, participants & results. In D. Fisman and S. Jacobs, editors, Proceedings Sixth Workshop on Synthesis, SYNT@CAV 2017, Heidelberg, Germany, 22nd July 2017., volume 260 of EPTCS, pages 116–143, 2017.

[69] S. Jacobs and R. Bloem. Parameterized synthesis. Logical Methods in Computer Science, 10(1), 2014.

[70] S. Jacobs, R. Bloem, R. Brenguier, A. Khalimov, F. Klein, R. Könighofer, J. Kreber, A. Legg, N. Narodytska, G. A. Pérez, J. Raskin, L. Ryzhyk, O. Sankur, M. Seidl, L. Tentrup, and A. Walker. The 3rd reactive synthesis competition (SYNTCOMP 2016): Benchmarks, participants & results. In Piskac and Dimitrova [113], pages 149–177.

[71] S. Jacobs, R. Bloem, M. Colange, P. Faymonville, B. Finkbeiner, A. Khalimov, F. Klein, M. Luttenberger, P. J. Meyer, T. Michaud, M. Sakr, S. Sickert, L. Tentrup, and A. Walker. The 5th reactive synthesis competition (SYNTCOMP 2018): Benchmarks, participants & results. CoRR, abs/1904.07736, 2019.

[72] S. Jacobs, R. Bloem, M. Colange, P. Faymonville, B. Finkbeiner, A. Khalimov, F. Klein, M. Luttenberger, P. J. Meyer, T. Michaud, M. Sakr, S. Sickert, L. Tentrup, and A. Walker. The 5th reactive synthesis competition (SYNTCOMP 2018): Benchmarks, participants & results. CoRR, abs/1904.07736, 2019.

[73] S. Jacobs, F. Klein, and S. Schirmer. A high-level LTL synthesis format: TLSF v1.1. In Piskac and Dimitrova [113], pages 112–132.

[74] D. Janin and M. Sperber, editors. Proceedings of the 4th International Workshop on Functional Art, Music, Modelling, and Design, FARM@ICFP 2016, Nara, Japan, September 24, 2016. ACM, 2016.

[75] G. Jing, T. Tosun, M. Yim, and H. Kress-Gazit. An end-to-end system for accomplishing tasks with modular robots. In D. Hsu, N. M. Amato, S. Berman, and S. A. Jacobs, editors, Robotics: Science and Systems XII, University of Michigan, Ann Arbor, Michigan, USA, June 18 - June 22, 2016, 2016.

[76] B. Jobstmann. Applications and Optimizations for LTL Synthesis. PhD thesis, Graz University of Technology, 03 2007.

[77] D. B. Johnson. Finding all the elementary circuits of a directed graph. SIAM J. Comput., 4(1):77–84, 1975.

[78] R. Kazhamiakin, M. Pistore, and M. Roveri. Formal verification of requirements using SPIN: A case study on web services. In 2nd International Conference on Software Engineering and Formal Methods (SEFM 2004), 28-30 September 2004, Beijing, China, pages 406–415. IEEE Computer Society, 2004.

[79] F. Klein and M. Zimmermann. What are strategies in delay games? borel determinacy for games with lookahead. In S. Kreutzer, editor, 24th EACSL Annual Conference on Computer Science Logic, CSL 2015, September 7-10, 2015, Berlin, Germany, volume 41 of LIPIcs, pages 519–533. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[80] F. Klein and M. Zimmermann. How much lookahead is needed to win infinite games? Logical Methods in Computer Science, 12(3), 2016.

[81] F. Klein and M. Zimmermann. Prompt delay. In A. Lal, S. Akshay, S. Saurabh, and S. Sen, editors, 36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2016, December 13-15, 2016, Chennai, India, volume 65 of LIPIcs, pages 43:1–43:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[82] J. C. Knight. Safety critical systems: challenges and directions. In Tracz et al. [139], pages 547–550.

[83] R. Könighofer, G. Hofferek, and R. Bloem. Debugging unrealizable specifications with model-based diagnosis. In S. Barner, I. G. Harris, D. Kroening, and O. Raz, editors, Hardware and Software: Verification and Testing - 6th International Haifa Verification Conference, HVC 2010, Haifa, Israel, October 4-7, 2010. Revised Selected Papers, volume 6504 of Lecture Notes in Computer Science, pages 29–45. Springer, 2010.

[84] R. Könighofer, G. Hofferek, and R. Bloem. Debugging formal specifications: a practical approach using model-based diagnosis and counter-strategies. STTT, 15(5-6):563–583, 2013.

[85] H. Kress-Gazit and H. Torfah. The challenges in specifying and explaining synthesized implementations of reactive systems. In B. Finkbeiner and S. Kleinberg, editors, Proceedings 3rd Workshop on formal reasoning about Causation, Responsibility, and Explanations in Science and Technology, CREST@ETAPS 2018, Thessaloniki, Greece, 21st April 2018., volume 286 of EPTCS, pages 50–64, 2018.

[86] S. Kripke. A completeness theorem in modal logic. J. Symb. Log., 24(1):1–14, 1959.

[87] O. Kupferman. Recent challenges and ideas in temporal synthesis. In M. Bieliková, G. Friedrich, G. Gottlob, S. Katzenbeisser, and G. Turán, editors, SOFSEM 2012: Theory and Practice of Computer Science - 38th Conference on Current Trends in Theory and Practice of Computer Science, Špindlerův Mlýn, Czech Republic, January 21-27, 2012. Proceedings, volume 7147 of Lecture Notes in Computer Science, pages 88–98. Springer, 2012.

[88] O. Kupferman, Y. Lustig, M. Y. Vardi, and M. Yannakakis. Temporal synthesis for bounded systems and environments. In T. Schwentick and C. Dürr, editors, 28th International Symposium on Theoretical Aspects of Computer Science, STACS 2011, March 10-12, 2011, Dortmund, Germany, volume 9 of LIPIcs, pages 615–626. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

[89] O. Kupferman, N. Piterman, and M. Y. Vardi. From liveness to promptness. Formal Methods in System Design, 34(2):83–103, 2009.

[90] O. Kupferman and M. Y. Vardi. Safraless decision procedures. In 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2005), 23-25 October 2005, Pittsburgh, PA, USA, Proceedings, pages 531–542. IEEE Computer Society, 2005.

[91] S. K. Lahiri and C. Wang, editors. Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings, volume 11138 of Lecture Notes in Computer Science. Springer, 2018.

[92] W. Li, L. Dworkin, and S. A. Seshia. Mining assumptions for synthesis. In S. Singh, B. Jobstmann, M. Kishinevsky, and J. Brandt, editors, 9th IEEE/ACM International Conference on Formal Methods and Models for Codesign, MEMOCODE 2011, Cambridge, UK, 11-13 July, 2011, pages 43–50. IEEE, 2011.

[93] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki. Learning rate based branching heuristic for SAT solvers. In N. Creignou and D. L. Berre, editors, Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings, volume 9710 of Lecture Notes in Computer Science, pages 123–140. Springer, 2016.

[94] C. Lignos, V. Raman, C. Finucane, M. P. Marcus, and H. Kress-Gazit. Provably correct reactive control from natural language. Auton. Robots, 38(1):89–105, 2015.

[95] S. Lindley, P. Wadler, and J. Yallop. Idioms are oblivious, arrows are meticulous, monads are promiscuous. Electr. Notes Theor. Comput. Sci., 229(5):97–117, 2011.

[96] H. Liu, E. Cheng, and P. Hudak. Causal commutative arrows. J. Funct. Program., 21(4-5):467–496, 2011.

[97] H. Liu and P. Hudak. Plugging a space leak with an arrow. Electr. Notes Theor. Comput. Sci., 193:29–45, 2007.

[98] C. Löding and S. Winter. Synthesis of deterministic top-down tree transducers from automatic tree relations. In A. Peron and C. Piazza, editors, GandALF 2014, volume 161 of EPTCS, pages 88–101, 2014.

[99] M. Luttenberger, P. J. Meyer, and S. Sickert. Practical synthesis of reactive systems from LTL specifications via parity games. Acta Inf., 57(1):3–36, 2020.

[100] P. Madhusudan. Synthesizing reactive programs. In Bezem [11], pages 428–442.

[101] G. Mainland, editor. Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016. ACM, 2016.

[102] D. A. Martin. Borel determinacy. Annals of Mathematics, 102:363–371, 1975.

[103] C. McBride and R. Paterson. Applicative programming with effects. J. Funct. Program., 18(1):1–13, 2008.

[104] G. H. Mealy. A method for synthesizing sequential circuits. The Bell System Technical Journal, 34(5):1045–1079, Sept 1955.

[105] P. J. Meyer, S. Sickert, and M. Luttenberger. Strix: Explicit reactive synthesis strikes back! In H. Chockler and G. Weissenbacher, editors, Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I, volume 10981 of Lecture Notes in Computer Science, pages 578–586. Springer, 2018.

[106] S. Miyano and T. Hayashi. Alternating finite automata on omega-words. Theor. Comput. Sci., 32:321–330, 1984.

[107] R. Mori and N. Yonezaki. Several realizability concepts in reactive objects. Proceedings of Information Modeling and Knowledge Bases IV: Concepts, Methods and Systems, 1993.

[108] T. E. Murphy. A livecoding semantics for functional reactive programming. In Janin and Sperber [74], pages 48–53.

[109] G. Patai. Efficient and compositional higher-order streams. In J. Mariño, editor, Functional and Constraint Logic Programming - 19th International Workshop, WFLP 2010, Madrid, Spain, January 17, 2010. Revised Selected Papers, volume 6559 of Lecture Notes in Computer Science, pages 137–154. Springer, 2010.

[110] R. Paterson. A new notation for arrows. In B. C. Pierce, editor, Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001, pages 229–240. ACM, 2001.

[111] I. Perez. GALE: a functional graphic adventure library and engine. In Proceedings of the 5th ACM SIGPLAN International Workshop on Functional Art, Music, Modeling, and Design, FARM@ICFP 2018, Oxford, UK, September 9, 2017, pages 28–35, 2017.

[112] I. Perez, M. Bärenz, and H. Nilsson. Functional reactive programming, refactored. In Mainland [101], pages 33–44.

[113] R. Piskac and R. Dimitrova, editors. Proceedings Fifth Workshop on Synthesis, SYNT@CAV 2016, Toronto, Canada, July 17-18, 2016, volume 229 of EPTCS, 2016.

[114] N. Piterman. From nondeterministic büchi and streett automata to deterministic parity automata. Logical Methods in Computer Science, 3(3), 2007.

[115] A. Platzer and J. Quesel. European train control system: A case study in formal verification. In K. K. Breitman and A. Cavalcanti, editors, Formal Methods and Software Engineering, 11th International Conference on Formal Engineering Methods, ICFEM 2009, Rio de Janeiro, Brazil, December 9-12, 2009. Proceedings, volume 5885 of Lecture Notes in Computer Science, pages 246–265. Springer, 2009.

[116] A. Pnueli. The temporal logic of programs. In 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977, pages 46–57. IEEE, IEEE Computer Society, 1977.

[117] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In G. Ausiello, M. Dezani-Ciancaglini, and S. R. D. Rocca, editors, Automata, Languages and Programming, 16th International Colloquium, ICALP89, Stresa, Italy, July 11-15, 1989, Proceedings, volume 372 of Lecture Notes in Computer Science, pages 652–671. Springer, 1989.

[118] E. L. Post. A variant of a recursively unsolvable problem. Bulletin of the American Mathematical Society, 52(4):264–268, 04 1946.

[119] J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings, volume 137 of Lecture Notes in Computer Science, pages 337–351. Springer, 1982.

[120] M. O. Rabin. Decidability of second-order theories and automata on infinite trees. Transactions of the american Mathematical Society, 141:1–35, 1969.

[121] V. Raman and H. Kress-Gazit. Explaining impossible high-level robot behaviors. IEEE Trans. Robotics, 29(1):94–104, 2013.

[122] V. Raman, C. Lignos, C. Finucane, K. C. T. Lee, M. P. Marcus, and H. Kress-Gazit. Sorry dave, i'm afraid I can't do that: Explaining unachievable robot tasks using natural language. In P. Newman, D. Fox, and D. Hsu, editors, Robotics: Science and Systems IX, Technische Universität Berlin, Berlin, Germany, June 24 - June 28, 2013, 2013.

[123] G. S. Rohde. Alternating Automata and the Temporal Logic of Ordinals. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1997.

[124] S. Safra. On the complexity of omega-automata. In 29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988, pages 319–327. IEEE Computer Society, 1988.

[125] A. Sangiovanni-Vincentelli and M. D. Natale. Embedded system design for automotive applications. Computer, 40(10):42–51, 2007.

[126] M. Santolucito, D. Quick, and P. Hudak. Media modules: Intermedia systems in a pure functional paradigm. In Looking Back, Looking Forward: Proceedings of the 41st International Computer Music Conference, ICMC 2015, Denton, TX, USA, September 25 - October 1, 2015. Michigan Publishing, 2015.

[127] K. Sawada and T. Watanabe. Emfrp: a functional reactive programming language for small-scale embedded systems. In L. Fuentes, D. S. Batory, and K. Czarnecki, editors, Companion Proceedings of the 15th International Conference on Modularity, Málaga, Spain, March 14 - 18, 2016, pages 36–44. ACM, 2016.

[128] S. Schewe. Solving parity games in big steps. J. Comput. Syst. Sci., 84:243–262, 2017.

[129] V. Schuppan. Towards a notion of unsatisfiable and unrealizable cores for LTL. Sci. Comput. Program., 77(7-8):908–939, 2012.

[130] V. Schuppan. Enhancing unsatisfiable cores for LTL with information on temporal relevance. Theor. Comput. Sci., 655:155–192, 2016.

[131] V. Schuppan. Extracting unsatisfiable cores for LTL via temporal resolution. Acta Inf., 53(3):247–299, 2016.

[132] D. Shah, E. Hung, C. Wolf, S. Bazanski, D. Gisselquist, and M. Milanovic. Yosys+nextpnr: An open source framework from verilog to bitstream for commercial fpgas. In 27th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2019, San Diego, CA, USA, April 28 - May 1, 2019, pages 1–4. IEEE, 2019.

[133] Z. Shan, T. Azim, and I. Neamtiu. Finding resume and restart errors in android applications. In E. Visser and Y. Smaragdakis, editors, Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016, pages 864–880. ACM, 2016.

[134] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. J. ACM, 32(3):733–749, 1985.

[135] M. K. Srivas and S. P. Miller. Applying formal verification to the AAMP5 microprocessor: A case study in the industrial use of formal methods. Formal Methods in System Design, 8(2):153–188, 1996.

[136] W. Thomas. Facets of synthesis: Revisiting church's problem. In L. de Alfaro, editor, Foundations of Software Science and Computational Structures, 12th International Conference, FOSSACS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings, volume 5504 of Lecture Notes in Computer Science, pages 1–14. Springer, 2009.

[137] W. Thomas and H. Lescow. Logical specifications of infinite computations. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, A Decade of Concurrency, Reflections and Perspectives, REX School/Symposium, Noordwijkerhout, The Netherlands, June 1-4, 1993, Proceedings, volume 803 of Lecture Notes in Computer Science, pages 583–621. Springer, 1993.

[138] J. C. Tiernan. An efficient search algorithm to find the elementary circuits of a graph. Commun. ACM, 13(12):722–726, 1970.

[139] W. Tracz, M. Young, and J. Magee, editors. Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA. ACM, 2002.

[140] B. Trakhtenbrot and I. Barzdin. Finite Automata; Behavior and Synthesis. Fundamental Studies in Computer Science, V. 1. North-Holland Publishing Company; New York: American Elsevier, 1973.

[141] R. Trinkle. Reflex-frp. `https://github.com/reflex-frp/reflex`, 2017.

[142] A. van der Ploeg and K. Claessen. Practical principled FRP: forget the past, change the future, frpnow! In K. Fisher and J. H. Reppy, editors, Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015, pages 302–314. ACM, 2015.

[143] M. Y. Vardi and P. Wolper. Automata theoretic techniques for modal logics of programs (extended abstract). In R. A. DeMillo, editor, Proceedings of the 16th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1984, Washington, DC, USA, pages 446–456. ACM, 1984.

[144] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak. Maple: simplifying SDN programming using algorithmic policies. In D. M. Chiu, J. Wang, P. Barford, and S. Seshan, editors, ACM SIGCOMM 2013 Conference, SIGCOMM'13, Hong Kong, China, August 12-16, 2013, pages 87–98. ACM, 2013.

[145] P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text, volume 925 of Lecture Notes in Computer Science, pages 24–52. Springer, 1995.

[146] H. Weinblatt. A new search algorithm for finding the simple cycles of a finite directed graph. J. ACM, 19(1):43–56, 1972.

[147] D. Winograd-Cort. Effects, Asynchrony, and Choice in Arrowized Functional Reactive Programming. PhD thesis, Yale University, December 2015.

[148] D. Winograd-Cort and P. Hudak. Settable and non-interfering signal functions for FRP: how a first-order switch is more than enough. In J. Jeuring and M. M. T. Chakravarty, editors, Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014, pages 213–225. ACM, 2014.

[149] K. W. Wong, R. Ehlers, and H. Kress-Gazit. Correct high-level robot behavior in environments with unexpected events. In D. Fox, L. E. Kavraki, and H. Kurniawati, editors, Robotics: Science and Systems X, University of California, Berkeley, USA, July 12-16, 2014, 2014.

[150] B. Wymann, E. Espie, and C. Guionneau. Torcs: The open racing car simulator, v1.3.4. `http://torcs.sourceforge.net/index.php`, 2017.

[151] J. Yallop and H. Liu. Causal commutative arrows revisited. In Mainland [101], pages 21–32.

[152] W. Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. Theor. Comput. Sci., 200(1-2):135–183, 1998.

[153] M. Zimmermann. Optimal bounds in parametric LTL games. Theor. Comput. Sci., 493:30–45, 2013.

[154] M. Zimmermann. Unbounded lookahead in WMSO+U games. CoRR, abs/1509.07495, 2015.

[155] M. Zimmermann. Delay games with WMSO+U winning conditions. RAIRO - Theor. Inf. and Applic., 50(2):145–165, 2016.

[156] M. Zimmermann. Finite-state strategies in delay games. In P. Bouyer, A. Orlandini, and P. S. Pietro, editors, Proceedings Eighth International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2017, Roma, Italy, 20-22 September 2017., volume 256 of EPTCS, pages 151–165, 2017.