Saarland University

PhD Thesis

Provably Accurate Verdictors

Foundations and Applications

A dissertation submitted towards the degree
Doktor der Ingenieurwissenschaften
of the Faculty of Mathematics and Computer Science of Saarland University.

Submitted by
Maximilian A. Köhl

Saarbrücken
2024

Dean of the Faculty	Prof. Dr. Roland Speicher
Date of the Colloquium	05 November 2024
Chair of the Committee	Prof. Dr. Sven Apel
Commission	Prof. Dr. Holger Hermanns Prof. Dr. Clemens Dubslaff Prof. Dr. Benjamin Kaminski Prof. Dr. Martin Leucker
Academic Assistant	Dr. Andreas Schmidt

Abstract

This thesis introduces provably accurate verdictors, systems designed to provide provably accurate answers to pressing operational questions about a system by observing its behavior. For instance, verdictors can detect faults in aircraft sensors or incorrect configurations in manufacturing equipment at runtime. Such information is critical for ensuring system safety and availability, enabling informed safeguarding and timely interventions. Adopting a model-based methodology, this thesis leverages formal models as the ground truth for a system's behavior and properties. A comprehensive theoretical framework is established, providing a precise formal characterization for what it means to provide provably accurate answers. Building on this framework, the thesis develops generic algorithms for implementing and synthesizing provably accurate verdictors in both discrete and continuous-time settings, focusing on robustness against observational imperfections such as losses and delays. The versatility and effectiveness of these algorithms is demonstrated across diverse applications, including runtime verification and fault diagnosis. Additionally, the thesis introduces variability-aware monitoring, addressing significant challenges in monitoring configurable systems with configurable verdictors. With Momba, a new state-of-the-art tool for formal modeling is presented. Using Momba, the theoretical contributions of the thesis are implemented, validated, and empirically evaluated.

Zusammenfassung

Diese Dissertation stellt nachweisbar akkurate Verdiktoren vor, Systeme, die durch die Beobachtung eines Systems akkurate Antworten auf betriebskritische Fragen liefern. So können zur Laufzeit Fehler in Flugzeugsensoren oder falsche Konfigurationen von Produktionsanlagen erkannt werden. Solche Informationen sind entscheidend für die Systemsicherheit und -verfügbarkeit, da sie fundierte Schutzmaßnahmen und rechtzeitige Eingriffe ermöglichen. Diese Arbeit verfolgt einen modellbasierten Ansatz, bei dem formale Modelle als Referenz für das Systemverhalten dienen. Es wird ein theoretisches Rahmenwerk entwickelt, das präzise definiert was es bedeutet, nachweisbar akkurate Antworten zu liefern. Darauf aufbauend werden generische Algorithmen zur Implementierung und Synthese von nachweisbar akkuraten Verdiktoren für diskrete und kontinuierliche Zeitmodelle entwickelt, wobei der Schwerpunkt auf Robustheit gegenüber Beobachtungsimperfektionen, z.B., Verzögerungen und Verlusten, liegt. Die Vielseitigkeit und Effektivität dieser Algorithmen wird in verschiedenen Anwendungsbereichen wie Laufzeitverifikation und Fehlerdiagnose demonstriert. Zudem wird variabilitätsgewahres Überwachen eingeführt, um konfigurierbare Systeme mithilfe von konfigurierbaren Verdiktoren zu überwachen. Mit Momba wird ein modernes Werkzeug für formales Modellieren vorgestellt. Die entwickelten Ansätze werden in Momba implementiert und durch empirische Studien evaluiert sowie validiert.

Acknowledgements

This work was partially funded by the Deutsche Forschungsgemeinschaft (DFG) under project number 389792660, TRR 248, CPEC, see https://perspicuous-computing.science, by the VolkswagenStiftung as part of Grant AZ 98514 (EIS), and by the ERC Advanced Investigators Grant 695614 (POWVER).

Usage of Generative AI. This thesis has been written shortly after generative AI gained widespread popularity, especially in the form of large language models for generating text. Following scientific best practices, the guidelines adopted by Nature and Springer,[1] and the position of the Deutsche Forschungsgemeinschaft [Deu23], I hereby acknowledge and document the usage of generative AI.

As a non-native speaker of English, producing idiomatic text can be a challenge. To this end, I utilized OpenAI's ChatGPT,[2] a large language model, to assist in the writing of parts of this thesis. The model helped in crafting more fluent and idiomatic English expressions, enhancing the readability and clarity of the presentation. All substantive research, theory, analyses, conclusions, and opinions presented herein are my own. The use of ChatGPT was limited to linguistic assistance and did not contribute to the intellectual content of this thesis. I thank OpenAI for providing this tool, which allowed me to improve the overall quality of my writing.

Personal Acknowledgements. Completing my PhD thesis was not a solitary journey, and I am deeply grateful to those who supported me along the way.

First and foremost, I would like to express my sincere gratitude to my supervisor, Holger Hermanns, for his guidance, insights, and support throughout my doctoral studies. Holger's emphasis on crafting a compelling narrative in every paper greatly influenced how I write and present my work. His challenging critiques, always driven by belief in my potential, pushed me to constantly improve. I am also especially thankful to Clemens Dubslaff for his invaluable feedback, guidance, and the collaboration on key papers underlying this thesis. I joyfully recall our discussions where we

[1] https://web.archive.org/web/20240505224504/https://www.nature.com/articles/d41586-023-00191-1
[2] https://web.archive.org/web/20240511214551/https://openai.com/chatgpt/

brainstormed and discussed ideas that eventually turned into great papers. Without both of you, this thesis would not have been possible.

In addition, I like to thank the independent reviewers Martin Leucker and Benjamin Kaminski for taking the time and effort to evaluate my thesis.

During my almost six years at the chair, I had the pleasure to work with amazing colleagues. I am grateful to Andreas Schmidt, Lena Becker, Sebastian Biewer, Gabriel Dengler, Felix Freiberger, Nazareno Garagiola, Alexander Graf-Brill, Nils Husung, Michaela Klauck, Nikolai Käfer, Gilles Nies, Robin Ohs, Florian Schießl, Sarah Sterz, Gregory Stock, Hanwei Zhang, and Dominic Zimmer for their camaraderie and support throughout my time at the chair. I enjoyed the time spent with everyone, sharing experiences, challenges, and successes. Some of my colleagues also became close friends. In particular, I would like to thank Florian for enduring my sometimes hour-long monologs reflecting on my work, thesis, and life, and Michaela for the great papers we wrote and the amazing time we spent together at various conferences. I also had many fun times with Felix, who has been my office mate since the beginning, and with Sarah, who always was ready for a philosophical discussion. Furthermore, I would like to thank Lena and Gregory for their relentless "early nom" support. A special thank you also goes to Sabine Nermerich and Christa Schäfer, who always guaranteed a smooth processing of all administrative matters.

Besides my colleagues, I am deeply thankful to Christel Baier, Kevin Baum, Dimitri Bohlender, Bernd Finkbeiner, Timo Gros, Jörg Hoffmann, Sascha Klüppelholz, Markus Langer, Daniel Oster, Yannik Schnitzer, Maximilian Schwenger, Timo Speith, Marcel Steinmetz, Julius Wenzel, and Verena Wolf for the opportunity to collaborate on exciting research and for the work we accomplished together.

Lastly, and most importantly, I want to thank my parents and family for their unwavering support and encouragement throughout my academic journey. Their belief in me has been a constant source of strength and motivation.

This thesis would not have been possible without the support of all of you, and I am truly grateful for everything you have done.

Contents

General Remarks

We herein state general remarks regarding references to websites, the availability of data and software used for this thesis, and the presentation of proofs.

References to Websites. References to websites are provided as *Internet Archive* URLs. The Internet Archive[3] is a non-profit organization providing a digital library of websites at specific points in time. Internet Archive URLs have the form

`https://web.archive.org/web/<time>/<url>`

where `<url>` is the URL of the website and `<time>` is a timestamp of the form:

`<year><month><day><hour><minute><second>`

Thus, these URLs specify specific versions of websites which also have a timestamp attached to them and which can be retrieved as long as the Internet Archive keeps operating. We believe this to be superior for traceability over merely stating access times. The Internet Archive has been founded in 1996 and still provides access to versions of websites going back all the way to the late 1990s.[4] Note that websites are not reindexed if they do not change and timestamps do correspond to the point in time a website has been indexed by the Internet Archive. Thus, this thesis may refer to rather old versions, which were still most recent at the time of writing.

Data Availability Statement. The results presented in this thesis are primarily based on peer-reviewed publications. Wherever applicable, the artifacts associated with these publications, such as datasets, source code, and supplementary materials, are publicly available via Zenodo. Zenodo is a platform operated by Cern and pledges to archive the artifacts for at least 20 years.[5] As a quick reference, we provide an overview over all artifacts and the publications they are associated with:

[3] https://web.archive.org/web/20240513000827/https://web.archive.org/
[4] https://web.archive.org/web/20240513015940/https://archive.org/about/
[5] https://web.archive.org/web/20240513044447/https://about.zenodo.org/policies

(AT1) *ATVA'24 Artifact: Configuration Monitor Synthesis* [KDH24]
https://zenodo.org/doi/10.5281/zenodo.12583621

(AT2) *Artifact: Robust Model-Based Diagnosis of Real-Time Systems* [KH23]
https://zenodo.org/doi/10.5281/zenodo.7896267

(AT3) *(TACAS21 Artifact) Momba: JANI Meets Python* [KKH21]
https://zenodo.org/doi/10.5281/zenodo.4431779

(AT4) *QComp 2023: State Space Exploration Artifact* [And+24]
https://zenodo.org/doi/10.5281/zenodo.10626176

(AT5) *Full Source Code and Documentation of Momba*
https://zenodo.org/doi/10.5281/zenodo.13205840

The artifact (AT5) contains the most recent version of Momba at the time of writing, including the source code of the implementations of the techniques developed in this thesis. The source code also contains further documentation clarifying implementation details, e.g., with respect to data structures. It does not contain the data produced by the experiments presented in the respective individual papers.

Detailed Formal Proofs. To ensure the readability of the main text, detailed formal proofs of the theoretical results established in this thesis are located in Appendix A. The body of the thesis primarily presents proof sketches and complete proofs of results that are sufficiently brief to maintain narrative flow. This arrangement has been chosen to allow the reader to focus on the conceptual and methodological developments within the core chapters, while providing the rigorous mathematical details separately for those interested in the deeper technical aspects.

Mathematical Notations

As a quick reference, we list main notations used throughout this thesis.

Interval Arithmetic

Sequences, Words, and Languages

Formal Models

Formal Models

Observations and Events

Chapter 1

Introduction

In today's highly interconnected and automated world, computers control almost everything from airplanes to manufacturing plants to medical devices. Ensuring the reliability of these systems is paramount as human lives often depend on them. Furthermore, damages or disruptions of service can often incur significant financial losses, for instance, when a manufacturing line becomes inoperable. However, maintaining a consistently safe and functional state amid complex operational demands, potential technical failures, and human errors presents a major challenge.

Addressing this challenge requires not only design-time strategies but also the ability to monitor a system's execution and to ascertain its operational state at runtime. Monitoring a system enables manual or automated *interventions* when issues arise as well as *safeguarding* the system from entering unsafe or non-functional states [e.g. Pin+17b; LBW09; Fal+11; Blo+15; Pin+16; Jan+20]. For instance, in aviation, pilots are constantly provided with diagnostic information that enables them to swiftly take corrective actions when necessary, e.g., when an engine is on fire [SKY]. At the same time, the aircraft is safeguarded from piloting errors, for instance, by preventing in-flight trust reversal. In manufacturing and healthcare, equipment typically needs to be monitored for technical issues and misconfigurations to detect potential problems early, to ensure the efficient operation of equipment, to prevent any damages, and importantly, to protect workers and patients at all times [Zon+20; THE93]. As a result, a medical x-ray machine may shut down automatically if it detects a problem with its radiation shielding or cooling system, and a factory worker may be prevented from starting a production process when the equipment is ill-configured.

In all these cases, techniques applied at runtime are also a crucial cornerstone of design-time considerations and assurance cases. They complement process-based measures and design-time verification approaches, all aimed at ensuring a system's reliable operation. Against this backdrop, this thesis develops techniques for obtaining accurate and decisive information about the current execution and state of a system

at runtime. These techniques are designed to effectively answer questions such as:

(Q1) Does the x-ray machine still operate within its safe limits?

(Q2) Has there been a fault in a vital sensor of the aircraft?

(Q3) Has the manufacturing equipment been correctly configured?

We loosely refer to such questions by the umbrella term *operational questions*. Answering these questions *accurately* enables not only timely interventions and corrective actions but also robust safeguarding mechanisms. Examples may be shutting down medical equipment before its use becomes unsafe, informing pilots about faults, and preventing workers from starting a manufacturing process. Hence, accurate answers to such questions are essential for keeping systems safe and functional. Conversely, *incorrect* answers can have catastrophic consequences: An x-ray machine may expose patients and medical staff to hazardous levels of radiation, misreported faults may lead pilots to taking wrong or no corrective actions, and ill-configured manufacturing equipment may halt production and put workers' lives at risk. Hence, this thesis strives for techniques that provide answers with *provable guarantees*.

A Model-Based Methodology. To provide accurate answers with provable guarantees to operational questions, the techniques developed in this thesis build upon *formal models*. Formal models are mathematical representations of systems and their possible behavior. Formal models precisely describe a system's behavior under various circumstances and its interaction with its environment. As such, they have found widespread and successful use for verifying systems at design time, particularly through various forms of *model checking* [CE81; QS82; CW96; BK08]. Given a formal model that faithfully represents the behavior of a system, model checking systematically verifies whether the system fulfills specific requirements. For instance, model checking can be used to verify that an x-ray machine never overexposes patients or that a traffic light controller always lets traffic pass through an intersection fairly.

While model checking is invaluable for verifying a system's behavior against requirements at design time, implementing the desired behavior often critically requires internal knowledge about a system's state at runtime. For example, an x-ray machine may only avoid overexposing patients because it can detect and appropriately respond to failures of its cooling system. Otherwise, overheating may cause an overdosage if it damages critical circuitry responsible for controlling the dosage. Although model checking can confirm that the machine is designed to react appropriately under such circumstances, model checking does not produce a detector for cooling system failures. Yet, such a detector may be essential to implement an overdosage protection. Any model of the machine verified to not overexpose patients will likely include such a detector together with other control components. So, model checking and design-time verification do not eliminate the need for techniques that provide accurate and decisive information about a system's state at runtime.

The techniques developed in this thesis leverage formal models by combining

the knowledge about a system they represent with *observations* of a running system. Thereby, they allow answering operational questions effectively and accurately. In this setting, formal models serve as a ground truth for system faults, configurations, and properties of interest at runtime. The strength of this *model-based methodology* lies in its ability to deliver provable guarantees on the provided answers. This ensures that interventions and safeguarding mechanisms can be based on well-grounded and accurate information about the present state of the system.

The Big Picture. Having motivated the challenges addressed by this thesis and the approach we are taking to address them on a high-level, we now need to introduce some technical terminology. This terminology will establish a common language to describe and understand the contributions made by this thesis.

First, we refer to answers to the broad range of operational questions stated earlier as *verdicts*. To capture the broad range of operational questions, verdicts can range from simple *yes-no-unknown* statements, indicating the satisfaction or violation of properties (Q1), to elaborate *diagnoses* indicating the presence of faults (Q2), and to sets of *possible configurations* the system be in (Q3). As we will show, by requiring only a few simple structural properties on verdicts, the techniques we develop can be generic across all those questions. This enables us to take a unified approach towards techniques for answering those questions effectively and accurately.

Second, we refer to systems designed to process observations and produce verdicts as *verdictors*. The techniques developed in this thesis aim at the effective implementation and synthesis of verdictors that are *provably accurate*. That is, they are proven to produce accurate verdicts based on a formal system model. As these verdicts constitute answers to practically relevant operational questions, they can then serve as a basis for timely interventions and safeguarding mechanisms. Figure 1.1 illustrates this general architecture upon which this thesis is built.

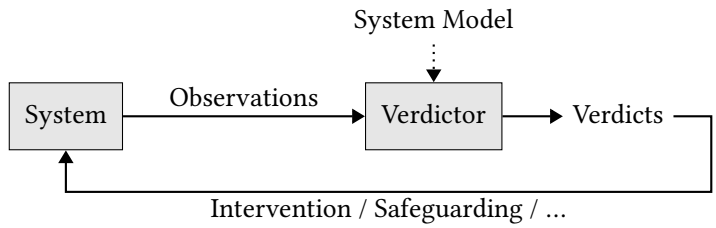


Figure 1.1: The general architecture upon which this thesis is built.

Throughout this thesis, we assume that observations are caused by *events* that occur within a system. While a system may internally be governed by continuous dynamics, we take events and observations to be discrete and instantaneous. For instance, the power drawn by an x-ray tube may change continuously. To observe it, however, it needs to be measured at specific discrete points in time. Measurements

are events that *occur within the system* and can cause observations *to be made by a verdictor*, for instance, the observation that the tube currently draws 2.4 kW.

Observational Imperfections. In real-world scenarios, observations obtained from a system are seldomly perfect. For instance, sensor data may be distorted by noise, observations sent over a network may be delayed or lost, and resource constraints may limit sensor deployment and data logging. We subsume these varied issues under the umbrella term *observational imperfections*, encompassing all effects that adversely affect the acquisition of observations by a verdictor. Observational imperfections pose significant challenges for obtaining accurate verdicts.

For example, lost or delayed observations may induce the phenomenon that the system transitioned into a completely different state than the observations would otherwise indicate. Consequently, if a verdictor does not account for these losses and delays, it risks producing inaccurate verdicts. Relying on such potentially erroneous verdicts for interventions and safeguarding could then have adverse effects on a system's availability and safety, as decisions would be based on potentially flawed information. For instance, erroneously activating the fire suppression system of an aircraft's engine despite there being no fire would significantly degrade the state of the aircraft by rendering the engine inoperable. Likewise, an erroneously activated airbag of a car can easily lead to an accident and an erroneously shut down manufacturing plant quickly incurs high financial losses. Thus, it is paramount to take observational imperfections into account to ensure accurate verdicts.

While dealing with observational imperfections is paramount, addressing all conceivable observational imperfections would be a Herculean effort and definitely infeasible within the scope of this (or any other) thesis. For instance we do not address *signal processing* [Orf95; RM87], which is an entire research field dedicated to dealing with noise and other imperfections in measurements and communication signals, e.g., by deploying sophisticated filtering techniques [WB95; Che+06]. Techniques developed in signal processing and similar areas can be exploited by preprocessing any measurements before they are passed as observations to a verdictor.

Within the scope of this thesis, we aim to account for observational imperfections that can be effectively dealt with by exploiting the information about a system's operational behavior encoded in the system's model. To this end, we focus on a subset of common and realistic observational imperfections. This thesis specifically focuses on *limited observability*, concerning inherent limitations on which events within a system can be observed, and on *delays, losses, and timing imprecisions* induced by networks. These observational imperfections are common and typically unavoidable in practice, in particular, as systems are increasingly more distributed, usually requiring that observations are transmitted over shared networks. For instance, different machines on a manufacturing floor are typically connected to each other via shared networks such as *Real-Time Ethernet* [Fel05], *Profibus* [TV99], or

CAN [Di +12; THW94]. By taking into account these imperfections, we enhance the practical relevance and robustness of the techniques we develop, ensuring they perform effectively under real-world conditions.

Observational imperfections are often aggravated by the fact that additional *active correction techniques* cannot be used as these would affect the system's behavior, especially in the case of real-time systems [Yan96; TFC90; Tsa+90]. Changing a system's behavior may invalidate the model on which verdicts are based, thereby rendering any guarantees void, or even have direct adverse effects on the safety and functioning of the system itself. For instance, in case of a shared network connecting manufacturing equipment, requiring lost observations to be sent again may induce additional load, change the timing of events, or cause overload situations for the network or equipment, thereby posing a potential hazard for the safety and functioning of the manufacturing process. For these reasons, we focus on completely passive techniques that cannot affect a system's behavior in any way.

Prior Work. Of course, the idea to use observations to obtain information about a system's state at runtime is not new at all. In particular, the research areas of *runtime verification* [HG05; LS09] and model-based *fault diagnosis* [Sam+95] are concerned with techniques aimed at detecting the satisfaction or violation of properties (Q1) and at detecting faults (Q2), respectively. The techniques we present in this thesis are inspired by, build upon, and expand the results established in those areas.

In the area of runtime verification, a plethora of different approaches have been proposed and studied. Prominent approaches roughly fit into two categories: *Logic-* and *automata-based* [e.g. BLS06b; FFM12; CPS08; Bar+04; MB06; AFR16] and *stream-based* [e.g. DAn+05; Fay+19; Con+18; GS18]. Logic- and automata-based approaches typically construct an automaton that yields verdicts when fed with a sequence of observations. Stream-based approaches are based on algorithms that incrementally consume multiple sequences of observations, coined *input streams*, and produce multiple sequences of values, coined *output streams*. In both cases, these techniques aim at the implementation or construction of a *runtime monitor*, a system that processes observations of some sort and typically produces a *truth verdict* indicating the satisfaction or violation of some property [HG05; LS09]. Conceptually, this fits the general architecture depicted in Figure 1.1, where a runtime monitor then is a special form of verdictor for answering operational questions about property satisfaction. However, runtime monitors are typically not constructed based on a formal model of a system but implemented or constructed for a given *specification*, i.e., a formal characterization of the property to be monitored. Although, some works also aim to exploit knowledge about a concrete system to address observational imperfections or for predictions [e.g. ZLD12; Pin+17a; Fer+21; CTT19].

Model-based fault diagnosis has been pioneered by Sampath et al. [Sam+95]. Here, the idea is to implement or construct a *diagnoser* based on a formal model of

the system that also models the effect of different faults on the system's behavior. Exploiting this knowledge about a system's behavior in the presence of faults, it then becomes possible to *diagnose faults* within the system based on observations of its behavior. Similarly to runtime verification, conceptually this fits the general architecture in Figure 1.1, where a diagnoser then is a special form of verdictor for answering operational questions about the occurrence of faults. Since its inception, several works have build upon this idea, also addressing observational imperfections to some extent [e.g. Car+13; Mha+17; Tri02; BCD05; ALH06; TYG08].

The striking similarities between runtime verification and fault diagnosis have been recognized and discussed among researches from the different fields [Hav+10]. Yet, we are not aware of any attempts to develop a general theoretical foundation unifying runtime verification and fault diagnosis into a coherent framework, e.g., such that results and algorithms can be shared and made usable for both. This thesis contributes such a foundation, or at least a versatile starting point for it, by approaching the problem of answering operational questions in a more generic way than existing work. Among other things, this allows us to address observational imperfections and predictions in a way that is usable for both runtime verification and fault diagnosis, and that can be combined with existing techniques.

Configurable Systems. Neither runtime verification nor fault diagnosis specifically addresses the question which configuration a system may have (Q3), nor do the established techniques take configurability of a system into account. Yet, almost all modern systems are instances of highly configurable designs. The usually huge amount of possible configurations of such systems do not only pose several significant challenges for their design and analysis [CE00; Ape+13] but naturally also raise further challenges for analyzing their behavior at runtime. In particular, a verdictor may need to be adapted to the configuration of a system in order to produce accurate verdicts. For instance, the properties of interest or the behavior indicative of a fault may depend on the configuration of the system.

While it may be necessary to adapt a verdictor to the configuration of a system, often a system's configuration is *not* readily exposed at runtime [AFW18], and thus is initially unknown. Recall our earlier example, where a factory worker configures a machine prior to a production step. This may involve setting up physical components whose configuration simply cannot be queried by software. Moreover, finding out an a-priori unknown configuration of a system is an operational question in its own right, e.g., to be able to check that it is correct (Q3).

Addressing these challenges, this thesis introduces *variability-aware monitoring*. It contributes tailored techniques, enabling verdictors to be adapted to the configuration of a system and to ascertain a system's configuration solely by observing its behavior. We refer to the latter as *configuration monitoring*. These techniques open up entirely novel applications not covered by any prior work.

1.1 Contributions

From the preceding discussions it should be clear that this thesis makes cross-cutting contributions across the research areas of runtime verification, fault diagnosis, and configurable systems. We aim to develop effective techniques for obtaining provably accurate information about a system's operational state and its execution at runtime. This information may range from whether a property is satisfied or violated, to the occurrences of faults and the configurations a system may have.

Concretely, the main contributions of this thesis are:

(1) A model-based *theoretical framework* that provides a precise formal characterization for what it means to produce accurate verdicts,
(2) *generic verdictor algorithms* for implementing and synthesizing provably accurate verdictors based on formal system models,
(3) a general-purpose and state-of-the-art *formal modeling toolbox* in which the developed algorithms have been implemented, and
(4) an exploration of concrete *applications* complemented by *empirical evaluations* of the developed techniques demonstrating their efficiency.

Figure 1.2 provides a high-level overview of the contributions of this thesis, how they are reflected in the thesis' structure, and key references to the papers on which they are primarily based. The theoretical framework and formal modeling toolbox are foundational contributions that enable the generic verdictor algorithms and their concrete applications and evaluation. We now discuss the individual contributions and their relations to each other in more detail.

	Theory	**Practice**
Foundations	(1) Contribution FT Theoretical Framework Chapter 3 [KDH24]	(3) Contribution FP Formal Modeling Toolbox Chapter 6 [KKH21]
Techniques	(2) Contribution TT Generic Algorithms Chapter 4, Chapter 5 [KDH24; KH23]	(4) Contribution TP Applications and Evaluation Chapter 7, Chapter 8 [KDH24; KH23; DK22]

Figure 1.2: High-level overview of the contributions and structure of this thesis, including references to the papers on which they are primarily based.

1.1.1 Theoretical Framework

Contribution FT. To provide rigorous criteria for what it means to produce accurate verdicts about a given system, we introduce a theoretical framework formalizing the key concepts behind Figure 1.1. We show that this framework also provides a unifying foundation for existing work in the spectrum of runtime verification and model-based fault diagnosis.

The model-based approach pursued in this thesis aims to deliver provable guarantees on verdicts. For safety- and mission-critical applications, it is vital that verdicts accurately reflect the actual state of the system. This ensures that any critical decisions based on them are well-grounded. To prove that the techniques we develop indeed deliver such guarantees, we first need to establish a theory that provides criteria for what it means to produce accurate verdicts about a given system.

As central ingredients of the theoretical framework, we introduce *verdict transition systems* (VTSs), *observation models*, and *verdict oracles*. VTSs are formal representations of verdictors, capturing how verdicts evolve and are refined over time as new observations are made. Observation models describe which observations may be generated by the executions of a system, and verdict oracles ascribe verdicts to executions of the system. The verdict ascribed to an execution by a verdict oracle serves as the ground truth for the respective execution of the system. Notably, observation models are sufficiently general to accommodate several common and realistic observational imperfections. By combining these central ingredients, we then obtain rigorous criteria for what it means to produce accurate verdicts about a given system and based on observations subject to observational imperfections. These criteria will later serve as proof obligations for the algorithms we develop.

VTSs are very general allowing them to also serve as a unifying foundation for existing work in the spectrum of automata-based runtime verification and model-based fault diagnosis. To substantiate this claim, we demonstrate that paradigmatic instances of existing work indeed naturally fit within the theoretical framework. Recall that while it has been recognized that runtime verification and diagnosis share similarities [Hav+10], we are not aware of any attempts to unify and generalize them into a coherent formal foundation, such that results and algorithms can be shared and made useable for both. As we will show, our theoretical framework can provide such a foundation, thereby enabling generic algorithms.

1.1.2 Generic Verdictor Algorithms

Contribution TT. We develop generic algorithms for implementing and synthesizing provably accurate verdictors. These algorithms are grounded in *verdict-annotated* system models and account for observational imperfections. In the discrete-time setting, the algorithms account for limited observability and network-induced delays, losses, and out-of-order observations. In the continuous-time setting, they account for limited observability, network-induced delays and out-of-order observations, as well as timing imprecisions concerning the specific point in time an observation is made.

The algorithms we develop are inherently generic and versatile, making them applicable to a wide range of operational questions. In particular, they are suitable to answer the broad range of operational questions introduced earlier concerning properties (Q1), the presence of faults (Q2), and possible system configurations (Q3). To this end, they take system models with *verdict annotations* as a basis. Verdict annotations can be used to encode information about properties, faults, and configurations into a system model. Furthermore, they serve as the ground truth for verdict oracles as per the theoretical framework. We will discuss several concrete applications that instantiate the generic algorithms as part of Contribution TP. By adopting a generic approach, we are able to establish general correctness theorems about our algorithms, proving that the resulting verdictors indeed provide accurate verdicts. The algorithms we develop cover both discrete- and continuous-time settings.

Modular Discrete-Time Verdictor Synthesis. The discrete-time setting is characterized by observations that do not carry an explicit time component, i.e., observations are assumed to be made by the verdictor in some order, however, not at a specific point in time. For the discrete-time case, we develop modular building blocks for the explicit synthesis of verdictors. To this end, we will be using VTSs as an intermediate and target representation. Concretely, we develop building blocks for constructing VTSs from verdict-annotated system models and for transforming VTSs to account for limited observability and network-induced delays, losses, and out-of-order observations. In addition, we present transformations for VTSs to produce accurate predictions, by taking the system's future behavior into account, and to optimize them for efficient implementations in hardware and software. These modular building blocks can then be combined to meet the specific requirements of an application, e.g., in terms of observational imperfections. Furthermore, they can also be applied to VTSs obtained with third-party techniques from runtime verification and fault diagnosis, thereby enhancing their practicability.

Continous Time Verdictor Algorithm. In the continuous-time setting, the timing of events—and thus of observations—becomes a crucial factor for answering operational questions. For instance, the time difference between two observations may indicate a fault within the system. While the timing is crucial, its accurate assessment is often hindered by timing imprecisions in practice. In particular, networks may induce variable delays and reorder observations, and hardware clocks are limited in precision, causing clocks to drift apart. To ensure accuracy, verdictors must account for these imprecisions, which is especially challenging as they open up an infinite continuum of possible observation times. Tackling this challenge, we develop a generic verdictor algorithm that takes into account the timing of observations and compensates for clock drift and offsets, varying delays, and delay-induced reordering. As in the discrete-time case, the algorithm rests on a verdict-annotated system model. Based on an observation model capturing the timing imprecisions, we prove that the algorithm indeed provides accurate verdicts as per the theoretical framework.

1.1.3 Formal Modeling Toolbox

Contribution FP. We present *Momba*, a toolbox for formal models. Momba provides APIs for the programmatic constructions of models, for state space exploration, and for conducting model analyses. We evaluate Momba's state space exploration engine empirically on community benchmarks. The techniques presented in this thesis have been implemented as part of Momba and, in particular, utilize Momba's state space exploration engine.

At the core of the model-based approach this thesis adopts are formal models. Dealing with formal models encompasses a variety of tasks some of which can be challenging from time to time—especially for newcomers. Momba is a flexible Python framework and strives to deliver an integrated and intuitive experience to aid the process of model construction, analysis, and validation. It provides convenience functions for the modular and programmatic construction of models. Building upon the JANI format [Bud+17], a community standard for exchanging formal models between tools, Momba is compatible with the existing JANI ecosystem and several state-of-the-art model analysis tools are readily available via Momba's APIs. Most important for the purposes of this thesis is Momba's state space exploration engine, which is used for implementing some of the algorithms developed in this thesis. Furthermore, this engine allows empirically validating a model, for instance, by rapidly prototyping a tool for interactive model exploration and visualization, or by connecting it to a testing framework. It is crucial that a model faithfully represents the real system as otherwise results and insights obtained from it may not be grounded in reality. Models build with Momba can then be used as a basis for verdictors.

As the techniques developed in this thesis, in part, rely on Momba's state space exploration engine, a significant amount of engineering effort has been spend to make it competitive with the state of the art. To this end, Momba's state space exploration engine has been written in Rust, a relatively new programming language offering best-in-class performance on par with low-level languages like C. To show that Momba's state space exploration engine can indeed compete with the state of the art, we empirically evaluate it and compare it to well-established tools.

Building upon Momba's extensible core, we implemented the algorithms developed in this thesis. This implementation will serve as the basis for the empirical evaluation of the contributed techniques as part of Contribution TP.

1.1.4 Applications and Evaluation

Contribution TP. We present several concrete and novel applications based on the generic algorithms developed as part of Contribution TT. They cover all three research areas: runtime verification, fault diagnosis, and configurable systems. Beyond instantiating the generic algorithms, we further extend them and develop complementary techniques for the configurable systems setting. Moreover, we empirically evaluate and demonstrate the efficacy of the developed techniques on benchmarks for selected applications.

The generic algorithms (Contribution TT) and the formal modeling toolbox Momba (Contribution FP) are the basis of this fourth contribution. This contribution is twofold, consisting in runtime verification and fault diagnosis techniques on the one hand and the entirely new area of *variability-aware monitoring* on the other.

Runtime Verification and Fault Diagnosis. While traditional runtime verification techniques do typically not take a system's model into account, the model-based approach adopted in this thesis allows several new contributions to this area. In particular, our techniques allow us to synthesize monitors that are robust with respect to certain observational imperfections or that predict the violation or satisfaction of properties in the future. Beyond these contributions, our model-based approach enables runtime verification for *computation tree logic* (CTL) [CE81]. CTL is a well-established logic for expressing temporal properties over possible future behaviors. As the name suggests, CTL formulas express properties of computation trees, covering all possible system behaviors. The crux of enabling CTL runtime verification lies in the usage of a formal system model to obtain such trees. For instance, one may ask whether there exists an execution path from which all future paths lead to an unsafe state—clearly, this path must then be avoided.

With regard to fault diagnosis, our algorithms extend the state of the art by enabling diagnosers to be robust against observational imperfections, to produce predictions about future faults, and to consider transient faults from which the system may recover. Beyond these contributions, our generic algorithms furthermore enable a novel diagnosis paradigm based on *modal logic* [BRV01]. Modal logic allows the expression of necessities and possibilities concerning faults and their combinations, extending the flexibility of questions regarding faults. While diagnosis in the continuous-time setting has been addressed in the literature before [Mha+17; Tri02; BCD05], robustness against variable delays and timing imprecisions, as enabled by our generic continuous time verdictor algorithm, represents a significant step forward. We evaluate the continuous time verdictor algorithm for this purpose and also empirically study the effect of various observational imperfections on the quality of obtained diagnoses in this setting.

Variability-Aware Monitoring. Recall that existing work on runtime verification and fault diagnosis barely addresses the variability found in almost all modern systems. To produce accurate verdicts, however, verdictors need to be aware of the possible configurations a system may have and, crucially, also be able to adapt to them. Towards such *variability-aware monitoring*, we consider configurability of verdictors themselves as well as of the system that is being observed.

We extend our theoretical framework with a configurable variant of VTSs, representing verdictors that can themselves be configured, e.g., to match the configuration of a system. Configurable VTSs also enable a compositional approach to verdictor synthesis. We demonstrate this approach on *featured linear temporal logic* [Cla+13] (FLTL). FLTL enables the expression of temporal properties that depend on a system's configuration. For instance, an FLTL formula may require that a certain component is eventually initialized, only if the system actually has such a component. Our approach enables the synthesis of runtime monitors for sets of FLTL formulas.

We also show how stream-based approaches can be made to account for system configurations. To this end, we introduce a configurable variant of the stream-based specification language Lola [DAn+05], together with efficient algorithms to analyze all possible configurations of a configurable Lola specification for *well-formedness* and *efficient monitorability*. Well-formedness guarantees that a monitor can indeed be synthesized and efficient monitorability guarantees that a monitor will only consume a statically-known and bounded amount of memory at runtime.

To adapt a configurable verdictor to a system's configuration, this configuration must be known. To determine an initially unknown configuration of a running system we introduce *configuration monitoring*. A *configuration monitor* determines the configuration of a system solely by observing its behavior. We show that configuration monitors can be effectively synthesized from *featured transition systems* (FTSs) [Cla+13], a well-established formal model for configurable systems, by instan-

tiating our generic verdictor synthesis techniques and deriving verdict annotations from an FTS model. Using benchmarks from the configurable systems community, we then evaluate the generic synthesis techniques for this purpose and, again, study the effect of limited observability. These experiments also reveal interesting insights as to why model checking techniques for configurable systems are effective.

Configuration monitors enable us to check a system's configuration at runtime (Q3), even if the system does not readily expose its configuration or we do not trust the configuration it exposes. When combined with configurable verdictors, they enable a powerful self-adaptive paradigm where a verdictor is configured to match a configuration determined by configuration monitoring, as we will discuss.

1.2 Relevant Publications and Artifacts

The contributions are primarily based on the following peer-reviewed work:

> [KDH24]: Maximilian A. Köhl, Clemens Dubslaff, and Holger Hermanns. "Configuration Monitor Synthesis". In: *Automated Technology for Verification and Analysis, ATVA 2024.*
>
> [KH23]: Maximilian A. Köhl and Holger Hermanns. "Model-Based Diagnosis of Real-Time Systems: Robustness Against Varying Latency, Clock Drift, and Out-of-Order Observations". In: *ACM Transactions on Embedded Computing Systems, TECS 2023.*
>
> [DK22]: Clemens Dubslaff and Maximilian A. Köhl. "Configurable-by-Construction Runtime Monitoring". In: *Leveraging Applications of Formal Methods, Verification and Validation, ISoLA 2022.*
>
> [KKH21]: Maximilian A. Köhl, Michaela Klauck, and Holger Hermanns. "Momba: JANI Meets Python". In: *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2021.*

The relation to the individual contributions is indicated in Figure 1.2. The evaluation of Momba's state space exploration engine has been conducted and published as part of the *Quantitative Verification Competition* (QComp) 2023 [And+24].

In addition to the peer-reviewed work, this thesis contains novel unpublished results. In particular, the overarching theoretical framework which binds together all the individual works has not been published before. Furthermore, some of the more specific results presented in the papers have been generalized for this thesis.

Revelant Artifacts. All publications listed above have their own artifacts enabling the reproduction of the empirical results they and this thesis present.

(AT1) *ATVA'24 Artifact: Configuration Monitor Synthesis* [KDH24]
https://zenodo.org/doi/10.5281/zenodo.12583621

(AT2) *Artifact: Robust Model-Based Diagnosis of Real-Time Systems* [KH23]
https://zenodo.org/doi/10.5281/zenodo.7896267

(AT3) *(TACAS21 Artifact) Momba: JANI Meets Python* [KKH21]
https://zenodo.org/doi/10.5281/zenodo.4431779

(AT4) *QComp 2023: State Space Exploration Artifact* [And+24]
https://zenodo.org/doi/10.5281/zenodo.10626176

The artifact (AT3) has undergone artifact evaluation as part of TACAS 2021 and has been granted all available badges: *complete*, *well-documented*, *easy to reuse*, and *consistent*. The artifact (AT1) has undergone artifact evaluation as part of ATVA 2024 and has been granted all available badges: *available*, *functional*, and *reusable*.

The empirical results presented in this thesis are taken from the original publications and can be reproduced with the respective artifacts. In addition, we provide a fifth artifact (AT5) including the most-recent version of Momba at the time of writing. Notably, this artifact includes the source code of the implementations of all the techniques developed in this thesis. The source code also contains further documentation clarifying implementation details, e.g., with respect to data structures:

(AT5) *Full Source Code and Documentation of Momba*
https://zenodo.org/doi/10.5281/zenodo.13205840

We consider (AT5) to be an explicit part of the contributions of this thesis. It represents the significant engineering effort which has been spent on Momba and the implementation of the techniques contributed by this thesis.

Other Work. Extending beyond the primary focus of this thesis, the author was also involved in the following noteworthy scientific works. While these works are not the primary focus of this thesis, they clearly influenced the overall research direction and motivation behind this thesis.

- Together with his coauthors, the author developed an operationalization of the *explainability* concept [Köh+19] for the purposes of requirements engineering. This ties in with earlier work on trustworthy AI the author was involved in [BKS17]. Diagnosis and monitoring techniques, which are a focus of this thesis, are motivated by the need to understand the runtime behavior of a system and ultimately are meant to increase the trustworthiness of the system.

- The author pioneered the usage of the stream-based specification language Lola for exhaust emission monitoring [KHB18]. This work has been the basis

for later work where the author was involved [Her+18; Bie+21; Bie+23]. These works are the primary motivation behind configurable verdictors.

- The author was involved in the development of a high-performance, safe, and modular decision diagram (DD) framework in Rust [Hus+24]. DDs are a core data structure used by some of the techniques presented in this thesis. This work was honored with the EASST best paper award at ETAPS 2024.

- Momba has been used in two other works of the author [Gro+22; Faq+20], where it has been shown to be effective and extensible.

The work on explainability and runtime verification can be seen as a starting point from which the distinguishing contributions of this thesis emerged.

1.3 Outline

This thesis is organized into three parts. In Part I, we introduce the foundational concepts and the theoretical framework (Contribution FT) that underpins this thesis, providing a solid basis for exploring the synthesis, implementation, and application of verdictors. In Part II, we develop the generic verdictor algorithms for discrete- and continuous-time settings (Contribution TT) that power the applications of this thesis. In Part III, we then put all the theory into practice by presenting Momba (Contribution FP) and by considering concrete applications (Contribution TP). The structure overall follows the contributions, as indicated in Figure 1.2.

Part I: Theoretical Foundation. The first part establishes the necessary foundations for the thesis. In Chapter 2, we introduce the basic mathematical concepts and notations that are pivotal for understanding the formalisms used throughout this thesis, and recapitulate other necessary background. After establishing all the basics, we present the novel theoretical framework in Chapter 3.

Part II: Generic Verdictor Techniques. The second part presents the generic verdictor algorithms. In Chapter 4, we introduce the modular building blocks for the flexible synthesis of verdictors in the discrete-time setting. In Chapter 5, we address the challenges posed by timed observations subject to delays and timing imprecisions by introducing the continuous time verdictor algorithm.

Part III: From Theory to Practice. This final part explores practical applications and empirically validates the developed techniques. In Chapter 6, we present Momba. In Chapter 7, we explore applications in the area of runtime verification and fault diagnosis and evaluate the continuous time verdictor algorithm in the fault

diagnosis setting. In Chapter 8, we introduce variability-aware monitoring covering configurable verdictors and configuration monitoring, and evaluate the discrete-time verdictor synthesis approach on configuration monitoring.

Part I

Theoretical Foundation

Chapter 2

Foundations

This thesis is rooted in *formal methods*, a research area concerned with mathematically rigorous approaches for ensuring the reliable operation of systems [CW96]. As such, the contributions of this thesis are built upon a firm mathematical foundation. In this chapter, we introduce the basic mathematical concepts and notations used throughout this thesis (Section 2.1). Furthermore, we recapitulate the necessary background on *formal models* (Section 2.2), *temporal and modal logics* (Section 2.3), modeling of *configurable systems* (Section 2.4), *runtime verification* (Section 2.5), *model-based fault diagnosis* (Section 2.6), and *fault trees* (Section 2.7), as far as they are relevant for the contributions presented in this thesis. Readers already familiar with those concepts and topics may skip this chapter and return to it on-demand.

2.1 Mathematical Basics

We assume familiarity with elementary higher-order logic, set theory, and arithmetic. Readers are referred to introductory text books on these subject matters.

For a set X, we denote the power set of X by $\wp(X)$, i.e., $\wp(X)$ is the set of all subsets of X. In case X is finite, we denote its size by $|X|$. For a singleton set $X = \{x\}$, we use the notation ∇X to denote the single element x of X. We use the usual notation $\cup$, $\cap$, $\setminus$, and $\times$ for *union*, *intersection*, *difference*, and the *Cartesian product*, respectively. If sets are disjoint, we also denote their union by $\uplus$. We use the notation $\{\, x \in X \mid \Phi(x) \,\}$ to denote the set of elements of X that do satisfy some constraint $\Phi(x)$. In case X is clear from the context, we may also omit it and simply write $\{\, x \mid \Phi(x) \,\}$. In addition, we use the notation $\{\, g(x) \mid \Phi(x) \,\}$ to denote the set whose elements are generated by the expression $g(x)$ over those x that satisfy the constraint $\Phi(x)$. We use angle brackets for tuples, e.g., $\langle x_1, x_2, x_3 \rangle$ is a tuple with three components x_1, x_2, and x_3. We use a centered dot $\cdot$ to denote that a component of a tuple is irrelevant in the context of a

constraint, quantification, or other binding context. For instance, $\{ x \mid \langle x, \cdot \rangle \in Y \}$ is the set of all first components of the tuples found in Y and $\forall \langle x, \cdot \rangle \in Y$ quantifies over all tuples in Y while binding their first component to x and ignoring the second component. This allows us to reduce unnecessary notational overhead by avoiding bindings of values to variables that do ultimately not matter.

As usual, we denote the set of *natural numbers*, of *integers*, of *rationals*, and of *reals* by $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{R}$, respectively. We take the natural numbers to include the number zero. We denote the set of positive reals and rationals, including zero, by $\mathbb{R}_0^+$ and $\mathbb{Q}_0^+$, respectively. We denote the set of natural numbers without zero by $\mathbb{N}^+$. We use $<$ and $\leq$ to denote the usual strict and non-strict order on numbers.

We use the notation $\bigcirc_{i=1}^{n} x_i$ to apply an operator $\bigcirc$ over an n-tuple $\langle x_1, \ldots, x_n \rangle$ of arguments. Likewise, we use $\{X_i\}_{i=1}^{n}$ to denote a family of n sets.

Relations and Functions. An n-ary relation R over a family $\{X_i\}_{i=1}^{n}$ of sets for $n > 0$ is a subset of the Cartesian product $\times_{i=1}^{n} X_i$, i.e., $R \subseteq \times_{i=1}^{n} X_i$. For $n = 2$ and $n = 3$, we call relations *binary* and *ternary*, respectively. For a binary relation $R \subseteq X \times Y$ over sets X and Y, we write $x \mathrel{R} y$ to denote that $\langle x, y \rangle \in R$, and $x \not\mathrel{R} y$ to denote that $\langle x, y \rangle \notin R$. A binary relation $R \subseteq X \times Y$ is

- *left-total* iff $\forall x \in X : \exists y \in Y : x \mathrel{R} y$,
- *surjective* iff $\forall y \in Y : \exists x \in X : x \mathrel{R} y$,
- *functional* iff $\forall x \in X : \forall y, y' \in Y : x \mathrel{R} y \wedge x \mathrel{R} y' \implies y = y'$, and
- *injective* iff $\forall y \in Y : \forall x, x' \in X : x \mathrel{R} y \wedge x' \mathrel{R} y \implies x = x'$.

A *partial function* $f \subseteq X \times Y$ from a set X to a set Y is a functional binary relation over X and Y. We also write $f : X \rightharpoonup Y$ to indicate that f is a partial function from X to Y and denote the set of all partial functions from X to Y by $X \rightharpoonup Y$. The *domain* $\mathrm{Dom}(f) \subseteq X$ of a partial function $f : X \rightharpoonup Y$ is the set of elements $x \in X$ such that $x \mathrel{f} y$ for some $y \in Y$, i.e., $\mathrm{Dom}(f) := \{ x \mid \exists y \in Y : x \mathrel{f} y \}$. For $x \in \mathrm{Dom}(f)$, we denote the unique element of Y assigned to x by $f(x)$. In case f is injective, its *inverse* $f^{-1} : Y \rightharpoonup X$ is defined by $f^{-1} := \{ \langle y, x \rangle \mid x \mathrel{f} y \}$. A *function* $f \subseteq X \times Y$ from set X to set Y is a partial function with $\mathrm{Dom}(f) = X$, i.e., a functional and left-total binary relation. We also write $f : X \to Y$ to indicate that f is a function from X to Y and denote the set of all functions from X to Y by $X \to Y$. A function f is *bijective* iff it is surjective and injective. In this case, we also call f a *bijection*. The inverse of a bijection is itself a bijection. For a partial function $f : X \rightharpoonup Y$ and a subset $\mathcal{X} \subseteq X$, we define the *$\mathcal{X}$-projection* $f|_{\mathcal{X}}$ of f as $\{ \langle x, y \rangle \in f \mid x \in \mathcal{X} \}$. Note that the $\mathcal{X}$-projection is itself a function from $\mathcal{X}$ to Y, i.e., $f|_{\mathcal{X}} \in \mathcal{X} \to Y$.

For a set X, a binary relation $R \subseteq X \times X$ is

- *reflexive* iff $\forall x \in X : x \mathrel{R} x$,
- *irreflexive* iff $\forall x \in X : x \not\mathrel{R} x$,
- *transitive* iff $\forall x, x', x'' \in X : x \mathrel{R} x' \wedge x' \mathrel{R} x'' \implies x \mathrel{R} x''$,

- *symmetric* iff $\forall x, x' \in X : x\, R\, x' \implies x'\, R\, x$,
- *antisymmetric* iff $\forall x, x' \in X : x\, R\, x' \wedge x'\, R\, x \implies x = x'$, and
- *asymmetric* iff it is antisymmetric and irreflexive.

A binary relation $R \subseteq X \times X$ is an *equivalence relation* iff it is reflexive, transitive, and symmetric. For a set X, let $\mathsf{Id}_X := \{\langle x, x\rangle \mid x \in X\}$ be the identity function. Note that Id_X is an equivalence relation. For a binary relation $R \subseteq X \times X$, the *reflexive closure* of R is defined as $R \cup \mathsf{Id}_X$ and the *irreflexive kernel* of R is defined as $R \setminus \mathsf{Id}_X$.

Orders and Lattices. A *partial order* $\preceq \subseteq X \times X$ over a set X is a reflexive, transitive, and antisymmetric binary relation. A *total order* $\preceq \subseteq X \times X$ is a partial order where all elements are comparable, i.e., $x \preceq x'$ or $x' \preceq x$ for all $x, x' \in X$. A (partially) ordered set is a tuple $\langle X, \preceq\rangle$ where $\preceq$ is a (partial) order over the set X. The *dual* $\preceq^{\mathrm{d}}$ of a partial order $\preceq$ inverts the order relationship, i.e., $x \preceq^{\mathrm{d}} x'$ iff $x' \preceq x$. The dual of a (partially) ordered set $\langle X, \preceq\rangle$ is the (partially) ordered set $\langle X, \preceq^{\mathrm{d}}\rangle$.

For a partially ordered set $\langle X, \preceq\rangle$, a *lower bound* of a subset $\mathcal{X} \subseteq X$ is an element $b \in X$ such that $b \preceq x$ for all $x \in \mathcal{X}$. A lower bound b of $\mathcal{X}$ is called *infimum*, or *greatest lower bound*, iff $b' \preceq b$ for all lower bounds b' of $\mathcal{X}$. An *upper bound* of a subset $\mathcal{X} \subseteq X$ is an element $b \in X$ such that $x \preceq b$ for all $x \in \mathcal{X}$. An upper bound b of $\mathcal{X}$ is called *supremum*, or *least upper bound*, iff $b \preceq b'$ for all upper bounds b' of $\mathcal{X}$. Infimum and supremum are not guaranteed to exist, but, if they exist, they are uniquely determined. In case they exist, we denote the infimum by $\inf \mathcal{X}$ and the supremum by $\sup \mathcal{X}$, respectively. In case $(\inf \mathcal{X}) \in \mathcal{X}$, we call $\inf \mathcal{X}$ the *minimum* of $\mathcal{X}$, and also denote it by $\min \mathcal{X}$. Analogously, in case $(\sup \mathcal{X}) \in \mathcal{X}$, we call $\sup \mathcal{X}$ the *maximum* of $\mathcal{X}$, and also denoted it by $\max \mathcal{X}$.

A partially ordered set $\langle X, \preceq\rangle$ is a *meet-semilattice* iff every two element subset $\{x, x'\} \in X$ has a greatest lower bound, coined *meet* and denoted by $x \sqcap x'$. Analogously, a *join-semilattice* is a partially ordered set where every two element subset $\{x, x'\} \in X$ has a least upper bound, coined *join* and denoted by $x \sqcup x'$. A meet- or join-semilattice is *complete* iff every non-empty subset $\mathcal{X} \subseteq X$ has a greatest lower bound (meet) or least upper bound (join), respectively. We use the notation $\bigsqcup \mathcal{X}$ to denote the join of the non-empty set $\mathcal{X} \subseteq X$ and $\bigsqcap \mathcal{X}$ to denote the meet of the non-empty set $\mathcal{X} \subseteq X$. The dual of a meet-semilattice is a join-semilattice and vice versa. Complete meet-semilattices, as we defined them, are also known as *Scott domains*, originating in the work of Scott on domain theory [Sco82]. Hence, complete join-semilattices, as we defined them, are duals of Scott domains. We write $\mathsf{LOpCost}(k)$ (lattice operation cost) for the worst-case time complexity of computing the join/meet of k elements. As a graphical representation of semilattices, we use *Hasse diagrams* [Bir40] where the order $\preceq$ is represented by arrows. See Figure 3.1 for examples. A meet- or join-semilattice is *bounded* if X has a minimum or maximum, respectively. In this case, we call $\min X$ the *bottom element*, denoted by $\bot$, and $\max X$ the *top element*, denoted

by $\top$. A partially ordered set $\langle X, \preceq \rangle$ is a *lattice* iff it is a meet-semilattice and a join-semilattice. A lattice is bounded if X has a minimum and a maximum. A lattice is complete if every set $\mathcal{X} \subseteq X$, including the empty set, has a greatest lower bound (meet) and least upper bound (join). A complete lattice is always bounded.

Given a pair of partially ordered sets $\langle X_1, \preceq_1 \rangle$ and $\langle X_2, \preceq_2 \rangle$ we define the *product partial order* of $\preceq_1$ and $\preceq_2$, denoted by $\preceq_1 \times \preceq_2$, over $X_1 \times X_2$ such that:

$$\langle x_1, x_2 \rangle \ (\preceq_1 \times \preceq_2) \ \langle x_1', x_2' \rangle \quad \iff \quad x_1 \preceq_1 x_1' \wedge x_2 \preceq_2 x_2'$$

The product order preserves lattice properties, i.e., iff $\langle X_1, \preceq_1 \rangle$ and $\langle X_2, \preceq_2 \rangle$ are both a meet- or join-semilattice, then so is the partially ordered set $\langle X_1 \times X_2, \preceq_1 \times \preceq_2 \rangle$. Likewise, boundedness and completeness are preserved.

Monotone and Continuous Functions. While there exist more general definitions of monotone and continuous functions, for the purposes of this thesis, the following specialized definitions suffice. A function $f : \mathbb{R} \to \mathbb{R}$ is *strictly monotone* iff either $f(x) < f(x')$ for all $x < x'$ (*strictly increasing*) or $f(x) > f(x')$ for all $x < x'$ (*strictly decreasing*). It is *continuous at a point* $x_0 \in \mathbb{R}$ iff for every real number $\epsilon > 0$ there exists a real number $\delta > 0$ such that for $|f(x) - f(x_0)| < \epsilon$ for all $x \in \mathbb{R}$ with $|x - x_0| < \delta$, also known as the *epsilon-delta definition of continuity*. It is *continuous* simpliciter iff it is continuous at every point $x_0 \in \mathbb{R}$.

Interval Arithmetic. A *closed interval*, denoted by $[a, b]$, over a set of numbers, e.g., $\mathbb{Q}$ or $\mathbb{R}$, is a set containing all numbers x such that $a \leq x \leq b$. For an interval $\mathtt{I} = [a, b]$, we use $\min \mathtt{I}$ and $\max \mathtt{I}$ to denote the lower bound a and upper bound b, respectively. We introduce the following arithmetic operations on intervals:

$$\begin{aligned}
[a_1, b_1] \boxplus [a_2, b_2] &\coloneqq [a_1 + a_2, b_1 + b_2] \\
[a_1, b_1] \boxminus [a_2, b_2] &\coloneqq [a_1 - b_2, b_1 - a_2] \\
[a_1, b_1] \boxdot [a_2, b_2] &\coloneqq [\min\{a_1a_2, a_1b_2, a_2b_1, a_2b_2\}, \max\{a_1a_2, a_1b_2, a_2b_1, a_2b_2\}] \\
[a_1, b_1] \boxslash [a_2, b_2] &\coloneqq [a_1, b_1] \boxdot \left[\frac{1}{b_2}, \frac{1}{a_2}\right] \text{ if } 0 \notin [a_2, b_2]
\end{aligned}$$

Note that the respective intervals contain exactly those numbers that can be obtained by combining the numbers contained in both intervals with the respective scalar operator. For operations operating on scalars and intervals, we use the same notation treating a scalar x as a closed interval $[x, x]$.

Words and Languages. For a non-empty set Σ of *symbols*, coined an *alphabet*, we denote the set of all finite and infinite sequences over Σ by $\Sigma^\star$ and Σ^ω, respectively. We also refer to such sequences as *words*. We denote the empty word by ϵ. Further, let $\Sigma^\infty \coloneqq \Sigma^\star \cup \Sigma^\omega$ be the set of finite and infinite words over Σ.

Formally, a word $\sigma \in \Sigma^{\infty}$ over Σ is a partial function $\sigma : \mathbb{N}^+ \rightharpoonup \Sigma$ without *gaps* starting at 1, i.e., a functional binary relation $\sigma \subseteq \mathbb{N}^+ \times \Sigma$ such that:

$$\forall k \geq 1 : k \in \mathrm{Dom}(\sigma) \implies \forall 1 \leq i \leq k : i \in \mathrm{Dom}(\sigma)$$

As a word σ is a partial function, we follow the usual notation and denote its i-th symbol by $\sigma(i)$. We denote the length of a finite word by $|\sigma|$. Note that every partial function $\varsigma : \mathbb{N} \rightharpoonup \Sigma$ can be *normalized* to represent a word by removing all gaps and shifting the symbols to start at 1. We call such functions with potential gaps *semiwords* and denote their normalization by $\mathrm{Word}(\varsigma)$. According to this definition, words and semiwords are sets of pairs $\langle i, \mathrm{a} \rangle \in \mathbb{N}^+ \times \Sigma$ where $i \in \mathbb{N}^+$ is an index into the word (or semiword) and $\mathrm{a} \in \Sigma$ is the symbol found at the respective index. This representation is advantageous for various definitions in this thesis.

For $\sigma \in \Sigma^{\star}$ and $\sigma' \in \Sigma^{\infty}$, let $\sigma \diamond \sigma'$ denote the concatenation of σ and σ':

$$\sigma \diamond \sigma' \coloneqq \mathrm{Word}\big(\sigma \cup \{ \langle |\sigma| + i, \mathrm{a} \rangle \mid \langle i, \mathrm{a} \rangle \in \sigma' \}\big)$$

For a set $A \subseteq \Sigma$ of symbols and $\sigma \in \Sigma^{\infty}$, we define the *$A$-projection* of σ, denoted by $\sigma{\downarrow_A}$, as the word obtained by removing all symbols from σ which are not in A:

$$\sigma{\downarrow_A} \coloneqq \mathrm{Word}\big(\{ \langle i, \mathrm{a} \rangle \in \sigma \mid \mathrm{a} \in A \}\big)$$

For a finite word $\sigma \in \Sigma^{\star}$ of length $n \in \mathbb{N}$, we also use the notation $(\mathrm{a}_i)_{i=1}^{n}$. Here, a_i denotes the i-th symbol of σ, i.e., $\mathrm{a}_i = \sigma(i)$. If $n > 0$, we refer to a_n as the *head* of σ, denoted by $\mathrm{Head}(\sigma)$. We refer to the prefix of σ without the head as the *tail*, denoted by $\mathrm{Tail}(\sigma)$, i.e., $\mathrm{Tail}(\sigma) = (\mathrm{a}_i)_{i=1}^{n-1}$. The tail of the empty word ϵ is the empty word itself. Hence, if $n > 0$, then $\sigma = \mathrm{Tail}(\sigma) \diamond \mathrm{Head}(\sigma)$. We use $\sigma[k..j]$ to denote the part of the word between the k-th and j-th symbol and $\sigma[k..]$ to denote the part of the word starting at the k-th symbol. For a word $\sigma \in \Sigma^{\infty}$, we denote the set of finite prefixes of σ by $\mathrm{Pref}(\sigma)$. In case σ is finite, then $\mathrm{Pref}(\sigma)$ includes σ itself. We define a partial order $\leq \subseteq \Sigma^{\star} \times \Sigma^{\star}$ over finite words such that $\sigma \leq \sigma'$ iff σ is a prefix of σ', i.e., iff $\sigma \in \mathrm{Pref}(\sigma')$. As usual, we denote the irreflexive kernel of $\leq$ by $<$.

A *language* $\mathcal{L}$ over an alphabet Σ is a set of finite words, i.e., $\mathcal{L} \subseteq \Sigma^{\star}$. For a set $A \subseteq \Sigma$, let $\mathcal{L}{\downarrow_A} \coloneqq \{ \sigma{\downarrow_A} \mid \sigma \in \mathcal{L} \}$ denote the A-projection of $\mathcal{L}$. A language $\mathcal{L}$ is *prefix-closed* iff it contains all prefixes of the words it contains, i.e., $\mathrm{Pref}(\sigma) \subseteq \mathcal{L}$ for all $\sigma \in \mathcal{L}$. Note that this is equivalent to $\mathrm{Tail}(\sigma) \in \mathcal{L}$ for all $\sigma \in \mathcal{L}$.

Boolean Expressions. Given a non-empty set AP of *atomic propositions*, the set of *Boolean expressions* $\mathbb{B}[\mathsf{AP}]$ over AP is defined according to the following grammar:

$$\mathbb{B}[\mathsf{AP}] \ni \phi \quad ::= \quad \mathtt{true} \mid p \in \mathsf{AP} \mid \neg\phi \mid \phi \wedge \phi$$

The semantics of Boolean expressions are defined as usual. Formally, we define a denotational semantics $[\![\cdot]\!]_{\mathbb{B}} : \mathbb{B}[\mathsf{AP}] \to \wp(\wp(\mathsf{AP}))$ mapping each Boolean expression

to a set of *satisfying sets*. The function $[\![\cdot]\!]_{\mathbb{B}}$ is defined as follows:

$$[\![\texttt{true}]\!]_{\mathbb{B}} \coloneqq \wp(\mathsf{AP}) \qquad [\![p]\!]_{\mathbb{B}} \coloneqq \{P \subseteq \mathsf{AP} \mid p \in P\}$$
$$[\![\neg\phi]\!]_{\mathbb{B}} \coloneqq \wp(\mathsf{AP}) \setminus [\![\phi]\!]_{\mathbb{B}} \qquad [\![\phi_1 \wedge \phi_2]\!]_{\mathbb{B}} \coloneqq [\![\phi_1]\!]_{\mathbb{B}} \cap [\![\phi_2]\!]_{\mathbb{B}}$$

The negation operator $\neg$ takes precedence over $\wedge$. As usual, we use parentheses to clarify the syntactical structure of nested expressions. For notational convenience, we define the following operators as usual:

- $\phi_1 \vee \phi_2 \coloneqq \neg(\neg\phi_1 \wedge \neg\phi_2)$
- $\phi_1 \rightarrow \phi_2 \coloneqq \neg\phi_1 \vee \phi_2$
- $\texttt{false} \coloneqq \neg\texttt{true}$

Note that $\langle \wp(\wp(\mathsf{AP})), \subseteq \rangle$ is a complete lattice with $\sqcup = \cup$ and $\sqcap = \cap$. Every Boolean expressions ϕ corresponds to an element $[\![\phi]\!]_{\mathbb{B}}$ of this lattice. Furthermore, for every element of the lattice, there exists a *characteristic* Boolean expression in conjunctive normal form, i.e., using only negation $\neg$ and conjunction $\wedge$. The $\sqcup$ and $\sqcap$ operators of the lattice correspond to the connectives $\vee$ and $\wedge$ as defined above.

We use the operator symbols $\wedge$, $\vee$, and $\neg$ for Boolean expressions and as logical meta operators, e.g., when stating theorems and definitions.

2.2 Formal Models

Formal models describe the operational behavior of systems in a mathematically rigorous way such that it becomes amenable to formal analysis techniques like model checking [CW96; BK08]. We consider system models with discrete time and continuous time, both based on the general model of transition systems.

2.2.1 Transition Systems

As a general mathematical model describing the operational behavior of systems, we rely on *transition systems* (TSs). In the literature, different variants of transition systems have been proposed, all rooted in the seminal work by Keller [Kel76]. For the purposes of this thesis, we will use the following variant.

Definition 2.2.1 *A* transition system $\mathfrak{S}$ *is a tuple* $\langle S, I, \mathsf{Act}, \twoheadrightarrow \rangle$ *where*
- S *is a set of* states*,*
- $I \subseteq S$ *is a non-empty set of* initial states*,*
- Act *is a set of* actions*, and*
- $\twoheadrightarrow \subseteq S \times \mathsf{Act} \times S$ *is a* transition relation*.*

We call $\mathfrak{S}$ *finite* iff S and $\twoheadrightarrow$ are finite.

The intuitive operational semantics is that at any point in time, the system is in some state $s \in \mathcal{S}$. From this state, it may then *transition* to a *successor state* $s' \in \mathcal{S}$ iff there exists a transition $\langle s, \alpha, s' \rangle \in \twoheadrightarrow$ between s and s'. Each transition has an action $\alpha \in \mathrm{Act}$ attached to it. In the context of this thesis, actions usually correspond to types of events that occur when the system takes a transition, i.e., an event is an occurrence of an action. If multiple transitions exist in a state, which of the possible transitions is taken in a state is left unspecified by a transition system. It may, for instance, be determined by the input given to the system. Note that transition systems are very general, as they can be infinite and do neither constrain the state set nor the action set in any way. As such, they are computationally universal, i.e., any Turing machine can be molded into a transition system. In fact, they go beyond what is computationally possible as, in general, a state set and action set may be uncountably infinite, which is required for modeling continuous time.

For a subset $S \subseteq \mathcal{S}$ of states and a subset $A \subseteq \mathrm{Act}$ of actions, let $\mathrm{Post}(S, A)$ denote the set of states reachable from a state $s \in S$ via some transition with some action $\alpha \in A$. Formally, we define $\mathrm{Post} \colon \wp(\mathcal{S}) \times \wp(\mathrm{Act}) \to \wp(\mathcal{S})$ as follows:

$$\mathrm{Post}(S, A) \coloneqq \bigcup_{s \in S} \bigcup_{\alpha \in A} \{\, s' \in \mathcal{S} \mid \langle s, \alpha, s' \rangle \in \twoheadrightarrow \,\} \tag{2.1}$$

A state $s \in \mathcal{S}$ is *terminal* iff $\mathrm{Post}(\{s\}, \mathrm{Act}) = \emptyset$, otherwise, it is *live*. For notational convenience, we abbreviate $\mathrm{Post}(\{s\}, \{\alpha\})$ as $\mathrm{Post}(s, \alpha)$.

We call $\mathfrak{S}$ *deterministic* iff $|I| = 1$ and $|\mathrm{Post}(s, \alpha)| \leq 1$ for all states $s \in \mathcal{S}$ and actions $\alpha \in \mathrm{Act}$, i.e., iff there exists exactly one initial state and at most one transition per state and action. Otherwise, we call it *non-deterministic*. We call $\mathfrak{S}$ *input-enabled* iff $|\mathrm{Post}(s, \alpha)| \geq 1$ for all states $s \in \mathcal{S}$ and actions $\alpha \in \mathrm{Act}$.

For a finite word $\sigma \in \mathrm{Act}^\star$, let $\mathrm{After}(\sigma)$ denote the set of states *reached after* σ, i.e., the set of states the system may be in after taking transitions labeled with the actions on σ. Formally, we define $\mathrm{After} \colon \mathrm{Act}^\star \to \wp(\mathcal{S})$ recursively as follows:

$$\mathrm{After}(\epsilon) \coloneqq I \qquad \mathrm{After}(\sigma \diamond \alpha) \coloneqq \mathrm{Post}(\mathrm{After}(\sigma), \{\alpha\}) \tag{2.2}$$

A word $\sigma \in \mathrm{Act}^\star$ is *accepted* by $\mathfrak{S}$ iff $\mathrm{After}(\sigma) \neq \emptyset$. We refer to the words accepted by $\mathfrak{S}$ as *traces*. The *language* $\mathcal{L}(\mathfrak{S})$ of $\mathfrak{S}$ is the set of its traces, i.e.:

$$\mathcal{L}(\mathfrak{S}) \coloneqq \left\{ \sigma \in \mathrm{Act}^\star \mid \mathrm{After}(\sigma) \neq \emptyset \right\}$$

It is easy to see that $\mathcal{L}(\mathfrak{S})$ is prefix-closed.

If $\mathfrak{S}$ is finite, then $\mathfrak{S}$ can be seen as a non-deterministic finite automaton over Act where all states are accepting. The class of non-deterministic finite automata where all states are accepting have been studied in the literature [Cev+14; KRS09]. They are exactly the prefix-closed languages over finite alphabets [KRS09, Theorem 8]. Minimization for them is known to be PSPACE-hard [KRS09, Corollary 3].

An *execution fragment* of $\mathfrak{S}$ is a word $\hat{\rho} = \left(\langle s_i, \alpha_i, s_i' \rangle\right)_{i=1}^{n} \in \twoheadrightarrow^\star$ such that $s_{i-1}' = s_i$ for all $1 < i \leq n$. A *run* of $\mathfrak{S}$ is an execution fragment starting in an initial state, i.e., where $s_1 \in I$. We denote the set of all runs of $\mathfrak{S}$ by $\mathsf{Runs}(\mathfrak{S})$. It is easy to see that $\mathsf{Runs}(\mathfrak{S})$ is a prefix-closed language, just like $\mathcal{L}(\mathfrak{S})$. Each run $\rho = \left(\langle s_i, \alpha_i, s_i' \rangle\right)_{i=1}^{n} \in \mathsf{Runs}(\mathfrak{S})$ induces a trace $(\alpha_i)_{i=1}^{n} \in \mathsf{Act}^\star$, denoted by $\mathsf{Trace}(\rho)$. A run is *terminal* iff it ends in a terminal state, i.e., iff $\mathsf{Post}(\{s_n'\}, \mathsf{Act}) = \varnothing$. For a given run $\rho \in \mathsf{Runs}(\mathfrak{S})$, let $\mathsf{After}(\rho)$ denote the set of states the system may be in after ρ. Formally, we define $\mathsf{After}(\rho) : \mathsf{Act}^\star \to \wp(\mathcal{S})$ recursively as follows:

$$\mathsf{After}(\epsilon) \;:=\; I \qquad\qquad \mathsf{After}(\rho \diamond \langle s, \alpha, s' \rangle) \;:=\; \{s'\}$$

A *path* is a word $\pi = (s_i)_{i=1}^{n}$ such that $s_{i+1} \in \mathsf{Post}(\{s_i\}, \mathsf{Act})$ for all $1 \leq i < n$. We denote the set of all paths starting in a state $s \in \mathcal{S}$ by $\mathsf{Paths}(s)$.

State and Transition Labels. When modeling systems, it is often useful to attach additional *labels* to states and transitions. For instance, it is quite common to label transitions or states with sets of atomic propositions for model checking purposes [BK08]. For a given transition system $\langle \mathcal{S}, I, \mathsf{Act}, \twoheadrightarrow \rangle$, a *state labeling* is a tuple $\langle \Lambda, \lambda \rangle$ where Λ is a set of *state labels* and $\lambda : \mathcal{S} \to \Lambda$ is a *state-labeling function*. Analogously, a *transition labeling* is a tuple $\langle \Gamma, \gamma \rangle$ where Γ is a set of *transition labels* and $\gamma : \twoheadrightarrow \to \Gamma$ is a *transition-labeling function*.

Depending on the context, one may think of Act as *inputs* and Λ as *outputs*. Under this interpretation, state-labeled transition systems are a potentially infinite and non-deterministic generalization of Moore machines [cf. Moo56].

Modeling Languages and Formats. Over the years, a plethora of different modeling languages and formats have emerged to conveniently describe different variants of transition systems and transfer them between tools. Process calculi such as Hoare's CSP (*Communicating Sequential Processes*) calculus [Hoa78] or Milner's CCS (*Calculus of Communicating Systems*) [Mil80], can be seen as early variants of *compositional* modeling languages. They enable the description of transition systems in terms of multiple communicating processes. The LOTOS specification language [BB87], the PRISM language of the PRISM model checker [KNP11], the Promela language of the Spin model checker [Hol97], and the Modest language of the Modest Toolset [Hah+13], are noteworthy modeling languages building upon the idea of composition modeling. The *JSON automata network interchange* (JANI) format [Bud+17], is a popular format for interchanging models between tools and supports a huge variety of different model types. It is mostly used in the quantitative verification community, where system properties such as reachability probabilities or expected rewards and timings are studied. JANI is supported by most state-of-the-art tools in that area, e.g., ePMC [Hah+14], the Modest Toolset [HH14], and Storm [Hen+22]. For the purpose of this thesis, we stay mostly agnostic with respect to modeling languages and formats.

Instead we directly work with the mathematical objects they describe. Momba and our tool support is centered around the JANI format.

Discrete-Time Models. In discrete-time models, time is not explicitly modeled and assumed to pass in discrete steps between transition, however, not necessarily in equidistant amounts. While any real system always operates within the spacetime continuum, discrete-time models abstract away that fact. As such, they are useful if we do not care about the actual amount of time that may pass between transitions, for instance, if we are interested in properties or aim to describe behaviors that are independent of the actual amount of time passing between transitions.

Example 2.1 As an illustrative example, Figure 2.1 shows a discrete-time model of a coffee machine. Here, as usual, states are depicted as circles (or rounded rectangles) and transitions between the states are depicted as arrows, labeled with the respective actions. In this case, the set of actions Act is defined as follows:

$$\text{Act} \coloneqq \{\texttt{request}, \texttt{dispense}, \texttt{blink}, \texttt{pump\_fault}, \texttt{short\_circuit}\}$$

The model captures the following behavior: The coffee machine accepts requests for coffee (`request`). In response to each request, it may then either dispense (`dispense`) the coffee or one of two faults may occur. Either the pump breaks (`pump_fault`), in which case the machine continues to accept requests but never dispenses a coffee, or there may be a short circuit (`short_circuit`), in which case the machine starts blinking (`blink`). Note that the model does not specify the amount of time it takes to brew a coffee. Clearly, brewing coffee is not instantaneous, so some time has to pass between each `request` and `dispense` transition. The model abstracts over this fact while still capturing useful properties of coffee machines, such that a coffee is eventually dispensed after a request unless the coffee machine is faulty.

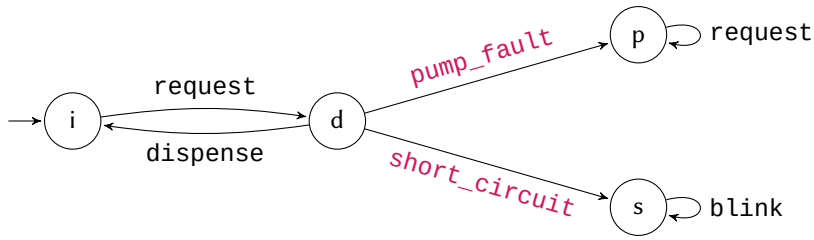


Figure 2.1: Model of a coffee machine including possible faults.

Discrete-time models and their extensions, e.g., for probabilistic behaviors, have successfully been used to describe and analyze the behaviors of a whole host of systems. To give a few examples, such systems range from network protocols [Duf+06; KNP12] to space exploration [HLP01]. Furthermore, they are also commonly used to describe planning problems [GN92; Hof+20].

2.2.2 Continuous-Time Models

While discrete-time models can faithfully capture the behavior of many systems where the timing of events does not play a role, they are not suitable in cases where the timing does play a role. Coming back to Example 2.1, one could imagine that even with a faulty pump, a coffee will be dispensed, however, it will be dispensed *slower* than when the pump is not faulty. To model this behavior, we need to consider time explicitly. This is where *continuous-time models* become necessary.

Definition 2.2.2 *For a set* Act *of actions such that* $\mathsf{Act} \cap \mathbb{R}_0^+ = \emptyset$*, a* continuous-time transition system *(CTS) is a TS* $\mathfrak{C} = \langle \mathcal{S}, I, \mathsf{Act} \cup \mathbb{R}_0^+, \twoheadrightarrow \rangle$.

Note that the actions of the TS $\mathfrak{C}$ are $\mathsf{Act} \cup \mathbb{R}_0^+$. Each such action $\alpha \in \mathsf{Act} \cup \mathbb{R}_0^+$ of a CTS $\mathfrak{C}$ either corresponds to a *discrete action* $\alpha \in \mathsf{Act}$ or to the elapse of some time $\Delta t \in \mathbb{R}_0^+$. Discrete actions are assumed to be instantaneous, i.e., they do not take any time. The *duration* $\mathsf{Dur}(\rho)$ of a run $\rho \in \mathsf{Runs}(\mathfrak{C})$ of $\mathfrak{C}$ is the total elapsed time:

$$\mathsf{Dur}(\epsilon) := 0 \qquad \mathsf{Dur}(\rho \diamond \alpha) := \begin{cases} \mathsf{Dur}(\rho) + \alpha & \text{if } \alpha \in \mathbb{R}_0^+ \\ \mathsf{Dur}(\rho) & \text{otherwise} \end{cases}$$

Each run $\rho \in \mathsf{Runs}(\mathfrak{C})$ further induces a *timed word* $\mathsf{Tw}(\rho) \in (\mathbb{R}_0^+ \times \mathsf{Act})^\star$:

$$\mathsf{Tw}(\epsilon) := \epsilon \qquad \mathsf{Tw}(\rho \diamond \alpha) := \begin{cases} \mathsf{Tw}(\rho) \diamond \langle \mathsf{Dur}(\rho), \alpha \rangle & \text{if } \alpha \in \mathsf{Act} \\ \mathsf{Tw}(\rho) & \text{otherwise} \end{cases}$$

A timed word is a sequence of tuples $\langle t, \alpha \rangle$ where t is the time the action α occurred. In the following, we will work with timed words, as it usually only matters when certain actions occurred while the exact steps in which time passes are irrelevant. This is in line with existing work on timed system models [AD91].

Timed Automata. *Timed automata* are a well-established formalism to model continuous-time systems. They have been pioneered by Alur and Dill [AD91]. Timed automata extend finite transition systems with *real-valued clocks* over which *clock constraints* can be defined to constrain how time may evolve.

Definition 2.2.3 *For a finite set* $\mathbb{C}$ *of* clocks*, a* clock constraint g *is an expression of the form* $x - y \sim c$ *where* $x, y \in \mathbb{C} \cup \{0\}$, $\sim \in \{<, \leq\}$, *and* $c \in \mathbb{Q}$ *[BY03].*

We denote the set of all finite sets of clock constraints over $\mathbb{C}$ by $\mathsf{C}[\mathbb{C}]$.

For a finite set $\mathbb{C}$ of clocks, a *clock valuation* $\eta : \mathbb{C} \to \mathbb{R}_0^+$ is a function mapping each clock to a real number. We denote the clock valuation assigning zeros to all clocks by η_0, i.e., $\eta_0(x) := 0$ for all $x \in \mathbb{C}$. A clock valuation η satisfies a clock

constraint $x - y \sim c$, denoted by $\eta \vDash g$, iff $\eta'(x) - \eta'(y) \sim c$ where $\eta' : \mathbb{C} \cup \{0\} \to \mathbb{R}_0^+$ such that $\eta'(x) = \eta(x)$ for $x \in \mathbb{C}$ and $\eta(0) = 0$. A clock valuation η satisfies a set of clock constraints $G \subseteq \mathsf{C}[\mathbb{C}]$, denoted by $\eta \vDash G$, iff $\eta \vDash g$ for each $g \in G$. Given a set $\mathsf{R} \subseteq \mathbb{C}$ of clocks and a clock valuation $\eta : \mathbb{C} \to \mathbb{R}_0^+$, we define $(\eta \downarrow \mathsf{R}) : \mathbb{C} \to \mathbb{R}_0^+$ such that $(\eta \downarrow \mathsf{R})(x) = 0$ if $x \in \mathsf{R}$ and $(\eta \downarrow \mathsf{R})(x) = \eta(x)$ otherwise. That is, $\eta \downarrow \mathsf{R}$ *resets* the clocks in R to zero. For $\Delta t \in \mathbb{R}_0^+$, we define $(\eta \oplus \Delta t) : \mathbb{C} \to \mathbb{R}_0^+$ such that $(\eta \oplus \Delta t)(x) = \eta(x) + \Delta t$. That is, $\eta \oplus \Delta t$ advances the values of all clocks by Δt into the future. With these definitions in place, we now define timed automata.

Definition 2.2.4 *For a finite set* $\mathbb{C}$ *of* clocks *and a finite set* Act *of* actions*, a* timed automaton $\mathfrak{T}$ *is a tuple* $\langle L, I, \mathsf{Act}, \mathbb{C}, E, \mathsf{Inv}\rangle$ *where*

- L *is a finite set of* locations*,*
- $I \subseteq L$ *is a set of* initial locations*,*
- $E \subseteq L \times \mathsf{C}[\mathbb{C}] \times \mathsf{Act} \times \wp(\mathbb{C}) \times L$ *is an* edge relation*, and*
- $\mathsf{Inv} : L \to \mathsf{C}[\mathbb{C}]$ *assigns a* progress invariant *to each* location*.*

We follow the usual terminology and call the states of a timed automaton *locations* in order to distinguish them from the states of its semantics.

The intuitive semantics is that the state of a system, at any point in time, is given by some location $l \in L$ and a clock valuation $\eta : \mathbb{C} \to \mathbb{R}_0^+$ assigning a value $\eta(x) \in \mathbb{R}_0^+$ to each clock $x \in \mathbb{C}$, i.e., the state is a pair $\langle l, \eta\rangle$. The progress invariant describes how time in any given location l may pass. For any state $\langle l, \eta\rangle$, η must satisfy $\mathsf{Inv}(l)$, i.e., $\eta \vDash \mathsf{Inv}(l)$. Among other things, this enables modeling upper bounds on the time the system may be in a given location. A transition $\langle l, G, \alpha, \mathsf{R}, l'\rangle$ can only be taken in a state $\langle l, \eta\rangle$ iff η satisfies the *clock guard* G and $(\eta \downarrow \mathsf{R})$ satisfies the progress invariant $\mathsf{Inv}(l')$ of the successor location l'. It will then lead to the state $\langle l', \eta \downarrow \mathsf{R}\rangle$ where the system is in location l' and the clocks in R are reset to zero. Formally, the real-time evolution of a timed automaton $\mathfrak{T}$ induces a continuous-time transition system $[\![\mathfrak{T}]\!]$ defined as follows.

Definition 2.2.5 *For a TA* $\mathfrak{T} = \langle L, I, \mathsf{Act}, \mathbb{C}, E, \mathsf{Inv}\rangle$*, we define the CTS*

$$[\![\mathfrak{T}]\!] \;:=\; \left\langle L \times (\mathbb{C} \to \mathbb{R}_0^+), I \times \{\eta_0\}, \mathsf{Act} \cup \mathbb{R}_0^+, \twoheadrightarrow \right\rangle$$

where the transition relation $\twoheadrightarrow$ *is the smallest relation such that:*

$$\frac{\langle l, G, \alpha, \mathsf{R}, l'\rangle \in E \qquad \eta \vDash G \qquad \eta' = (\eta \downarrow \mathsf{R}) \qquad \eta' \vDash \mathsf{Inv}(l')}{\langle \langle l, \eta\rangle, \alpha, \langle l', \eta'\rangle\rangle \in \twoheadrightarrow}$$

$$\frac{\Delta t \in \mathbb{R}_0^+ \qquad l' = l \qquad \eta' = \eta \oplus \Delta t \qquad \forall 0 \le \Delta t' \le \Delta t : (\eta \oplus \Delta t') \vDash \mathsf{Inv}(l)}{\langle \langle l, \eta\rangle, \Delta t, \langle l', \eta'\rangle\rangle \in \twoheadrightarrow}$$

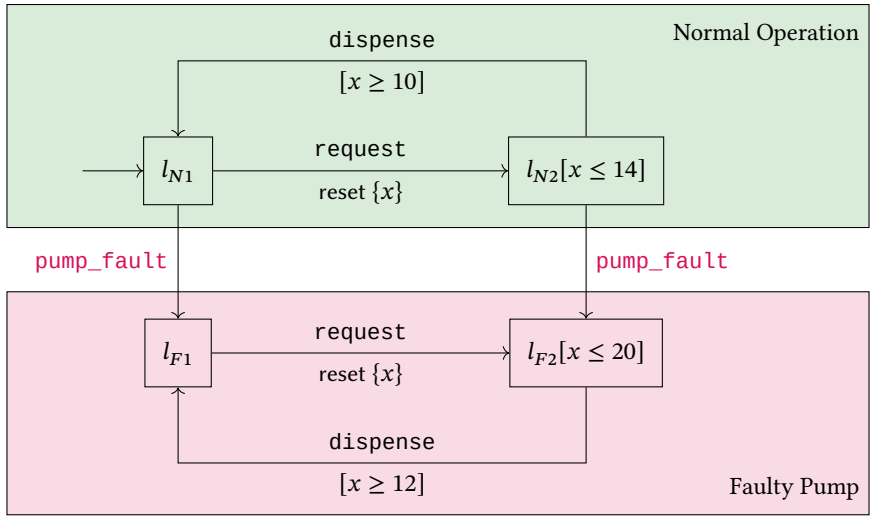


Figure 2.2: Continuous-time model of a coffee machine.

Example 2.2 Figure 2.2 shows a timed automaton modeling a coffee machine. In contrast to Figure 2.1, time is explicitly modeled. We use rectangles to depict locations and arrows to depict edges. Clock guards are written in square brackets and the clocks to be reset are indicated by reset followed by a set of clocks. The model encodes the following behavior: When the coffee machine is operating normally, it takes between 10 and 14 seconds for the coffee to be dispensed. When a coffee is requested (`request`) the clock x is reset to zero. The dispense transition (`dispense`) only becomes enabled, after the clock x reaches 10 seconds. The progress invariant of l_{N2} ensures that the machine takes the dispense transition after at most 14 seconds. In this model, the pump may also become faulty. Unlike in Figure 2.1, the coffee is still dispensed in this case, however, at a slower rate. If the pump is faulty, it takes between 12 and 20 seconds to dispense the coffee.

As with transition systems, it can sometimes be useful to attach additional labels to the locations or edges of a timed automaton, analogously to state and transition labels. We use the same notation as for transition systems.

2.2.3 Lattice Automata

Lattice automata, pioneered by Kupferman and Lustig, generalize Boolean acceptance of classical finite automata to the multi-valued setting [KL07], providing an automata-theoretic foundation for multi-valued reasoning about and verification of systems [Kup22; BG04; BG99; AK14]. For a lattice $\langle L, \preceq \rangle$, a *lattice automaton* (LA) is a tuple $\langle L, \Sigma, Q, Q_0, \delta, F \rangle$ where Σ is a finite alphabet, Q is a finite set of states,

$Q_0 : Q \to L$, $\delta : Q \times \Sigma \times Q \to L$, and $F : Q \to L$. A run of a lattice automaton on a word $\sigma = (a_i)_{i=1}^n \in \Sigma^\star$ of length n is a sequence $r = (q_i)_{i=0}^n \in Q^\star$ of $n+1$ states. Each such pair of a word and a run, induces a value of the lattice L:

$$\mathrm{val}(\sigma, r) \;\coloneqq\; Q_0(q_0) \sqcap \left(\prod_{i=1}^{n} \delta(q_{i-1}, a_i, q_i) \right) \sqcap F(q_n)$$

The *L-language* $\mathcal{L} : \Sigma^\star \to L$ of a lattice automaton maps each word $\sigma \in \Sigma^\star$ to an element of the lattice. It is defined by joining the values induced by all runs on a word, i.e., $\mathcal{L}(\sigma) \coloneqq \bigsqcup \{ \mathrm{val}(\sigma, r) \mid r \text{ is a run on } \sigma \}$ [KL07]. Kupferman and Lustig study *L-languages* as induced by lattice automata and their closure properties.

Due to the finiteness of the state set and alphabet, only finitely many elements of the lattice L can actually be generated by words. Hence, one does not loose any expressiveness[6] by restricting lattice automata to finite lattices. While the original definition is not restricted to finite lattices, the results shown by Kupferman and Lustig indeed operate under the assumption that the lattices have a finite number of elements [cf. KL07, Section 3]. Furthermore, Kupferman and Lustig often refer to top and bottom elements, which are only guaranteed to be defined if the lattice is bounded and must not be assumed if LA were defined over arbitrary lattices. As one does not loose any expressiveness by restricting lattice automata to finite lattices and Kupferman and Lustig assume finiteness, and thus bondendess, for their results, we assume that it was their intention to restrict LA to finite lattices. Thus, in the following we assume that LA are always defined over finite lattices.

Following Kupferman and Lustig, we may interpret $\mathrm{val}(\sigma) = \top$ and $\mathrm{val}(\sigma) = \bot$ as clear acceptance and rejection of a word σ, respectively. In fact, the class of traditional non-deterministic finite automata corresponds to the class of lattice automata over $\{\top, \bot\}$ [KL07]. A lattice automaton is called *simple*, iff the image of Q_0 and δ is $\{\top, \bot\}$. For a simple lattice automaton, $Q_0(q) = \top$ marks q as an initial state and $\delta(q, a, q') = \top$ marks the existence of an a-labeled transition from q to q'. As established by Kupferman and Lustig, every LA can be transformed into an equivalent[7] simple one with a linear blowup in the size of the lattice [KL07, Theorem 6].

Following the original work by Kupferman and Lustig, which focused primarily on the theoretical framework and closure properties of LA and L-languages, the theory of lattice automata has been further extended in subsequent works. In particular, minimization of deterministic lattice automata has been studied [HK11; HK15], which turns out to be NP-complete. Furthermore, approximations of lattice automata [HK12] and the ability of changing values of individual transitions without changing the L-language of a lattice automaton [GK15] has been studied.

[6] In terms of expressible L-languages.
[7] In the sense that it preserves the L-language.

2.3 Temporal and Modal Logics

Temporal logics are used to formally specify temporal properties of systems. In case of the coffee machine example, such a property might be that every request is eventually met or met within a certain time bound. The predominant variants of temporal logic are *linear temporal logic* (LTL) [Pnu77] and *computation tree logic* (CTL) [CE81], or generalizations and variants thereof such as the modal μ-calculus [BS69; Koz82]. Temporal logics are not a focus of this thesis, however, we will encounter them a few times, so a brief introduction is in order. For a comprehensive introduction, we refer to standard textbooks [e.g. BK08]. *Modal logics* are closely related to temporal logics and used to formally specify properties over possibilities and necessities.

Temporal logics are often used in the context of model checking to specify properties a system must satisfy. Given a set of atomic propositions AP, it is common to assume that the system is modeled as a transition system $\langle \mathcal{S}, I, \mathrm{Act}, \twoheadrightarrow \rangle$ where states are annotated with sets of atomic propositions by some state labeling $\langle \wp(\mathsf{AP}), \lambda \rangle$. In fact, when combining such a state labeling with a transition system, we obtain a typical definition used for model checking [BK08, p. 20].

2.3.1 Linear Temporal Logic (LTL)

Linear temporal logic has been pioneered by Pnueli [Pnu77]. It extends traditional Boolean expressions with temporal operators to express properties of linear strands of time. For a set AP of atomic propositions, we define the set of *LTL formulas* $\mathsf{LTL}[\mathsf{AP}]$ over AP according to the following grammar:

$$\mathsf{LTL}[\mathsf{AP}] \ni \varphi \quad ::= \quad \mathtt{true} \mid p \in \mathsf{AP} \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \,\mathsf{U}\, \varphi \mid \bigcirc\varphi$$

In addition to the traditional Boolean operators, LTL introduces the *until operator* U and the *next operator* $\bigcirc$. The semantics of LTL is defined over infinite words $\pi \in \wp(\mathsf{AP})^\omega$ in terms of a binary satisfaction relation $\vDash_{\mathsf{LTL}} \subseteq \wp(\mathsf{AP})^\omega \times \mathsf{LTL}[\mathsf{AP}]$ defined as follows:

$$\begin{aligned}
&\pi \vDash_{\mathsf{LTL}} \mathtt{true} && \\
&\pi \vDash_{\mathsf{LTL}} p && \text{iff } p \in \pi(1) \\
&\pi \vDash_{\mathsf{LTL}} \neg\varphi && \text{iff } \pi \nvDash_{\mathsf{LTL}} \varphi \\
&\pi \vDash_{\mathsf{LTL}} \varphi_1 \wedge \varphi_2 && \text{iff } \pi \vDash_{\mathsf{LTL}} \varphi_1 \text{ and } \pi \vDash_{\mathsf{LTL}} \varphi_2 \\
&\pi \vDash_{\mathsf{LTL}} \varphi_1 \,\mathsf{U}\, \varphi_2 && \text{iff } \exists n \in \mathbb{N} : \pi[n..] \vDash_{\mathsf{LTL}} \varphi_2 \text{ and } \forall 1 \leq i < n : \pi[i..] \vDash_{\mathsf{LTL}} \varphi_1 \\
&\pi \vDash_{\mathsf{LTL}} \bigcirc\varphi && \text{iff } \pi[2..] \vDash_{\mathsf{LTL}} \varphi
\end{aligned}$$

The formula $\varphi_1 \,\mathsf{U}\, \varphi_2$ expresses that φ_2 eventually holds and, until it does, φ_1 holds. The formula $\bigcirc\varphi$ expresses that φ holds next. We also define $\vee$, $\rightarrow$, and `false` as

syntactic sugar like we did for Boolean expressions. In addition, we define $\Diamond\varphi$, read as *eventually* φ, as $\texttt{true}\,\mathsf{U}\,\varphi$, and $\Box\varphi$, read as *always* φ, as $\neg(\Diamond(\neg\varphi))$.

Example 2.3 Coming back to the coffee machine example, we can define the property that every request for a coffee is eventually met as:

$$\Box(\texttt{request} \rightarrow \Diamond\texttt{dispense})$$

Note that the discrete-time model of the coffee machine (see Figure 2.1) does not satisfy this property as faults may prevent a coffee from being dispensed.

For continuous-time systems, the real-time extension of LTL introduces a timed until operator that requires that φ_2 holds within a certain time bound. For further details, we refer to the existing literature [AH90; Hen98; Bou07].

2.3.2 Computation Tree Logic (CTL)

While the semantics of LTL is defined over infinite words corresponding to linear strands of time with a predetermined future, CTL takes a different approach, known as *branching time* [CE81]. For a set AP of atomic propositions, the set of *CTL formulas* $\mathsf{CTL}[\mathsf{AP}]$ over AP is defined according to the following grammar:

$$\mathsf{CTL}[\mathsf{AP}] \ni \Psi \quad ::= \quad \texttt{true} \mid p \in \mathsf{AP} \mid \neg\Psi \mid \Psi \wedge \Psi \mid \mathsf{E}[\Psi\,\mathsf{U}\,\Psi] \mid \mathsf{EX}\,\Psi \mid \mathsf{A}[\Psi\,\mathsf{U}\,\Psi]$$

As CTL requires a time with branches, its semantics is usually defined over *Kripke structures* [Kri63], a variation of transition systems. For our purposes, we define the semantics of CTL over a transition system $\langle \mathcal{S}, I, \mathsf{Act}, \twoheadrightarrow \rangle$, whose states are labeled with $\langle \wp(\mathsf{AP}), \lambda \rangle$, in terms of a binary satisfaction relation $\models_{\mathsf{CTL}} \subseteq \mathcal{S} \times \mathsf{CTL}[\mathsf{AP}]$:

$$\begin{aligned}
&s \models_{\mathsf{CTL}} \texttt{true} && \\
&s \models_{\mathsf{CTL}} p && \text{iff } p \in \lambda(s) \\
&s \models_{\mathsf{CTL}} \neg\Psi && \text{iff } s \not\models_{\mathsf{CTL}} \Psi \\
&s \models_{\mathsf{CTL}} \Psi_1 \wedge \Psi_2 && \text{iff } s \models_{\mathsf{CTL}} \Psi_1 \wedge s \models_{\mathsf{CTL}} \Psi_2 \\
&s \models_{\mathsf{CTL}} \mathsf{E}[\Psi_1\,\mathsf{U}\,\Psi_2] && \text{iff } \exists\,(s_i)_{i=1}^{n} \in \mathsf{Paths}(s) : \exists k \in \mathbb{N} : \\
&&& \quad s_k \models_{\mathsf{CTL}} \Psi_2 \wedge \forall 1 \leq i < n : s_i \models_{\mathsf{CTL}} \Psi_1 \\
&s \models_{\mathsf{CTL}} \mathsf{EX}\,\Psi && \text{iff } \exists s' \in \mathsf{Post}(\{s\}, \mathsf{Act}) : s' \models_{\mathsf{CTL}} \Psi \\
&s \models_{\mathsf{CTL}} \mathsf{A}[\Psi_1\,\mathsf{U}\,\Psi_2] && \text{iff } \forall\,(s_i)_{i=1}^{n} \in \mathsf{Paths}(s) : \exists k \in \mathbb{N} : \\
&&& \quad s_k \models_{\mathsf{CTL}} \Psi_2 \wedge \forall 1 \leq i < n : s_i \models_{\mathsf{CTL}} \Psi_1
\end{aligned}$$

We also define $\vee$, $\rightarrow$, and $\texttt{false}$ as syntactic sugar like we did for Boolean expressions. In addition, we further define the following syntactic sugar:

$$\mathsf{AX}\,\Psi := \neg\mathsf{EX}\,\neg\Psi \qquad \mathsf{AF}\,\Psi := \mathsf{A}[\texttt{true}\,\mathsf{U}\,\Psi] \qquad \mathsf{AG}\,\Psi := \neg\mathsf{EF}\,\neg\Psi$$
$$\mathsf{EF}\,\Psi := \mathsf{E}[\texttt{true}\,\mathsf{U}\,\Psi] \qquad \mathsf{EG}\,\Psi := \neg\mathsf{EF}\,\neg\Psi$$

While LTL expresses properties of infinite sequences without taking branching into account, CTL has explicit operators to take branching into account. For instance, the intuitive semantics of $\mathsf{E}[\Psi_1 \mathsf{U} \Psi_2]$ is that there exists some path, i.e., some way to take transitions in the underlying TS, such that on that path, Ψ_2 holds for the last state of the path and Ψ_1 holds for all states visited in between. As such quantifiers over paths can be nested, complex branching properties can be expressed.

Given a CTL formula Ψ, we denote the set of all states that do satisfy Ψ by $[\![\Psi]\!]_{\mathsf{CTL}}$, i.e., $[\![\Psi]\!]_{\mathsf{CTL}} \coloneqq \{ s \in \mathcal{S} \mid s \vDash \Psi \}$. Although LTL and CTL look similar, both allow the expression of properties that are not expressible with the other.

Example 2.4 In case of the coffee machine (see Figure 2.1), we may have a property like $\mathsf{EX}\,\mathsf{i}$ expressing that there exists a path such that in the next step the coffee machine is in the *idle state* i (assuming $\lambda(\mathsf{i}) = \{\mathsf{i}\}$). The set of states satisfying this property is $\{\mathsf{d}\}$. The property $\mathsf{AX}\,\mathsf{i}$ is not satisfied by any state because there always exists a path not leading to i in the next step. The property $\mathsf{EF}\,\mathsf{i}$ is satisfied in the state i and d but violated in the states p and s. In these states the coffee machine is stuck in a broken state and there exists no path back to the idle state i.

Modal μ-Calculus. The *modal μ-calculus* [BS69; Koz82] generalizes and extends both LTL and CTL. Like CTL, the semantics of a modal μ-calculus formula is defined over a transition system and induces a set of states. Some of the techniques we develop in this thesis work for any logic that has a semantics over transition systems and whose formulas induce sets of states. The details of the modal μ-calculus will not be relevant, we merely note that our techniques will straightforwardly generalize to it. For concrete examples, we will be using CTL.

2.3.3 Basic Modal Logic

In the literature, a variety of different modal logics have been proposed and studied [BRV01]. For our purposes, we introduce a variant known as *basic modal logic* [cf. BRV01]. For a set of atomic propositions AP, the set of *basic modal logic* formulas $\mathsf{MO}[\mathsf{AP}]$ over AP is defined according to the following grammar:

$$\mathsf{MO}[\mathsf{AP}] \ni \Psi \quad ::= \quad \mathtt{true} \mid p \in \mathsf{AP} \mid \neg\Psi \mid \Psi \wedge \Psi \mid \quad \mid \mathsf{N}\,\Psi$$

Basic modal logic extends Boolean expressions with the *necessity operator* $\mathsf{N}\,\Psi$ expressing that Ψ is necessary in all *possible worlds*. Formally, a possible world is a set of atomic propositions. The traditional semantics of basic modal logic is defined over pairs $\langle P, W \rangle$ where $W \subseteq \wp(\mathsf{AP})$ is the set of all possible worlds and $P \in W$ is a possible world. As for LTL and CTL, we capture the semantics in terms of a binary satisfaction relation $\vDash_{\mathsf{MO}} \subseteq \big(\wp(\mathsf{AP}) \times \wp(\wp(\mathsf{AP}))\big) \times \mathsf{MO}[\mathsf{AP}]$:

$$
\begin{aligned}
&\langle P, W\rangle \vDash_{\mathrm{MO}} \texttt{true} && \\
&\langle P, W\rangle \vDash_{\mathrm{MO}} p && \text{iff } p \in P \\
&\langle P, W\rangle \vDash_{\mathrm{MO}} \neg\Psi && \text{iff } \langle P, W\rangle \nvDash_{\mathrm{MO}} \Psi \\
&\langle P, W\rangle \vDash_{\mathrm{MO}} \Psi_1 \wedge \Psi_2 && \text{iff } \langle P, W\rangle \vDash_{\mathrm{MO}} \Psi_1 \wedge \langle P, W\rangle \vDash_{\mathrm{MO}} \Psi_2 \\
&\langle P, W\rangle \vDash_{\mathrm{MO}} \mathsf{N}\,\Psi && \text{iff } \forall P' \in W : \langle P', W\rangle \vDash_{\mathrm{MO}} \Psi
\end{aligned}
$$

In line with the intuitive semantics, the necessity operator $\mathsf{N}\,\Psi$ states that Ψ holds in all possible worlds, i.e., that Ψ *is necessary*. We also define $\vee$, $\rightarrow$, and `false` as syntactic sugar like we did for Boolean expressions. In addition, we define the *possibility operator* $\mathsf{P}\,\Psi \coloneqq \neg\mathsf{N}\,(\neg\Psi)$ as syntactic sugar. The possibility operator $\mathsf{P}\,\Psi$ thus states that Ψ holds in some possible world, i.e., that Ψ *is possible*.

Denotational Semantics. In addition to the satisfaction relation and analogously to $[\![\cdot]\!]_{\mathbb{B}}$ for Boolean expressions, we define a denotational semantics

$$[\![\cdot]\!]_{\mathrm{MO}} : \mathrm{MO}[\mathsf{AP}] \rightarrow \wp(\wp(\wp(\mathsf{AP})))$$

for modal logic formulas. To this end, we assume that atomic propositions only occur inside necessity or possibility operators. This restriction is necessary as formulas such as $\neg p$ are only ascribed meaning based on an arbitrary world fixed for the satisfaction relation, while the denotational semantics ascribes meaning independent of any fixed world. The denotational semantics of a modal logic formula is the set of sets of possible worlds for which the formula is satisfied:

$$
\begin{aligned}
[\![\texttt{true}]\!]_{\mathrm{MO}} &\coloneqq \wp(\wp(\mathsf{AP})) \\
[\![\neg\Psi]\!]_{\mathrm{MO}} &\coloneqq \wp(\wp(\mathsf{AP})) \setminus [\![\Psi]\!]_{\mathrm{MO}} \\
[\![\Psi_1 \wedge \Psi_2]\!]_{\mathrm{MO}} &\coloneqq [\![\Psi_1]\!]_{\mathrm{MO}} \cap [\![\Psi_2]\!]_{\mathrm{MO}} \\
[\![\mathsf{N}\,\Psi]\!]_{\mathrm{MO}} &\coloneqq \{\, W \subseteq \wp(\mathsf{AP}) \mid \forall P \in W : \langle P, W\rangle \vDash_{\mathrm{MO}} \Psi \,\}
\end{aligned}
$$

Similar to Boolean expressions, the denotational semantics corresponds to a complete lattice $\langle \wp(\wp(\wp(\mathsf{AP}))), \subseteq\rangle$. Every modal logic formula Ψ where atomic propositions only occur inside necessity or possibility operators corresponds to an element $[\![\Psi]\!]_{\mathrm{MO}}$ of this lattice. The $\sqcup$ and $\sqcap$ operators of the lattice correspond to the connectives $\vee$ and $\wedge$ as defined above. For some set $X \in \wp(\wp(\wp(\mathsf{AP})))$ of sets of possible worlds, we say that X *satisfies* Ψ iff $X \subseteq [\![\Psi]\!]_{\mathrm{MO}}$.

2.4 Configurable Systems

Most modern systems are highly *configurable*. For the purposes of this thesis, we adopt the well-studied feature-oriented approach where the variability within a

product line is described via *features* representing incremental or optional units of functionality [Zav00; Ape+13; CN02]. Given a set of *features* F, also called a *feature domain*, *configurations* are represented by sets $c \subseteq F$ of features. A system can only be configured towards *valid configurations* $\mathsf{Conf} \subseteq \wp(F)$. Valid configurations are typically modeled by *feature diagrams* [Kan+90; SHT06]. Feature diagrams also capture the hierarchical composition of features from sub features.

Figure 2.3 shows an example of a feature diagram for a vending machine product line [cf. Cla+13; Dev+14]. Features are depicted as boxes. In this case, the vending machine root feature consists of three sub features cancel, beverages, and free, where beverages is again decomposed into a soda and tea feature.

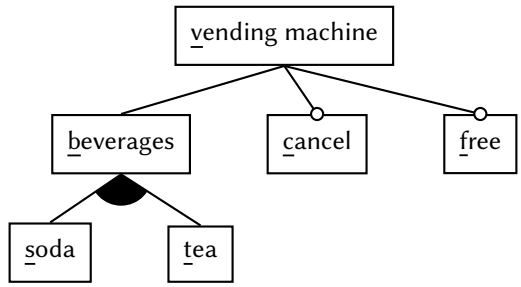


Figure 2.3: Feature diagram of a vending machine product line [cf. Cla+13; Dev+14].

The valid compositions of features are represented by different types of branchings within the diagram. Figure 2.4 shows the standard branchings together with their interpretations.

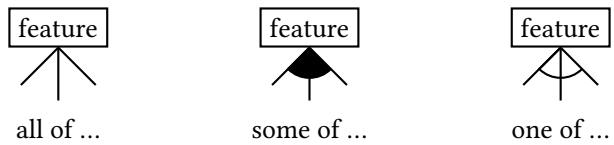


Figure 2.4: Standard branchings of feature diagrams [cf. Kan+90; SHT06].

Small circles above a feature indicate exceptions to the branchings, e.g., the hollow circles in Figure 2.3 indicate that the cancel and free feature are both optional. Hence, a soda vending machine must at least offer soda or tea and may optionally offer free drinks and the option to cancel a request for a drink.

We underline the parts of the feature name that we use in formal definitions. For example, in case of the vending machine, we have $F = \{\mathsf{v}, \mathsf{c}, \mathsf{b}, \mathsf{f}, \mathsf{s}, \mathsf{t}\}$. Note that the number of valid configurations is in general exponential in the number of features. In case of the vending machine, we have 12 distinct valid configurations.

For further details regarding feature diagrams and their semantics, we refer to Schobbens, Heymans, and Trigaux [SHT06]. For the purposes of this thesis, we take feature diagrams to be graphical representations of Boolean expressions over F. That is, each feature diagram corresponds to a Boolean expression $\phi \in \mathbb{B}[F]$ such that $[\![\phi]\!]_{\mathbb{B}}$ is the set of valid configurations Conf.

Featured Transition Systems. Behaviors of configurable systems are commonly modeled as *featured transition systems* (FTSs) [Cla+13]. FTSs are an extension of transition systems where transitions and initial states are labeled with *feature guards*. A feature guard is a Boolean expression over some feature domain F.

Definition 2.4.1 *Given a set of actions* Act *and of features* F*, a* featured transition system *(FTS)* $\mathfrak{F}$ *is a tuple* $\langle \mathcal{S}, I, \mathsf{Act}, \twoheadrightarrow, F, \mathsf{Conf}, g, \iota \rangle$ *where*

- $\langle \mathcal{S}, I, \mathsf{Act}, \twoheadrightarrow \rangle$ *is a TS,*
- $\mathsf{Conf} \subseteq \wp(F)$ *is a set of valid configurations,*
- $g : \twoheadrightarrow \to \mathbb{B}[F]$ *assigns a feature guard to each transition, and*
- $\iota : I \to \mathbb{B}[F]$ *assigns a feature guard to each initial state.*

The set Conf is a set of valid configurations as discussed above. It may be given by a feature diagram or some Boolean expression. An FTS assigns a feature guard $g(\mathrm{t})$ to each transition $\mathrm{t} \in \twoheadrightarrow$. These feature guards restrict the configurations in which the respective transitions can be taken. Additionally, an FTS assigns a feature guard $\iota(s)$ to each initial state $s \in I$. These feature guards restrict the initial states for a given configuration. We also refer to the tuple $\langle F, \mathsf{Conf}, g, \iota \rangle$ as a *feature extension* of the underlying transition system.

Formally, the *configuration semantics* of an FTS for a given valid configuration $c \in \mathsf{Conf}$ is obtained by keeping only the initial states and transitions whose guards are fulfilled by the configuration c.

Definition 2.4.2 *For a given FTS* $\mathfrak{F} = \langle \mathcal{S}, I, \mathsf{Act}, \twoheadrightarrow, F, \mathsf{Conf}, g, \iota \rangle$ *and valid configuration* $c \in \mathsf{Conf}$*, the* c-projection *of* $\mathfrak{F}$*, denoted by* $\mathfrak{F}{\downharpoonright}_c$*, is a TS*

$$\langle \mathcal{S}, I{\downharpoonright}_c, \mathsf{Act}, \twoheadrightarrow{\downharpoonright}_c \rangle$$

where $I{\downharpoonright}_c := \{ s \in I \mid c \in [\![\iota(s)]\!]_{\mathbb{B}} \}$ *and* $\twoheadrightarrow{\downharpoonright}_c := \{ \mathrm{t} \in \twoheadrightarrow \mid c \in [\![g(\mathrm{t})]\!]_{\mathbb{B}} \}$.

Note that these definitions naturally carry over to timed automata labeling edges instead of transitions and initial locations instead of initial states. For further details, we refer to the work by Cordy et al. [Cor+12].

Example 2.5 As an illustrative example, consider an email system with an encryption and signing feature [cf. Dub22]. Figure 2.5 shows the FTS and feature diagram

modeling the system. We use annotations of the form $\phi : \alpha$ to indicate that some transition $\langle s, \alpha, s' \rangle$ has a feature guard ϕ as per g. The arrows pointing at the initial states are also annotated with guards as per ι. According to the feature diagram shown in Figure 2.5, the valid configurations of the email system are $c_e = \{m, e\}$, $c_s = \{m, s\}$, and $c_{s \wedge e} = \{m, e, s\}$. Depending on the configuration, an email is signed (`sign`), encrypted (`enc`), or both, before it is sent (`send`).

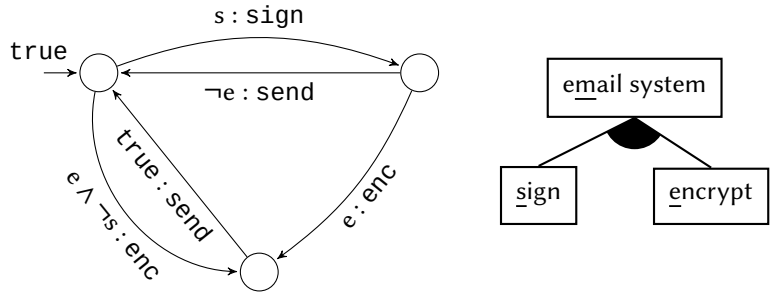


Figure 2.5: An email system with an encryption and signing feature [cf. Dub22].

2.5 Runtime Verification

Runtime verification techniques aim at detecting the violation or satisfaction of properties at runtime by observing a system's behavior [HG05; LS09]. As such, they complement traditional verification techniques, such as model checking [CW96]. Unlike those traditional techniques, runtime verification techniques do not seek to rule out property violations before a system is deployed. Instead, they enable the detection of *when* a runtime property is satisfied or violated. They are considered more lightweight and are particularly useful in scenarios where the full system cannot be verified before deployment, e.g., due to scalability issues. They are useful even if it is known that a system may violate a property as they enable taking corrective actions in response to detected incorrect behaviors [LS09].

The usual approach to runtime verification is to construct or implement a *monitor* from some specification of a property. Such a specification can, for instance, be an LTL property [BLS06b]. A monitor then reads a finite trace produced by a system and, based on that trace, indicates whether the property is satisfied or violated—a process known as *monitoring*. Monitoring can be *online*, where a monitor incrementally reads the trace from a running system as it is generated, or *offline*, where a monitor has full access to a previously recorded and stored trace [LS09].

There exist many different runtime verification techniques (see Chapter 1 for references). As paradigmatic instances, we recapitulate LTL runtime verification [BLS06b;

BLS11; BLS07] and the stream-based specification language Lola [DAn+05; Bau+20]. LTL runtime verification takes a logic- and automaton-based approach, explicitly synthesizing a monitor from an LTL formula. Lola pioneered stream-based runtime verification where output streams are computed from input streams.

2.5.1 LTL Runtime Verification

For an LTL formula φ, LTL runtime verification aims at deciding from a finite word σ whether all continuations of σ satisfy or violate φ [BLS06b; BLS11; FFM12; BLS07], which corresponds to detecting good and bad prefixes of φ [KV01]. The seminal work of Bauer, Leucker, and Schallhart introduced LTL_3 monitoring [BLS06b]. Recall that the LTL semantics are defined over infinite words (see Section 2.3.1). To lift the LTL semantics to finite words, LTL_3 monitoring uses the three-valued *truth domain* $\mathbb{B}_3 = \{\mathsf{t}, ?, \mathsf{f}\}$ where t indicates that the formula must be satisfied, f indicates that the formula must be violated, and $?$ indicates that it is unknown whether the formula is satisfied or violated. Formally, the *three-valued semantics* of an LTL formula φ over a set AP for a finite word $\sigma \in \wp(\mathrm{AP})^\star$, denoted by $[\sigma \vDash \varphi]^3_{\mathrm{LTL}}$, is given by:

$$[\sigma \vDash \varphi]^3_{\mathrm{LTL}} \coloneqq \begin{cases} \mathsf{t} & \text{if } \forall \sigma' \in \wp(\mathrm{AP})^\omega : \sigma \diamond \sigma' \vDash_{\mathrm{LTL}} \varphi \\ \mathsf{f} & \text{if } \forall \sigma' \in \wp(\mathrm{AP})^\omega : \sigma \diamond \sigma' \nvDash_{\mathrm{LTL}} \varphi \\ ? & \text{otherwise} \end{cases} \tag{2.3}$$

An LTL_3 monitor for an LTL formula φ is a deterministic finite input-enabled transition system $\langle \mathcal{S}, I, \wp(\mathrm{AP}), \twoheadrightarrow \rangle$ together with a state labeling $\langle \mathbb{B}_3, \lambda \rangle$ such that

$$\lambda(\nabla \mathrm{After}(\sigma)) = [\sigma \vDash \varphi]^3_{\mathrm{LTL}}$$

for all words $\sigma \in \wp(\mathrm{AP})^\star$. As a monitor is deterministic and input-enabled, the set $\mathrm{After}(\sigma)$ is guaranteed to be a singleton set, i.e., contain exactly one state for each finite word σ. Thus, ∇ is used to denote this state. The outcome of the monitor is then obtained by taking the label of this state.

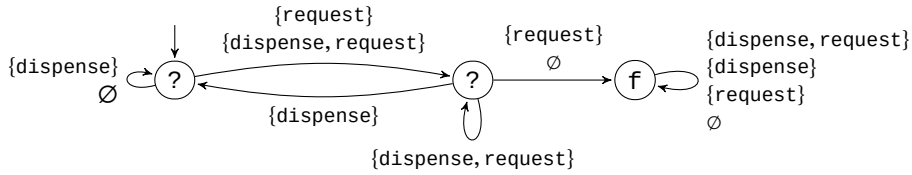


Figure 2.6: LTL_3 monitor for the LTL property defined in Example 2.6.

Example 2.6 Recall the example of the coffee machine (Example 2.1). Figure 2.6 depicts an LTL_3 monitor over the set $\mathrm{AP} = \{\texttt{dispense}, \texttt{request}\}$ for the property that every

request for a coffee is met in the next step, i.e., $\square(\texttt{request} \rightarrow \bigcirc\texttt{dispense})$. Note that we can never be sure that this property is satisfied because it may always be violated by a request in the unknown future that is not met. Hence, there is no state labeled with `t`. On the other hand, we can be sure that the property has been violated when we observe an instance of `request` that is not followed by `dispense` in the next step. The monitor detects such occurrences and transitions to the state labeled with `f` in response, indicating a violation of the property. The other states are labeled with `?` as the monitor cannot be sure that the property is satisfied but has not yet detected a violation either. While the coffee machine model (Figure 2.1) does not satisfy the property, the monitor enables us to detect when the coffee machine becomes faulty and does not dispense a coffee upon request.

Beyond LTL_3 Monitoring. LTL_3 monitoring falls short for properties like

$$\square(\texttt{request} \rightarrow \Diamond\texttt{response}) \tag{2.4}$$

i.e., that every request is eventually met with a response. The fundamental reason is that those properties do neither have good nor bad prefixes. After a request, it is always possible, that a response will eventually follow, i.e., the property is never violated in finite time. At the same time, it is always possible that a future request will not be followed by a response, i.e., the property is never satisfied in finite time. To deal with such properties, Bauer, Leucker, and Schallhart introduce *RV-LTL monitoring* using the truth domain $\mathbb{B}_4 = \{\texttt{t}, \texttt{t}^\text{p}, \texttt{f}^\text{p}, \texttt{f}\}$ instead of $\mathbb{B}_3$, where $\texttt{t}^\text{p}$ denotes *possibly true* and $\texttt{f}^\text{p}$ denotes *possibly false* [BLS07]. An RV-LTL monitor for the property (2.4) will produce $\texttt{f}^\text{p}$ iff there is a pending request that has not been met yet and $\texttt{t}^\text{p}$ otherwise. For a detailed discussion of different truth domains and the property classes they cover, we refer to Falcone, Fernandez, and Mounier [FFM12].

2.5.2 Stream-Based Monitoring with Lola

Another popular approach to runtime verification is stream-based monitoring. In the literature, a variety of different stream-based monitoring techniques have been proposed [DAn+05; Con+18; Fay+19; GS18]. As a paradigmatic example, we focus on the stream-based specification language Lola which has first been introduced by D'Angelo et al. [DAn+05]. In later work, Lola has been extended with parametrized streams (Lola 2.0) [Fay+16] and for real-time systems (RTLola) [Fay+19; Bau+20]. For the purposes of this thesis, we introduce the original variant of Lola.

The fundamental idea of stream-based monitoring is to compute *output streams* from *input streams*, where a stream is a sequence of values of some type. For instance, a stream may be a sequence $\sigma \in \mathbb{Z}^\star$ of integers.

Lola's Syntax. Let $\mathcal{T}$ be a set of *data types* and $\mathbb{S}$ be a set of *typed stream variables*, i.e., each $s \in \mathbb{S}$ has an associated type $T_s \in \mathcal{T}$. In the following, we treat types as sets of values that inhabit it. A *Lola specification* over $\mathcal{T}$ and $\mathbb{S}$ is a partial function $\mathsf{L} : \mathbb{S} \rightharpoonup \mathbb{E}$ assigning stream variables to *typed stream expressions*. The set of typed stream expressions $\mathbb{E}$ is defined inductively as follows:

(i) Constants and stream variables of type T are expressions of type T.
(ii) Let $g : T_1 \times \cdots \times T_k \to T$ be a k-ary operator and $\eta_1, \dots, \eta_k$ be expressions of type $T_1, \dots, T_k$, then $g(\eta_1, \dots, \eta_k)$ is an expression of type T.
(iii) Let η be a Boolean expression and η_1 and η_2 be expressions of some type T, then $\text{ite}(\eta, \eta_1, \eta_2)$ is an expression of type T.
(iv) Let s be a stream variable of type T, c be a constant of type T, and $z \in \mathbb{Z}$, then $s[z, c]$ is a stream expression of type T.

We slightly deviate from the original definition for notational convenience. In particular, we do not allow expressions of the form $\eta[z, c]$ where η is an arbitrary stream expression. It has been shown that those can be rewritten to $s'[z, c]$ by introducing an additional stream variable s' such that $\mathsf{L}(s') = \eta$ [DAn+05].

A Lola specification L defines stream expressions for the set $\text{Dom}(\mathsf{L}) \subseteq \mathbb{S}$ of stream variables. Those variables are coined *dependent* while the remaining variables are coined *independent*. The idea is that the stream expressions constrain the values the dependent variables can have at certain points in time.

Lola's Semantics. The semantics of Lola is defined in terms of *evaluation models*. An evaluation model μ of length N assigns a sequence $\mu(s) = (\mu_i(s))_{i=1}^{N}$ of values of type T_s, i.e., a stream of type T_s, to each stream variable $s \in \mathbb{S}$. Note that the individual streams all have the same length N. Given an evaluation model μ of length N we inductively define an evaluation function $[\![\cdot]\!]_i$ for evaluating stream expressions at certain points $1 \leq i \leq N$ in time as follows:

$$[\![c]\!]_i = c \qquad [\![s]\!]_i = \mu_i(s) \qquad [\![g(\eta_1, \dots, \eta_k)]\!]_i = g([\![\eta_1]\!]_i, \dots, [\![\eta_k]\!]_i)$$

$$[\![\text{ite}(\eta, \eta_1, \eta_2)]\!]_i = \begin{cases} [\![\eta_1]\!]_i & \text{if } [\![\eta]\!]_i = \mathsf{t} \\ [\![\eta_2]\!]_i & \text{otherwise} \end{cases}$$

$$[\![s[z, c]]\!]_i = \begin{cases} \mu_{i+z}(s) & \text{if } 1 \leq i + z \leq N \\ c & \text{otherwise} \end{cases}$$

An evaluation model μ *is consistent with* a Lola specification L, denoted by $\mu \vDash \mathsf{L}$, iff $[\![\mathsf{L}(s)]\!]_i = \mu_i(s)$ for all $s \in \text{Dom}(\mathsf{L})$ and $1 \leq i \leq N$. That is, the values of each dependent stream variable $s \in \text{Dom}(\mathsf{L})$ over time are exactly those values to which the respective expressions specified by L evaluate to.

By using an offset expression of the form $s[z, c]$, one can access the value of a stream variable s in the past as well as the future. The default value c in an offset

expression is used at time step i iff the time step $i + z$ lies outside of the streams provided by the evaluation model. Offset expressions are arguably the most innovative feature of Lola and responsible for its great expressive power. We refer to the original Lola paper for a detailed discussion [DAn+05].

Example 2.7 Imagine we would like to calculate the acceleration of a vehicle from its velocity. Assume that the velocity is measured every second in kilometers per hour ($\mathrm{km\,h^{-1}}$) and we want the acceleration to be computed in meters per square second ($\mathrm{m\,s^{-2}}$), taking into account the velocity measurement one step into the past and the measurement one step into the future. Formally that is, given a sequence $v = (v_i)_{i=1}^{N}$, we aim to compute a sequence $a = (a_i)_{i=1}^{N}$ such that

$$a_i = \frac{v_{i-1} + v_{i+1}}{2 \cdot 3.6}$$

for all $2 \leq i \leq N-1$. For the edge cases $i = 1$ and $i = N$, we may assume the velocity to be zero. Figure 2.7 exemplifies the computation for the different cases. The translation into a Lola specification L is straightforward with

$$\mathsf{L}(a) \coloneqq \frac{v[-1, 0] + v[+1, 0]}{2 \cdot 3.6}$$

and leaving v as in independent variable. An evaluation model consistent with L then provides a stream a according to the approach described above.

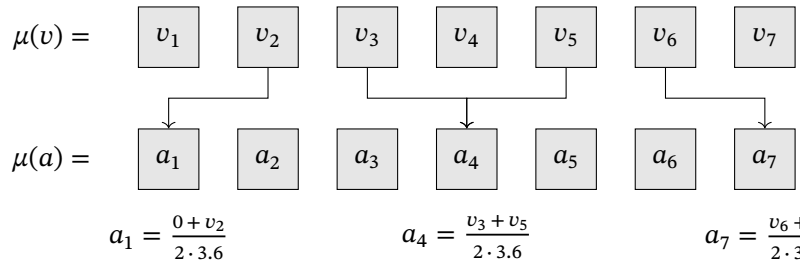


Figure 2.7: Computing accelerations from velocities as per Example 2.7.

Monitoring and Well-Formedness. A monitoring algorithm for a Lola specification L should incrementally compute values for the dependent stream variables based on incrementally provided values for the independent stream variables such that these values together form an evaluation model consistent with L. For this reason, we also refer to the streams for the independent variables as input streams and to the streams for the dependent variables as output streams.

The semantics of Lola is defined declaratively and does not directly induce a monitoring algorithm. In fact, it may even be impossible to compute output streams for a

given Lola specification and input streams, as there may not exist any consistent evaluation model or there may exist multiple such models leading to ambiguity [DAn+05]. For instance, imagine a dependent variable x for a Boolean stream that is defined as its own negation at each point in time:

$$\mathtt{L}(x) \;\coloneqq\; \neg x[0, \mathtt{true}]$$

While this is a syntactically valid Lola specification, no evaluation model is consistent with it. To rule out such specifications, D'Angelo et al. call a Lola specification *well-defined* iff for any set of appropriately typed input streams, all of the same length, it has exactly one consistent evaluation model [DAn+05]. Well-definedness ensures that monitoring is a well-defined algorithmic problem.

From a practical perspective, well-definedness is still difficult to check and deal with. Instead, the original paper introduces a purely syntactic criterion called *well-formedness* such that the following theorem holds [DAn+05, Theorem 1]:

Theorem 2.5.1 *If a specification is well-formed, then it is well-defined.*

Well-formedness is defined by means of a *dependency graph*:

Definition 2.5.1 *Let* $\mathtt{L}$ *be a Lola specification over the stream variables* $\mathbb{S}$*. The* dependency graph *for* $\mathtt{L}$ *is a directed and weighted multi-graph* $G = \langle \mathbb{S}, E \rangle$ *where* E *is the set of* edges*. An* edge *is a triple* $\langle s_x, s_y, z \rangle$ *where* $s_x, s_y \in \mathbb{S}$ *and* $z \in \mathbb{Z}$*. The set* E *of edges contains an edge* $\langle s_x, s_y, z \rangle$ *iff* $s_x \in \mathsf{Dom}(\mathtt{L})$ *and the expression* $\mathtt{L}(s_x)$ *contains an offset expression* $s_y[z, c]$ *for some constant* c*.*

Intuitively, the existence of an edge $\langle s_x, s_y, z \rangle$ in E records the fact that the stream for s_x depends on the stream for s_y with an offset of z. Now, if there exists a cycle whose weights z sum up to zero, i.e., a *zero-weight cycle*, then the value of some stream at a given time circularly depends on the very same value. Due to such cycles, there may then be multiple evaluation models for a given specification or none at all. That is why well-formedness forbids precisely such cycles [DAn+05, Definition 4].

Definition 2.5.2 *A Lola specification is* well-formed *iff its dependency graph does not contain any zero-weight cycle.*

By checking well-formedness of a specification, one makes sure that the monitoring problem is well-defined for it. It is this property of well-formedness that has to be checked in order to ensure that the monitoring algorithm presented in the original Lola paper can be used [DAn+05]. We refer to this paper for further details regarding the monitoring algorithm and Theorem 2.5.1. Note that the *zero-weight*

cycle problem, i.e., the problem of deciding whether a zero-weight cycle exists, is known to be NP-complete, which can be shown by a simple reduction from the NP-complete subset-sum problem [Bai+18, Theorem 3.12].

Efficient Online Monitoring. In general, monitoring for a given Lola specification may require access to the entire history of all streams. In case of online monitoring, this is usually undesirable, as it means that the monitor will run out of memory when the input streams become too long. To address this issue, D'Angelo et al. introduce the notion of *efficiently monitorable* specifications. If the dependency graph of a specification has no positive cycles, then the specification is efficiently monitorable, meaning that it is monitorable with a bounded amount of memory independent of the length of the input streams [DAn+05, Theorem 3].

In the context of online monitoring, the computation of output streams may also be delayed if their values depend on future values. For an example, recall Example 2.7. Here, the velocity one step into the future is required to compute the acceleration at the present point in time. Efficient monitorability also puts an upper bound on the delay with which streams may lag behind.

2.6 Model-Based Fault Diagnosis

Model-based fault diagnosis aims at detecting faults based on a formal system model and from the observable behavior of a system [e.g. Tri02; Sam+95; BCD05; Car+13; ALH06; TYG08]. For the purpose of this thesis, we recapitulate the seminal work on model-based fault diagnosis by Sampath et al. [Sam+95]. We refer to this work and the techniques it developed as *traditional model-based fault diagnosis*.

Traditional model-based fault diagnosis assumes a system to be modeled as a deterministic and finite TS $\mathfrak{S} = \langle S, I, \mathsf{Act}, \twoheadrightarrow \rangle$, whose actions are partitioned into a set of *observable* actions $\mathsf{OAct} \subseteq \mathsf{Act}$ and *unobservable* actions $\mathsf{UAct} \subseteq \mathsf{Act}$. The latter includes a set $\mathsf{Faults} \subseteq \mathsf{UAct}$ of *fault actions*, partitioned into *fault classes* $\mathcal{F} = \{f_1, \ldots, f_n\}$. The diagnosis techniques developed by Sampath et al. synthesize a *diagnoser* from such a system model. A diagnoser reads a sequence of observable actions and produces a *diagnosis* indicating which faults have occurred.

Assuming that faults may occur multiple times, formally, a diagnoser $\mathcal{D}$ is a deterministic TS $\langle \wp(S \times \wp(\mathcal{F})), I_{\mathcal{D}}, \mathsf{OAct}, \twoheadrightarrow_{\mathcal{D}} \rangle$. Each state $q \in \wp(S \times \wp(\mathcal{F}))$ of a diagnoser corresponds to a diagnosis $\mathrm{d}(q) \subseteq \wp(\mathcal{F})$ defined as follows:

$$\mathrm{d}(q) \;\coloneqq\; \{ F \mid \langle \cdot, F \rangle \in q \} \tag{2.5}$$

For a diagnosis $d \subseteq \wp(\mathcal{F})$, each $F \in d$ indicates a possibility that faults of the classes $f_i \in F$ occurred. Hence, a fault of class f_i *certainly occurred* iff $f_i \in F$ for all $F \in d$ and it *possibly occurred* iff $f_i \in F$ for some $F \in d$ [Sam+95, cf. Definition 6]. The diagnosis

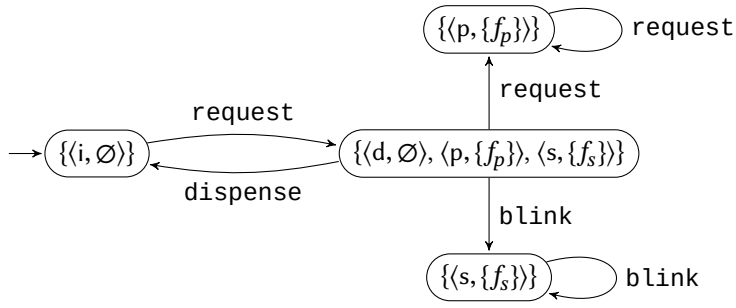


Figure 2.8: Diagnoser synthesized from the model of the coffee machine (Figure 2.1).

produced for a sequence $\omega \in \mathcal{L}(\mathfrak{S})|_{\text{OAct}}$ of observable actions, generated by running the system $\mathfrak{S}$, is then given by $\mathrm{d}(\nabla\mathrm{After}(\omega))$. Note that $\mathcal{D}$ is constructed such that $\mathcal{L}(\mathcal{D}) = \mathcal{L}(\mathfrak{S})|_{\text{OAct}}$. Hence, for each trace $\sigma \in \mathcal{L}(\mathfrak{S})$ of the diagnosed system, the diagnoser $\mathcal{D}$ produces a diagnosis $\mathrm{d}(\nabla\mathrm{After}(\sigma|_{\text{OAct}}))$ indicating which faults occurred, taking only observable actions into account. For further details, we refer to the work of Sampath et al. [Sam+95].

Example 2.8 Figure 2.8 depicts the diagnoser constructed from the model of the coffee machine (Figure 2.1). Here, `request`, `dispense`, `blink` $\in$ OAct are observable and `pump_fault`, `short_circuit` $\in$ Faults are fault actions. Further, each fault action forms a fault class f_p and f_s, respectively. In states $\{\langle \mathrm{p}, \{f_p\}\rangle\}$ and $\{\langle \mathrm{s}, \{f_s\}\rangle\}$ of the diagnoser, `pump_fault` and `short_circuit` certainly occurred, respectively, and in state $\{\langle \mathrm{i}, \emptyset\rangle\}$ no fault possibly occurred. In state $\{\langle \mathrm{d}, \emptyset\rangle, \langle \mathrm{p}, \{f_p\}\rangle, \langle \mathrm{s}, \{f_s\}\rangle\}$, both faults possibly occurred but no fault certainly occurred.

Continuous-Time Systems. Building upon the work by Sampath et al., Tripakis extended model-based fault diagnosis to the continuous-time setting [Tri02]. Here, a timed automaton serves as the model of the system and its possible faults. Like with traditional model-based diagnosis, the actions of the timed automaton are partitioned into observable, unobservable, and fault actions. Faults are then diagnosed based on timed words of observable actions, taking the timing of events into account. Bouyer, Chevalier, and D'Souza further improved the work by Tripakis. They developed techniques based on deterministic and event-recording timed automata [BCD05]. For further details, we refer to the respective original papers. Notably these existing techniques share the assumption that the timing of events can be observed precisely, an assumption that we relax with the contributions of this thesis.

Other Model-Based Techniques. The term "model-based diagnosis" is used in various ways in the literature. Kleer and Williams [KW87] and Reiter [Rei87] in-

dependently developed an approach which can also be classified as model-based diagnosis [cf. BLS06a]. In contrast to the approach by Sampath et al., their approach does not rest on an explicit modeling of individual faults and their influence on observable behavior. Instead, they merely require a model of the nominal behavior of the different components of a system from which they then identify misbehaving components. Here, a model can also be a specification in terms of pre- and post-conditions of component behavior. A component is considered misbehaving iff the pre-conditions on its input are satisfied but its post-conditions are violated. Building upon this idea Bauer, Leucker, and Schallhart propose to use runtime verification techniques to detect misbehaving individual components and then extrapolate causes in terms of minimal sets of misbehaving components responsible for the malfunctioning of a system [BLS06a]. Again other works consider nominal system models in terms of differential equations or other descriptions of continuous system behavior, also commonly referred to as "models" and "model-based diagnosis" [Fra90; Ise05]. Furthermore, there is an entire class of techniques that build machine-learned models from historic data of system behavior in order to identify and predict faults [Car+19]. Note that throughout this thesis and unless stated otherwise, "model" means an operational model in terms of some kind of transition system.

2.7 Fault Trees

Fault trees are a popular tool to model, understand, and analyze how faults of individual components may lead to failures of an entire system. Their inception goes back to the early 1960s. Watson and Mearns pioneered fault trees for the launch control system of the Minuteman intercontinental ballistic missiles developed by the United States [Eri99]. Since then, fault trees have been adopted for a wide range of safety-critical applications from nuclear power plants to aviation to medial applications. They became a key tool to asses risks of failure as part of reliability analyses. We refer to Ruijters and Stoelinga for a comprehensive overview [RS15].

As the name "fault tree" suggests, a fault tree is usually a tree. The leaves of a fault tree correspond to *basic fault events* of individual components, and inner nodes correspond to Boolean *gates* describing how higher-level faults result from lower-level faults. Figure 2.9 shows the standard graphical representation of disjunction (or) and conjunction (and) gates. Note that despite the name "fault tree," definitions found in the literature often allow sharing of subtrees between nodes. Thus, a fault tree can also be a directed acyclic graph [RS15, cf. Section 2.1.2].

Example 2.9 Figure 2.10 shows a fault tree for the coffee machine (Example 2.1). Here, the pump fails iff its inlet becomes clogged *or* its shaft breaks. This fact is modeled by an *or gate* indicating that a fault of any sub fault is sufficient for the pump to fail. In contrast, a short circuit occurs iff there are exposed wires *and* there is fluid leakage.

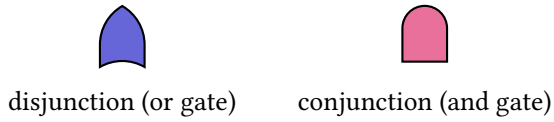


Figure 2.9: Standard disjunction and conjunction fault tree gates [cf. RS15].

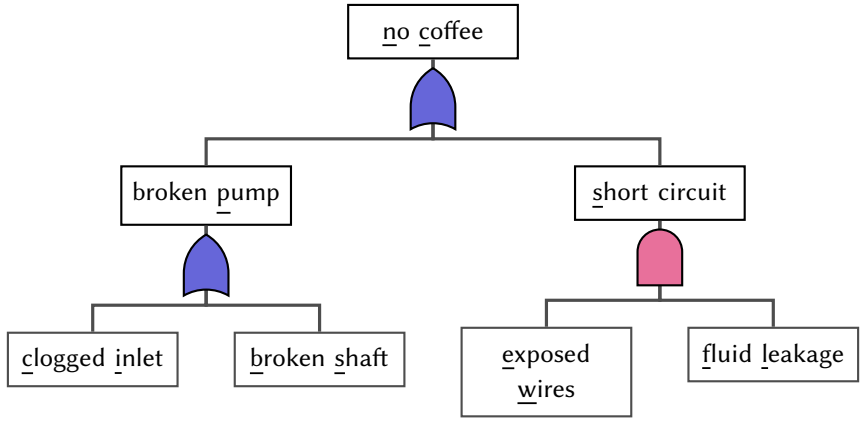


Figure 2.10: A fault tree for the coffee machine (Example 2.1).

This fact is modeled by an *and gate* indicating that a short circuit requires all sub faults to occur. As for feature diagrams, we underline the parts of the fault names that we use in formal definitions. The set of basic events is given by:

$$\{\mathsf{ci}, \mathsf{bs}, \mathsf{ew}, \mathsf{fl}\}$$

While the graphical representation of fault trees as in Figure 2.10 is useful for humans, from a mathematical perspective, fault trees can be represented as negation-free Boolean expressions over the set of basic fault events where we ascribe additional names to subexpressions. For instance, the entire fault tree shown in Figure 2.10 corresponds to the following Boolean expression:

$$\overbrace{\underbrace{(\mathsf{ci} \vee \mathsf{bs})}_{\text{broken pump}} \vee \underbrace{(\mathsf{ew} \wedge \mathsf{fl})}_{\text{short circuit}}}^{\text{no coffee}}$$

The broken pump subexpression corresponds to the event that the pump broke. The short circuit subexpression corresponds to the event that there was a short circuit. If any of these events occurs, then there will be no coffee. For this thesis, we always directly work with the Boolean expressions corresponding to fault trees.

Chapter 3

Theoretical Framework

In this chapter, we establish an overarching theoretical framework that serves as the cornerstone for the subsequent contributions of this thesis. Centered around the concept of verdictors, as previously discussed in Chapter 1 (recall Figure 1.1), this framework fulfills three pivotal roles within the scope of this thesis:

(FT1) The framework offers rigorous formalizations of core concepts and criteria with respect to which we then prove the remaining contributions correct. In particular, following the model-based methodology layed out earlier (cf. Chapter 1), we establish formal criteria for what it takes for a verdictor to produce accurate verdicts with respect to a given system model and based on observations subject to observational imperfections.

(FT2) The framework introduces *verdict transition systems* (VTSs) as a representation of verdictors, which then also serves as the target representation for the generic and modular discrete-time verdictor synthesis algorithms developed in the subsequent chapter (cf. Contribution TT).

(FT3) The framework offers a unifying foundation for automata-based runtime verification and model-based fault diagnosis. It elucidates the connection between those traditional techniques and the contributions of this thesis. Furthermore, it enables the broad range of applications of the algorithms developed in the subsequent chapters of this thesis.

The theoretical framework constitutes Contribution FT of this thesis.

Overview. The formalizations introduced in this chapter include *verdict domains*, verdict transition systems, and *observation models*. Together they capture all concepts appearing in Figure 1.1 and form a complete framework for verdictors.

Verdict domains define the set of possible verdicts a verdictor can produce, along with a partial order that ranks verdicts based on their *specificity*. For instance, in the case of runtime verification, it may sometimes be *unknown* whether a property is satisfied or not (cf. Section 2.5.1). This verdict is *less specific* than a verdict indicating a clear satisfaction or violation of the property. In particular, when faced with observational imperfections, one often needs to make some concessions with respect to the specificity of verdicts: We want verdicts to correctly reflect the state of the system at all times, however, it is fine if a verdict is less specific in cases where a more specific verdict simply cannot be justified, e.g., due to observational imperfections. We demonstrate that both traditional LTL runtime verification truth domains (recall Section 2.5.1) and diagnoses from model-based fault diagnosers (recall Section 2.6) constitute verdict domains with a respective specificity ranking.

VTSs formally represent how verdictors obtain and refine verdicts over time as new observations are made. Observation models represent how system runs generate sequences of observations that are fed into a verdictor. As a ground truth for verdicts, we further introduce *verdict oracles* that assign a verdict to each run of a system. Based on these concepts, we then specify what it takes for a verdictor to produce accurate verdicts with respect to a given system and observation model.

Relevant Publications. Verdict transition systems, including some of the key results, have been introduced by the author of this thesis in:

> [KDH24]: Maximilian A. Köhl, Clemens Dubslaff, and Holger Hermanns. "Configuration Monitor Synthesis". In: *Automated Technology for Verification and Analysis, ATVA 2024.*

The broader framework presented here has been specifically developed to show how the various contributions of the first author integrate under a coherent theoretical umbrella. It hence extends and generalizes the published works.

Chapter Structure. Section 3.1 introduces verdict domains, Section 3.2 introduces verdict transition systems, and Section 3.3 introduces observation models. Section 3.4 establishes the formal criteria for what it takes for a verdictor to produce accurate verdicts with respect to a given system and observation model. Section 3.5 discusses how existing work in the spectrum of automata-based runtime verification and model-based fault diagnosis naturally fits into the theoretical framework. Finally, Section 3.6 concludes this chapter.

3.1 Verdict Domains

A verdictor processes observations and produces *verdicts* (recall Figure 1.1). As a fundamental assumption made throughout this thesis, we assume that verdicts can

be ordered based on their *specificity*. As a simple example, recall the three-valued truth domain $\mathbb{B}_3 = \{\mathtt{t}, \mathtt{?}, \mathtt{f}\}$ of LTL_3 runtime verification (Section 2.5.1). Here, `t` and `f` are *more specific* than `?`, since they represent definite truth values while `?` does not, and `t` and `f` are incomparable as neither is more specific than the other. As a generalization, we assume that verdicts form a *verdict domain*.

Definition 3.1.1 *A* verdict domain *is a complete join-semilattice* $\langle \mathcal{V}, \sqsubseteq \rangle$.

We call $\sqsubseteq$ the *specificity order*. A verdict v_1 *is more specific than* a verdict v_2 iff $v_1 \sqsubset v_2$, where $\sqsubset$ is the irreflexive kernel of $\sqsubseteq$. Given a non-empty set $V \subseteq \mathcal{V}$ of verdicts, their join $\bigsqcup V$ is the *most specific* verdict subsuming all verdicts in V.

Remark. The fact that the join exists for any non-empty set $V \subseteq \mathcal{V}$ of verdicts will play a crucial role throughout this thesis, in particular, when we consider observational imperfections. The upshot is that, in cases where we cannot discern certain runs of a system due to observational imperfections, we are interested in the most specific verdict that is in line with all the plausible runs.

Let us now have a look at different instances of verdict domains extrapolated from LTL_3 runtime verification and traditional model-based fault diagnosis, as well as for the novel application of configuration monitoring.

LTL_3 Runtime Verification Verdicts. As discussed above, the three-valued truth domain $\mathbb{B}_3 = \{\mathtt{t}, \mathtt{?}, \mathtt{f}\}$ of LTL_3 runtime verification forms a verdict domain. This domain has its origins in Kleene's three-valued logic [Kle38]. Figure 3.1 shows two partial orders over $\mathbb{B}_3$. Figure 3.1a shows the traditional *truth order*. Here, $\sqcap$ and $\sqcup$ correspond to conjunction and disjunction, respectively, as defined by Kleene. In contrast, Figure 3.1b shows the specificity order. While the truth order induces a complete lattice, the specificity order only induces a complete join-semilattice. Note that traditional work on runtime verification always considers the truth ordering [FFM12], which does not capture the specificity of the verdicts.

(a) Three-valued truth domain. (b) Three-valued verdict domain.

Figure 3.1: Partial orders on the truth domain of LTL_3 runtime verification.

Diagnoses as Verdicts. Recall that a diagnosis produced by traditional model-based fault diagnosis techniques is a set $d \subseteq \wp(\mathcal{F})$ of sets of fault classes. Also recall that a fault of class f_i *certainly occurred* iff $f_i \in F$ for all $F \in d$ and it *possibly occurred* iff $f_i \in F$ for some $F \in d$ (cf. Section 2.6). These considerations lead to an inherent notion of specificity: A diagnosis d is more specific than another diagnosis d' iff it considers less sets of fault classes possible, i.e., iff $d \subsetneq d'$. For instance, in case of the coffee machine (recall Example 2.8), the diagnosis $d = \{\{f_p\}\}$ is more specific than $d' = \{\emptyset, \{f_p\}, \{f_s\}\}$ as $d \subsetneq d'$. The diagnosis d indicates that the pump certainly is broken whereas the diagnosis d' indicates that it is possible that the machine functions normally ($\emptyset$), has a broken pump ($\{f_p\}$), or has a short circuit ($\{f_s\}$). Formally, we obtain the following definition for diagnosis verdicts:

Definition 3.1.2 *Given a set $\mathcal{F}$ of fault classes, the* traditional diagnosis verdict domain *is $\langle \wp(\wp(\mathcal{F})), \subseteq \rangle$. Verdicts $d \in \wp(\wp(\mathcal{F}))$ are diagnoses.*

Note that for $N = |\mathcal{F}|$ fault classes, there are 2^{2^N} possible diagnoses. For the coffee machine example, this equates to 16 possibilities.

Simplified Diagnosis Verdicts. For illustrative purposes, we also introduce a simplified variant of the verdict domain of traditional diagnosis as per Definition 3.1.2. This simplified version can only indicate fault classes of which faults certainly occurred, giving rise to the following verdict domain:

Definition 3.1.3 *Given a set $\mathcal{F}$ of fault classes, the* simplified diagnosis verdict domain *is $\langle \wp(\mathcal{F}), \supseteq \rangle$. Verdicts $v \in \wp(\mathcal{F})$ are sets of fault classes of which faults certainly occurred.*

Note that the specificity order here is $\supseteq$ and not $\subseteq$. This is the case as the fault classes $f \in v$ within a verdict $v \in \wp(\mathcal{F})$ now indicate necessities, since they indicate fault classes of which faults certainly occurred. Therefore, a verdict v_1 is more specific than a verdict v_2, denoted by $v_1 \sqsubset v_2$, iff it indicates that more faults certainly occurred, i.e., iff $v_1 \supsetneq v_2$. In the case of traditional diagnosis, verdicts are sets of possibilities instead, hence, there the order is $\subseteq$ (cf. Definition 3.1.2).

As an example, the simplified diagnosis verdict domain for the coffee machine (cf. Example 2.8) is $\langle \wp(\{f_p, f_s\}), \supseteq \rangle$. Here, $\emptyset$ is the least specific verdict and $\{f_p, f_s\}$ is the most specific verdict. Figure 3.2 depicts this verdict domain. For instance, the verdict $\{f_p, f_s\}$ according to which the pump certainly failed and a short circuit also certainly occurred (not actually possible in the model) is more specific than the verdict $\{f_p\}$ according to which only the pump certainly failed.

Each traditional diagnosis according to Definition 3.1.2 can be simplified by just intersecting all the sets within a diagnosis thereby reducing it to the set of fault classes of which faults certainly occurred.

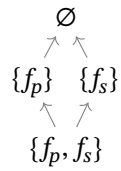


Figure 3.2: Simplified diagnosis verdict domain for the coffee machine example.

Configuration Verdicts. One of the novel applications enabled by the contributions of this thesis is configuration monitoring (cf. Chapter 1). Recall that a system can be configured towards a set Conf of valid configurations (cf. Section 2.4). A verdictor for configuration monitoring produces *configuration verdicts*, indicating which configurations the system may possibly be in. A configuration verdict thus is a non-empty set $C \in \wp(\mathsf{Conf}) \setminus \{\varnothing\}$, as the system must be in some configuration. Intuitively, such a set $C \in \wp(\mathsf{Conf}) \setminus \{\varnothing\}$ is more specific than another $C' \in \wp(\mathsf{Conf}) \setminus \{\varnothing\}$ iff it considers less configurations possible, i.e., iff $C \subseteq C'$. Formally, the verdict domain of configuration monitoring is defined as follows:

Definition 3.1.4 *Given a set* Conf *of valid configurations, the* configuration verdict domain *is* $\langle \wp(\mathsf{Conf}) \setminus \{\varnothing\}, \subseteq \rangle$.

As an example, Figure 3.3 depicts the verdict domain for configuration monitoring of the email system introduced earlier (Example 2.5).

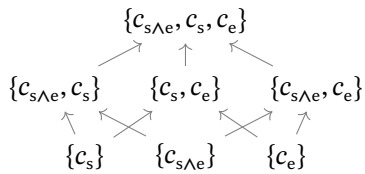


Figure 3.3: Verdict domain for configuration monitoring of the email system.

Remark. The lattice $\langle \wp(\mathsf{Conf}), \subseteq \rangle$ is commonly used in the context of model checking of configurable systems [GLS08; Cla+10; Bee+16]. Here, elements of the lattice represent sets of system configurations that may satisfy/violate a given specification. In contrast to the verdict domain considered here, those sets can be empty iff no configuration satisfies/violates the specification.

Answers to Operational Questions. The presented verdict domain definitions demonstrate the versatility of verdict domains. Verdict domains can be used to capture

the truth domain of LTL_3 runtime verification and diagnoses from traditional model-based diagnosis techniques. At the same time, they generalize and unify them under a coherent definition, which can also be used for novel applications, as demonstrated by configuration verdicts. As such, verdicts can serve as answers to the pressing operational questions exemplified in Chapter 1. The contributions of this thesis will exploit the complete join-semilattice properties of verdict domains.

3.2 Verdict Transition Systems

With verdict domains in place, we now introduce *verdict transition systems* (VTS) as a formal representation of verdictors. VTSs capture how *verdicts* are obtained and evolve over time as new observations are made.

Definition 3.2.1 *For a set* Obs *of* observables*, a* verdict transition system $\mathfrak{V}$ *is a tuple* $\langle Q, J, \mathrm{Obs}, \rightarrowtriangle, \mathcal{V}, \sqsubseteq, \nu \rangle$ *where*
- $\langle Q, J, \mathrm{Obs}, \rightarrowtriangle \rangle$ *is a TS,*
- $\langle \mathcal{V}, \sqsubseteq \rangle$ *is a verdict domain, and*
- $\langle \mathcal{V}, \nu \rangle$ *is a state labeling.*

The state labeling $\langle \mathcal{V}, \nu \rangle$ assigns a verdict $\nu(q)$ to each state $q \in Q$. We also refer to the tuple $\langle \mathcal{V}, \sqsubseteq, \nu \rangle$ as a *verdict extension* of the underlying TS $\langle Q, J, \mathrm{Obs}, \rightarrowtriangle \rangle$.

Modeling Verdictors. Intuitively, a verdictor has an internal state, modeled by the elements of the set Q. In response to new observations, it updates its internal state, modeled by $\rightarrowtriangle$, and then *produces* a verdict based on this state, modeled by the state-labeling function ν. Note that any practical verdictor can be modeled this way as any Turing machine can be molded into a VTS (cf. Section 2.2).

In practice, it is desirable that verdictors are deterministic. When feeding an *observation sequence* $\omega \in \mathcal{L}(\mathfrak{V})$ into a verdictor, we want to obtain a definite and uniquely defined verdict. In case a VTS is indeed deterministic, the verdict *produced* for some observation sequence $\omega \in \mathcal{L}(\mathfrak{V})$ is $\nu(\nabla\mathrm{After}(\omega))$.[8] Formally, an *observation* is an occurrence of an observable on an observation sequence, i.e., an observation is a tuple $\theta = \langle i, \mathsf{o} \rangle \in \omega$ of some observation sequence ω. Recall that sequences are a special form of partial functions which are sets of pairs $\langle i, \mathsf{o} \rangle$ where i is the index at which the symbol o occurs within the sequence (see Section 2.1).

The careful reader may have already noticed that this definition is very close to how LTL runtime monitors (Section 2.5.1) and traditional diagnosers (Section 2.6) are defined. Given that we already established that their outputs form verdict domains

[8] Recall that ∇X denotes the element x of a singleton set $X = \{x\}$.

(see examples in Section 3.1), it is easy to see that VTSs indeed generalize LTL runtime monitors and traditional diagnosers under a unified definition.

Example 3.1 Figure 3.4 shows an example of a VTS as it may be synthesized from the model of the coffee machine (recall Figure 2.1) with the generic algorithms presented in this thesis. It is a simplified version of a diagnoser (cf. Example 2.8) that observes non-fault actions of the system model and indicates fault classes of which faults certainly occurred, i.e., the verdict domain in this case is $\langle \wp(\{f_p, f_s\}), \supseteq \rangle$ (recall Definition 3.1.3 and the discussion around it). For instance, after observing two requests in direct succession we can be certain that the pump failed leading to the verdict $\{f_p\}$. We will later introduce the formal tools to characterize the relation between this VTS and the system model precisely.

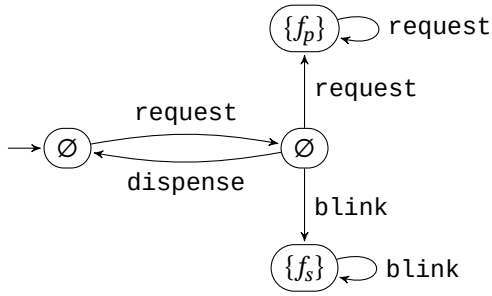


Figure 3.4: VTS for diagnosing certain faults of the coffee machine.

Non-Determinism. According to Definition 3.2.1, VTSs may be non-deterministic. Non-deterministic VTSs provide more modeling freedom for verdictors. We will also exploit this freedom later when synthesizing VTSs from system models. As a VTS $\mathfrak{V}$ may be non-deterministic, an observation sequence $\omega \in \mathcal{L}(\mathfrak{V})$, may lead to multiple states with different verdicts. To account for these possibilities, we utilize the fact that the verdict domain is a complete join-semilattice to determine the most specific verdict subsuming all non-deterministically possible verdicts:

Definition 3.2.2 *For each trace $\omega \in \mathcal{L}(\mathfrak{V})$, $\mathfrak{V}$ produces a verdict $\nu(\omega)$:*

$$\nu(\omega) := \bigsqcup \{ \nu(q) \mid q \in \mathsf{After}(\omega) \} \tag{3.1}$$

Recall that $\mathsf{After}(\omega)$ is the set of states reached after ω, which is guaranteed to be non-empty for every trace $\omega \in \mathcal{L}(\mathfrak{V})$. Further, recall that the join $\bigsqcup V$ of a non-empty set of verdicts V corresponds to the most specific verdict subsuming all verdicts in V. Hence, $\nu(\omega)$ is the most specific verdict subsuming all the non-deterministic

possibilities. Again, take $\mathbb{B}_3$ as an example, if both `t` and `f` are non-deterministically possible, $\mathfrak{B}$ produces `?` indicating an unknown truth value.

Relation to Lattice Automata. Recall Section 2.2.3 for the definition of lattice automata. From the definition of verdict domains and VTSs, it is obvious that VTSs share similarities with lattice automata. VTSs are strictly more general than lattice automata in the sense of expressible L-languages.

Lemma 3.2.1 *For a bounded lattice L, any L-language expressible by a lattice automaton can also be expressed by a corresponding VTS.*

Proof. As established by Kupferman and Lustig, every lattice automaton can be transformed into a simple one [KL07, Theorem 6]. Recall that for simple lattice automata, $Q_0(q) = \top$ marks initial states and $\delta(q, a, q') = \top$ marks the existence of transitions (see Section 2.2.3). Hence, any simple lattice automaton $\langle L, \Sigma, Q, Q_0, \delta, F\rangle$ is trivially transformed into a VTS with $\mathcal{Q} = Q$, $\mathrm{Obs} = \Sigma$, $J = \{\, q \mid Q_0(q) = \top \,\}$, $\rightarrow = \{\, t \mid \delta(t) = \top \,\}$, $\mathcal{V} = L$, and $\upsilon = F$. This transformation retains the L-language $\mathcal{L} : \Sigma^\star \to L$ of the original lattice automaton. For each $\sigma \in \Sigma^\star$, we have:

$$\mathcal{L}(\sigma) \;=\; \begin{cases} \nu(\sigma) & \text{if } \sigma \in \mathcal{L}(\mathfrak{B}) \\ \bot & \text{otherwise} \end{cases}$$

Here, $\bot$ is the bottom element of the bounded lattice L (cf. Section 2.2.3). While the reverse transformation also applies if $\mathcal{V}$ is a bounded lattice and the VTS is finite, VTSs are more general as they only require the verdict domain $\mathcal{V}$ to be a join-semilattice, may be non-deterministic, and may be infinite. □

Relation to Runtime Monitoring. Leucker and Schallhart characterize a runtime monitor as "a device that reads a finite trace and produces a certain *verdict*" [LS09, p. 294]. They go on to note that "a verdict is typically a value from some truth domain [and] a truth domain is a lattice with a unique top element *true* and a unique bottom element *false*." VTSs are similar to this idea, however, they use a specificity ordering instead of a classical truth ordering (cf. Section 3.1), which is the crux enabling us to deal with observational imperfections. Again, the order of a truth domain would follow a traditional truth order, where *true* and *false* are the bounds, however, a specificity ordering will diverge from that (recall Figure 3.1).

Relation to Moore Machines. As pointed out in Section 2.2.1, state-labeled transition systems are a potentially infinite and non-deterministic generalization of Moore machines [cf. Moo56]. At its core, a VTS is a transition system with verdict-labeled states. If a VTS is finite and deterministic, then it is indeed a Moore machine outputting a verdict when fed an observation sequence as input.

3.2.1 Monotonicity, Refinement, and Equivalence

Next, we introduce VTS *monotonicity*, *refinement*, and *equivalence*. All three are important foundational notions for VTSs and verdictors.

Monotonicity. An interesting class of VTSs is the class of *monotonic* VTSs. Intuitively, a VTSs is monotonic iff verdicts become only more specific as new observations are made. Let us first define what it takes for a state to be monotonic:

Definition 3.2.3 *A state $q \in Q$ of a VTS $\langle Q, J, \mathrm{Obs}, \rightarrowtriangle, \mathcal{V}, \sqsubseteq, \nu \rangle$ is called* monotonic *iff $\nu(q') \sqsubseteq \nu(q)$ for all its successors $q' \in \mathrm{Post}(\{q\}, \mathrm{Obs})$.*

That is, the verdicts of the state's successors are at least as specific as the verdict of the state itself. This definition is straightforwardly lifted to VTSs:

Definition 3.2.4 *A VTS is monotonic iff all its states are monotonic.*

If a VTS is not monotonic, then it is called *non-monotonic*.

Example 3.2 The simplified diagnoser for the coffee machine discussed in Example 3.1 is monotonic. Recall that it identifies faults which certainly occurred. Once a fault certainly occurred, it cannot be made undone. Hence, any VTS identifying faults which certainly occurred is monotonic. If we look at traditional diagnoser constructed from the coffee machine (see Example 2.8) and its verdict domain (recall Definition 3.1.2), we see that the traditional diagnoser is non-monotonic. After requesting a coffee in the initial state of the diagnoser, the verdict becomes less specific, as it is now possible that the machine functions normally, that the pump broke, or that there has been a short circuit.

Refinement. Typically, one is interested in verdicts that are as specific as possible, i.e., most specific without sacrificing correctness. For instance, in case of Example 3.1, one wants the verdict to contain all faults that certainly occurred but no faults that did not certainly occur. To formally capture that a VTS yields more specific verdicts than some other VTS, we introduce a refinement relation.

Definition 3.2.5 *Let $\mathfrak{V}_1$ and $\mathfrak{V}_2$ be two VTSs over some verdict domain $\langle \mathcal{V}, \sqsubseteq \rangle$. We say that $\mathfrak{V}_1$* refines *$\mathfrak{V}_2$, denoted by $\mathfrak{V}_1 \preceq \mathfrak{V}_2$, iff*

- *(RE1) $\mathfrak{V}_1$ and $\mathfrak{V}_2$ have the same language, i.e., $\mathcal{L}(\mathfrak{V}_1) = \mathcal{L}(\mathfrak{V}_2)$, and*
- *(RE2) $\mathfrak{V}_1$ produces at least as specific verdicts as $\mathfrak{V}_2$, i.e., $\nu_1(\omega) \sqsubseteq \nu_2(\omega)$ for all observation sequences ω of their language, i.e., for all $\omega \in \mathcal{L}(\mathfrak{V}_1)$.*

Equivalence. If two VTSs produce exactly the same verdict for each observation sequence, then we may favor one over the other for other properties. For instance, one may be smaller than the other. To formally capture that two VTSs are equivalent in that sense, we introduce an equivalence relation.

Definition 3.2.6 *Let $\mathfrak{V}_1$ and $\mathfrak{V}_2$ be two VTSs over the same verdict domain. We say that $\mathfrak{V}_1$ and $\mathfrak{V}_2$ are* verdict-equivalent, *denoted by $\mathfrak{V}_1 \equiv \mathfrak{V}_2$ iff they refine each other, i.e., iff $\mathfrak{V}_1 \preceq \mathfrak{V}_2$ and $\mathfrak{V}_2 \preceq \mathfrak{V}_1$.*

It is easy to see that $\equiv$ is indeed an equivalence relation, i.e., that it is reflexive, transitive, and symmetric. Furthermore, it is the coarsest equivalence relation such that equivalent VTSs produce the same verdicts for all observation sequences:

Lemma 3.2.2 *Given two VTSs, $\mathfrak{V}_1$ and $\mathfrak{V}_2$, over some verdict domain $\langle \mathcal{V}, \sqsubseteq \rangle$ and with $\mathcal{L}(\mathfrak{V}_1) = \mathcal{L}(\mathfrak{V}_2)$, $\mathfrak{V}_1 \equiv \mathfrak{V}_2$ iff $\nu_1(\omega) = \nu_2(\omega)$ for all $\omega \in \mathcal{L}(\mathfrak{V}_1)$.*

Proof. As $\mathcal{L}(\mathfrak{V}_1) = \mathcal{L}(\mathfrak{V}_2)$, (RE1) is mutually satisfied. If $\mathfrak{V}_1 \equiv \mathfrak{V}_2$, then mutually (RE2) and, hence, $\nu_1(\omega) = \nu_2(\omega)$ for all $\omega \in \mathcal{L}(\mathfrak{V}_1)$ due to antisymmetry of $\sqsubseteq$. If $\nu_1(\omega) = \nu_2(\omega)$, then mutually (RE2) due to reflexivity of $\sqsubseteq$. □

Remark. For a given language $\mathcal{L} \subseteq \mathrm{Obs}^\star$ and verdict domain $\langle \mathcal{V}, \sqsubseteq \rangle$, refinement induces a complete join-semilattice among those equivalence classes of VTSs that accept $\mathcal{L}$. The top element of this join-semilattice is the class of VTSs that accept $\mathcal{L}$ and produce the least-specific verdict of the verdict domain. The join of this join-semilattice combines VTSs towards a VTS that produces most specific verdicts that are at most as specific as the verdicts produced by the individual VTSs. For finite VTSs, the join can be computed by a straightforward product construction and combining the verdicts of product states with the join of the verdict domain.

3.2.2 Determinization and Minimization

Recall that VTSs will serve as a target representation for verdictor synthesis (FT2). For this, it is desirable that synthesized VTSs are deterministic and *minimal*. Deterministic VTSs are straightforwardly implemented by an efficient lookup table and storing a single state which is updated in response to new observations. If a VTS is minimal, then the size of the individual states as well as the lookup table is minimized, which is especially important when implementing VTSs on space-constrained devices such as embedded devices or FPGAs [Bod+04; Zha+22].

In the following, we establish results for the determinization and minimization of finite VTSs, as they may be used to explicitly represent verdictors.

Determinization. In automata theory it is well-known that a non-deterministic finite automaton can be transformed into an equivalent deterministic one at the cost of an exponential blow up of the state space [RS59]. A similar result has been shown for lattice automata [KL07]. Similarly, any finite VTS can be determinized by applying the usual power set construction [RS59].

Definition 3.2.7 *Let $\mathfrak{V} = \langle Q, J, \mathsf{Obs}, \rightarrow, \mathcal{V}, \sqsubseteq, \nu \rangle$ be a finite VTS. We define a deterministic VTS $\mathfrak{V}_{\text{det}} \coloneqq \langle \wp(Q) \setminus \varnothing, \{J\}, \mathsf{Obs}, \rightarrow', \mathcal{V}, \sqsubseteq, \nu' \rangle$ with*

$$\nu'(Q) \coloneqq \bigsqcup \{ \nu(q) \mid q \in Q \} \tag{3.2}$$

and where $\langle Q, \mathsf{o}, Q' \rangle \in \rightarrow'$ iff $Q' = \mathsf{Post}(Q, \{\mathsf{o}\}) \neq \varnothing$.

Importantly, determinization preserves the produced verdicts:

Theorem 3.2.1 *For each finite VTS $\mathfrak{V}$, $\mathfrak{V}$ and $\mathfrak{V}_{\text{det}}$ are verdict-equivalent.*

Proof. From the traditional power set construction we inherit that $\mathfrak{V}_{\text{det}}$ is indeed deterministic, that it accepts the same language as $\mathfrak{V}$, and that $\nabla\mathsf{Post}_{\text{det}}(\omega) = \mathsf{Post}(\omega)$ for all $\omega \in \mathcal{L}(\mathfrak{V})$. Applying Lemma 3.2.2, it remains to show $\nu(\omega) = \nu_{\text{det}}(\omega)$ for all $\omega \in \mathcal{L}(\mathfrak{V})$. Using definitions (3.1) and (3.2), we obtain:

$$\nu(\omega) = \bigsqcup \{ \nu(q) \mid q \in \mathsf{Post}(\omega) \} = \nu_{\text{det}}(\nabla\mathsf{Post}_{\text{det}}(\omega)) = \nu_{\text{det}}(\omega)$$

□

This result also generalizes to infinite VTSs. If a VTS is infinite, then the join in (3.2) may be over an infinite non-empty set of verdicts. As the verdict domain is a complete join-semilattice (recall Definition 3.1.1), the join of any infinite set of verdicts is defined. Hence, Definition 3.2.7 also applies to infinite VTSs.

The complexity of the construction for finite VTSs is inherited from the traditional power set construction. As with standard finite automata, determinization may lead to an exponential blowup of the reachable state space. In addition, for each of the $2^{|Q|}$ states of $\mathfrak{V}_{\text{det}}$ the join over at most $|Q|$ verdicts must be computed as per (3.2). Hence, the worst-case time complexity is $\mathcal{O}(2^{|Q|} \cdot \mathsf{LOpCost}(|Q|))$. Recall that $\mathsf{LOpCost}(k)$ is the complexity of computing the join/meet over k elements (cf. Section 2.1).

Minimization. In addition to determinization, another well-known result in automata theory concerns minimization. For any finite automaton there is a unique minimal deterministic finite automaton accepting the same language [Ner58; Myh57]. This result carries over to VTSs. Let us first define *minimality*.

Definition 3.2.8 *A finite deterministic VTS $\mathfrak{V}$ is* minimal *iff all deterministic VTSs that are verdict-equivalent to $\mathfrak{V}$ have at least as many states as $\mathfrak{V}$.*

We now establish the following result.

Theorem 3.2.2 *For each finite VTS there is a unique minimal deterministic VTS.*

Proof Sketch. In contrast to deterministic finite automata, the transition relation of a deterministic VTS may be a partial function. To see why the results for finite automata carry over to finite VTSs, assume all missing transitions to end in a non-accepting trap state, while all other states are accepting. In the minimal VTS, states then correspond to the classes of the coarsest partition where states with distinct verdicts or Myhill-Nerode equivalence classes [Ner58; Myh57] are separated. These classes guarantee verdict-equivalence as states with different verdicts or a different language belong to different classes. Further, there cannot be less states since this would require merging states with different verdicts or Myhill-Nerode classes, which would result in a VTS that is no longer verdict-equivalent. □

Typical minimization algorithms for finite automata use partition refinement. Building upon Hopcroft's earlier work [Hop71], Valmari and Lehtinen present an algorithm starting with a partition into accepting and non-accepting states [VL08]. Their algorithm is specifically designed for finite automata with partial transitions. VTSs do not have accepting and non-accepting states but they do have partial transitions, i.e., they are not input-enabled. The algorithm by Valmari and Lehtinen is easy to adapt for VTS minimization by initially partitioning states according to their verdicts instead of into accepting and non-accepting states. The worst-case time complexity for the resulting VTS minimization algorithm is inherited from the original algorithm for finite automata with partial transitions since all states of a VTS may still form their own class. It lies in $\mathcal{O}(|\rightarrow| \cdot \log |Q|)$.

Recall that VTSs generalize lattice automata via simplification (see Lemma 3.2.1). A similar minimization result for simple lattice automata has been stated by Halamish and Kupferman, while for general LAs minimization has been shown to be NP-complete [HK15]. General LAs allow for a more succinct representation than simple LAs explaining the complexity difference for their minimization.

3.3 Observation Models

With VTSs, we introduced a formal representation of verdictors (recall Figure 1.1). We will now introduce *observation models* as a formalization of how system runs give rise to observation sequences, which can then be fed into a verdictor to obtain a verdict. Given a set Obs of observables and a TS $\mathfrak{S}$, the idea is that each run $\rho \in \mathsf{Runs}(\mathfrak{S})$ gives

rise to some observation sequence in a set $\Omega(\rho) \subseteq \text{Obs}^\star$, i.e., $\Omega(\rho)$ is the non-empty set of observation sequences that a given run may *generate*. Having a set for each run enables us to have multiple possible observation sequences per run, e.g., as the result of non-deterministic losses of observations. If a run ρ generates some observation sequence $\omega \in \Omega(\rho)$, then the observation sequence should be a continuation of some earlier observation sequence generated by some prefix of the run. The intuition behind this continuity criterion is that observations are assumed to arrive one at a time. Thus, there must not be any gaps or jumps from one observation sequence to a completely different one. Formally, these considerations give then rise to the following definition of observation models.

Definition 3.3.1 *Given a transition system* $\mathfrak{S}$ *and a set* Obs *of observables, an* observation model *is a function* $\Omega : \text{Runs}(\mathfrak{S}) \to \wp(\text{Obs}^\star) \setminus \{\varnothing\}$ *mapping each run* $\rho \in \text{Runs}(\mathfrak{S})$ *of* $\mathfrak{S}$ *to a non-empty set of observation sequences* $\Omega(\rho) \subseteq \text{Obs}^\star$ *such that the following condition is met for each* $\omega \in \Omega(\rho)$*:*

$$\exists 0 \leq m \leq |\rho| : \text{Tail}(\omega) \in \Omega(\rho[1..m]) \wedge \forall m < k \leq |\rho| : \omega \in \Omega(\rho[1..k]) \quad (3.3)$$

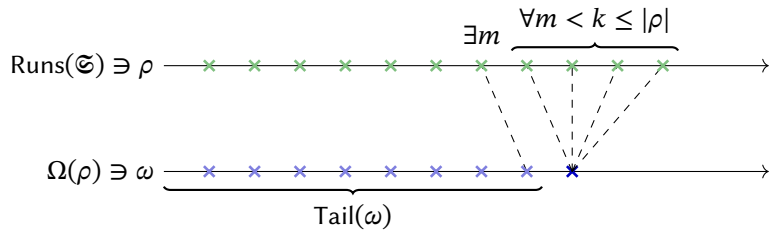


Figure 3.5: Visual representation of condition (3.3) of Definition 3.3.1.

Figure 3.5 visualizes the condition (3.3) of Definition 3.3.1. The dashed lines represents the mapping from prefixes of the run to observation sequences. The definition requires that some prefix of length m generates the tail of the observation sequence and that all longer prefixes generate the entire observation sequence. As any prefix of a run is itself a run, the definition then also applies to the prefixes and the observation sequences they generate, in particular, to the prefix of length m that generates $\text{Tail}(\omega)$. Hence, all prefixes of the run must be mapped to corresponding prefixes of the observation sequence while adhering to Definition 3.3.1. Figure 3.6 shows a possible mapping by some observation model. While any transition of a run may give rise to none, a single, or multiple new observations, Definition 3.3.1 effectively ensures that observation sequences can always be extended one observation at a time.

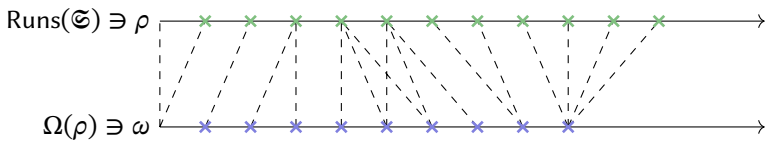


Figure 3.6: Possible mapping from a run's prefixes to observation sequences.

We call an observation model *deterministic* iff it assigns a unique observation sequence to each run, i.e., iff $|\Omega(\rho)| = 1$ for all $\rho \in \mathsf{Runs}(\mathfrak{S})$. Otherwise, it is *non-deterministic*. Non-deterministic observation models will become key when we model observational imperfections, such as non-deterministic losses, where there simply is no uniquely determined observation sequence per system run.

Example 3.3 Recall Example 3.1 where we gave a VTS indicating faults which certainly occurred only by observing the non-fault actions of the system in Figure 2.1. Formally, the observations fed to the VTS are obtained through the model

$$\Omega(\rho) \coloneqq \{\mathsf{Trace}(\rho)|_{\mathsf{Act}\setminus\mathsf{Faults}}\}$$

where $\mathsf{Faults} \coloneqq \{\texttt{short\_circuit}, \texttt{pump\_fault}\}$ is the set of fault actions. For each run ρ, Ω takes the trace $\mathsf{Trace}(\rho)$ and then projects onto the non-fault actions. Hence, Ω is deterministic as each run maps to exactly one projected trace.

Possible Runs. While the observation model maps runs to sets of possible observations sequences, we can also consider the opposite direction in order to obtain a set $\mathsf{Runs}(\omega)$ of *possible runs* for each observation sequence $\omega \in \mathsf{Obs}^\star$:

$$\mathsf{Runs}(\omega) \coloneqq \{\, \rho \in \mathsf{Runs}(\mathfrak{S}) \mid \omega \in \Omega(\rho) \,\} \tag{3.4}$$

Having observed ω, we know that the potentially ongoing run of the system must be in the set $\mathsf{Runs}(\omega)$ as only those runs can give rise to the observation sequence ω through the observation model Ω. Having observed ω, we are then usually interested in properties of the set $\mathsf{Runs}(\omega)$ of possible runs.

Example 3.4 In case of diagnosis, having observed some sequence ω of observable actions, a fault certainly occurred iff it occurred on all runs $\mathsf{Runs}(\omega)$ possible given ω: If a fault occurred on all runs possible given ω, then there exists no run that may generate the observation sequence ω and where the fault did not occur. Hence, the fault certainly occured. Recall the VTS for diagnosing the coffee machine (Figure 3.4). Indeed, assuming that the coffee machine (Figure 2.1) is observed through the model defined previously in Example 3.3, this VTS produces the largest set of fault classes of which faults occurred on all possible runs for an observation sequence, i.e., of which faults certainly occurred. Phrased differently, it produces most specific verdicts that accurately capture which faults certainly occurred.

Observable Language. When viewed through an observation model, the runs of a system give rise to an *observable language* $\mathcal{L}|_\Omega(\mathfrak{S}) \subseteq \text{Obs}^\star$. That is, the set of all observation sequences that may be generated by some run of the system. It captures which observations a verdictor may possible make when observing the system.

Definition 3.3.2 *Given a transition system $\mathfrak{S}$, a set* Obs *of observables, and an observation model $\Omega : \text{Runs}(\mathfrak{S}) \to \wp(\text{Obs}^\star) \setminus \{\varnothing\}$, we define the* observable language $\mathcal{L}|_\Omega(\mathfrak{S})$ *of $\mathfrak{S}$ with respect to Ω as follows:*

$$\mathcal{L}|_\Omega(\mathfrak{S}) \coloneqq \bigcup \{ \Omega(\rho) \mid \rho \in \text{Runs}(\mathfrak{S}) \}$$

Intuitively, all the observation sequences in $\mathcal{L}|_\Omega(\mathfrak{S})$ may be observed by a verdictor at some point. Note that $\mathcal{L}|_\Omega(\mathfrak{S})$ is prefix-closed due to the properties of observation models, hence, if ω may be observed, then all prefixes of it may be observed. This does not come as a surprise as we defined observation models in a way such that they can be continued one observation at a time (cf. Definition 3.3.1).

Lemma 3.3.1 *Given a transition system $\mathfrak{S}$, a set* Obs *of observables, and an observation model $\Omega : \text{Runs}(\mathfrak{S}) \to \wp(\text{Obs}^\star) \setminus \{\varnothing\}$, the observable language $\mathcal{L}|_\Omega(\mathfrak{S})$ of $\mathfrak{S}$ with respect to Ω is prefix-closed. Formally, that is:*

$$\forall \omega' \in \mathcal{L}|_\Omega(\mathfrak{S}) : \text{Tail}(\omega') \in \mathcal{L}|_\Omega(\mathfrak{S})$$

Proof. Lemma 3.3.1 follows from Definition 3.3.1 and the fact that $\text{Runs}(\mathfrak{S})$ is prefix-closed. Let $\omega' \in \mathcal{L}|_\Omega(\mathfrak{S})$. As $\omega' \in \mathcal{L}|_\Omega(\mathfrak{S})$, there must exist a run $\rho' \in \text{Runs}(\mathfrak{S})$ such that $\omega' \in \Omega(\rho')$. We have to show that $\text{Tail}(\omega') \in \mathcal{L}|_\Omega(\mathfrak{S})$. In case $\omega' = \epsilon$, we have $\text{Tail}(\omega') \in \mathcal{L}|_\Omega(\mathfrak{S})$ since $\text{Tail}(\omega') = \epsilon$, $\epsilon \in \Omega(\epsilon)$ as per Definition 3.3.1, and $\epsilon \in \text{Runs}(\mathfrak{S})$ as $\text{Runs}(\mathfrak{S})$ is prefix-closed. Otherwise, if $\omega' = \omega \diamond o$, we have $\text{Tail}(\omega') \in \mathcal{L}|_\Omega(\mathfrak{S})$ since there exists a prefix $\rho \in \text{Pref}(\rho')$ such that $\omega \in \Omega(\rho)$ by condition (3.3) of Definition 3.3.1. □

As any observable language is prefix-closed (Lemma 3.3.1), each observation sequence can indeed be continued one observation at a time. To this end, we define a set $\text{Next}(\omega)$ of *possible next observables* and a set $\text{Cont}(\omega)$ of *possible continuations* of an observation sequence $\omega \in \mathcal{L}|_\Omega(\mathfrak{S})$. Given $\omega \in \mathcal{L}|_\Omega(\mathfrak{S})$, we define:

$$\text{Next}(\omega) \coloneqq \{ o \in \text{Obs} \mid \omega \diamond o \in \mathcal{L}|_\Omega(\mathfrak{S}) \} \tag{3.5}$$

$$\text{Cont}(\omega) \coloneqq \{ \omega' \in \mathcal{L}|_\Omega(\mathfrak{S}) \mid \omega \in \text{Pref}(\omega') \} \tag{3.6}$$

Trace Observation Model. It is a common assumption that the actions, or a subset of them, of a TS can be observed. This is the case for traditional model-based diagnosis (cf. Section 2.6) and the atomic propositions of our LTL runtime verification

example were also based on the actions of the coffee machine model (cf. Section 2.5.1). Capturing this assumption with an observation model is straightforward:

Definition 3.3.3 *For a TS $\mathfrak{S}$, we define the* trace observation model Ω_{Trace} *by:*

$$\Omega_{\text{Trace}}(\rho) \coloneqq \{\text{Trace}(\rho)\}$$

The trace observation model is deterministic as there is a unique trace for any given run of a TS. Furthermore, the observable language of a TS according to the trace observation model is exactly the language of the TS.

Proposition 3.3.1 *Let $\mathfrak{S}$ be a TS. The observable language $\mathcal{L}|_{\Omega_{\text{Trace}}}(\mathfrak{S})$ induced by the trace observation model Ω_{Trace} is the language of $\mathfrak{S}$. Formally, that is:*

$$\mathcal{L}|_{\Omega_{\text{Trace}}}(\mathfrak{S}) = \mathcal{L}(\mathfrak{S})$$

Proof. Follows trivially from the respective definitions. □

Observation models are a very general modeling formalism for capturing which runs of a system may generate which observation sequences. For the purposes of this thesis, we mostly consider observation models that are derived from the trace observation model. In general, however, different models are conceivable, for instance, models that extract variables from states, or even combine state and action information. While we will usually take the trace observation model as a basis, some of our results, in particular the VTS transformations developed as part of Contribution TT, also work for arbitrary other observation models. This is enabled by the concept of observation model transformers that we discuss next.

3.3.1 Observation Model Transformers

Technically, we could describe different observational imperfections by spelling out observation models for each of them individually, as we have done in Example 3.3. Instead, we aim to make observational imperfections composable such that multiple orthogonal imperfections can be layered on-top of existing observation models. To this end, we introduce *observation model transformers* as a central instrument for modeling observational imperfections in a composable way.

Approaching observational imperfections in a composable way has two main advantages: First, it makes it easier to specify specific observation models, as they can be obtained by taking a base model and transforming it appropriately, thereby allowing reusability. Second, it enables us to take a modular approach towards VTS synthesis. In particular, the correctness of the VTS transformations developed as part

of Contribution TT will be proven relative to observation model transformers and, thus, also work for arbitrary base models.

An observation model transformer Θ transforms one observation model into another. Formally, an observation model transformer is a partial function $\Theta : X \rightharpoonup X$ where X is the set of all observation models for all transition systems and sets of observables. They are partial functions as they may require certain preconditions on the observation models to which they apply. To avoid unnecessary formal overhead, we omit a detailed formal exposition of this general concept. The value lies primarily in the concept itself. We are going to define concrete instances throughout this thesis to handle limited observability and network-induced delays and losses (cf. Chapter 1). The observation model transformers for delays and losses will be introduced later in Section 4.3, where we also present the corresponding VTS transformations. The transformer for limited observability is introduced here.

Limited Observability. In practice, there are always limits to what can be observed. For instance, network bandwidth, the cost of deploying sensors, and the performance penalty for instrumenting software with logging, limit the observations that can be made about a system in practice. Formally, we capture this by assuming that only a certain subset $\mathsf{OAct} \subseteq \mathsf{Act}$ of actions Act of a system model can be observed. In case of fault diagnosis of the coffee machine (Example 3.3), we already saw such an example where only non-fault actions can be observed.

In general, given an observation model $\Omega : \mathsf{Runs}(\mathfrak{S}) \to \wp(\mathsf{Obs}^\star) \setminus \{\varnothing\}$ over a set Obs of observables, limited observability restricts the observables to a subset $\mathsf{O} \subseteq \mathsf{Obs}$, i.e., starting with an observation model where Obs are the observables, we end up with an observation model where the set of observables is limited to O. Formally, this transformation is defined as follows:

Definition 3.3.4 *Given an observation model* $\Omega : \mathsf{Runs}(\mathfrak{S}) \to \wp(\mathsf{Obs}^\star) \setminus \{\varnothing\}$ *over a set* Obs*, we define the* O-restriction *of* Ω *for* $\mathsf{O} \subseteq \mathsf{Obs}$ *as:*

$$\Omega{\downarrow_\mathsf{O}}(\rho) := \{\, \omega{\downarrow_\mathsf{O}} \mid \omega \in \Omega(\rho) \,\}$$

Recall that $\omega{\downarrow_\mathsf{O}}$ removes all the symbols from ω that are not in O (see Section 2.1). The observation model transformer $\cdot{\downarrow_\mathsf{O}}$ preserves determinism.

Proposition 3.3.2 *If* Ω *is deterministic, then* $\Omega{\downarrow_\mathsf{O}}$ *is deterministic.*

Proof. Trivially follows from Definition 3.3.4. □

Example 3.5 The observation model defined in Example 3.3 for diagnosis of the coffee machine can now simply be composed from Ω_{Trace} and $\cdot{\downarrow_\mathsf{O}}$. It is $\Omega_{\mathsf{Trace}}{\downarrow_{\mathsf{Act}\setminus\mathsf{Faults}}}$. That

is, all actions except the fault actions are observable. In general, the observation model implicitly assumed for traditional model-based fault diagnosis is $\Omega_{\text{Trace}}|_{\text{OAct}}$, where OAct is the set of observable actions (cf. Section 2.6). The existing work does, however, not define this observation model compositionally.

Observability limits as described above have been discussed in the literature under the term "partial observability" before [RW89]. They are also central to model-based diagnosis where faults are assumed unobservable [Sam+95] (recall Section 2.6). We reframed those ideas in the broader theoretical framework introduced thus far. Importantly, while being able to capture observability limitations, the framework we present here is more general and powerful. We will demonstrate and exploit this expressive power later when modeling network-induced delays and losses as well as clock drift and offsets within the framework. Related to limited observability is "partial observability" as it appears in the literature on decision making under uncertainty [Åst65]. In contrast to the notion of partial observability in the works mentioned before [RW89; Sam+95], here observations are subject to probabilistic noise, i.e., whenever an event occurs observations are made according to some probability distribution. Observation models as we introduced them can over-approximate this notion of partial observability by considering all observables possible which do have a non-zero probability. We leave it to future work to investigate the extension of observation models to the probabilistic setting. For the observational imperfections we consider in this thesis, such an extension is not required.

3.3.2 Applicability and Tightness

Verdictors should be able to account for any observation sequences that a system may generate according to an observation model. Otherwise, we may end up in an undefined situation where we have observed something but the verdictor will not provide any verdict for it. Recall that a system may generate any observation sequence of its observable language (recall Definition 3.3.2). Thus, we require that the VTS representing a verdictor provides a verdict for all these words. We call this requirement *applicability* and formally define it as follows.

Definition 3.3.5 *Let $\mathfrak{S}$ be a TS and $\Omega : \mathsf{Runs}(\mathfrak{S}) \to \wp(\mathrm{Obs}^\star) \setminus \{\emptyset\}$ be an observation model. A VTS $\mathfrak{V}$ is* applicable *with respect to $\mathfrak{S}$ and Ω iff the language $\mathcal{L}(\mathfrak{V})$ of $\mathfrak{V}$ is a superset of the observational language $\mathcal{L}|_\Omega(\mathfrak{S})$ of $\mathfrak{S}$ with respect to Ω, i.e., $\mathfrak{V}$ is* applicable *iff $\mathcal{L}(\mathfrak{V}) \supseteq \mathcal{L}|_\Omega(\mathfrak{S})$.*

If a VTS $\mathfrak{V}$ is applicable, then it produces a verdict $\nu(\omega)$ for each observation sequence that may be generated by the system through the observation model. Otherwise, if a VTS is not applicable, then there exist observation sequences which may be generated by the system but for which the VTS does not produce a verdict. We

always require applicability as it guarantees that we never end up in an undefined situation at runtime where no verdict would be provided.

From a language-theoretic perspective, the question whether a VTS is applicable with respect to a given system and observation model constitutes a language inclusion problem. Language inclusion has been studied in the literature for various classes of languages. In case the language of the VTS and the observable language of the TS system model with respect to an observation model fall into a certain class, those results naturally carry over to deciding applicability. For instance, if the respective languages are regular then applicability is decidable [HMU01]. Notably, this is the case if both the VTS the and system model are finite and the observation model preserves regularity. An example of an observation model preserving regularity is the trace model combined with limited observability. This is the case because projection is known to preserve regularity [JM11]. In general, neither VTSs nor TSs nor observation models pose any restrictions on the respective languages (except being prefix-closed). Without any restrictions, language inclusion is known to be undecidable [HMU07]. This result carries over to general VTSs and TSs which can be infinite and, therefore, can model the execution of an arbitrary Turing Machine. Furthermore, observation models may not be computable. The particular VTS constructions presented in this thesis guarantee applicability for the resulting VTSs.

Tightness. A VTS that is applicable may produce verdicts for observation sequences that are never actually generated by a given system. As such, it represents and accounts for behavior that cannot actually be realized by a given system. The VTS transformations presented later, in particular, for predictions, will require a VTS to tightly capture the behavior of a given system. To this end, we introduce a stronger variant of applicability, which we call *tightness*.

Definition 3.3.6 *Let* $\mathfrak{S}$ *be a TS and* $\Omega : \mathsf{Runs}(\mathfrak{S}) \to \wp(\mathsf{Obs}^\star) \setminus \{\varnothing\}$ *be an observation model. A VTS* $\mathfrak{V}$ *is* tight *with respect to* $\mathfrak{S}$ *and* Ω *iff the language* $\mathcal{L}(\mathfrak{V})$ *of* $\mathfrak{V}$ *is the same as the observational language* $\mathcal{L}|_\Omega(\mathfrak{S})$ *of* $\mathfrak{S}$ *with respect to* Ω*, i.e.,* $\mathfrak{V}$ *is* tight *iff* $\mathcal{L}(\mathfrak{V}) = \mathcal{L}|_\Omega(\mathfrak{S})$.

If a VTS $\mathfrak{V}$ is tight, then it produces a verdict $\nu(\omega)$ for all and only those observation sequences that may be generated by the system through the observation model. It is easy to see that any tight VTS is also applicable.

From a language-theoretic perspective, the question whether a VTS is tight with respect to a given system and observation model constitutes a language equivalence problem. Language equivalence has been studied in the literature for various classes of languages and is known to be undecidable in general [HMU07]. As for deciding applicability, those results carry over to tightness if the languages fall into the

respective classes. For the VTS constructions and transformations presented in this thesis, we show tightness and its preservation, respectively.

Example 3.6 The VTS for diagnosing the coffee machine shown in Figure 3.4 is tight with respect to the system model shown in Figure 2.1 and the observation model as defined in Example 3.3. It accepts exactly the traces of the original model with the fault actions being removed by projection.

If a given finite VTS $\mathfrak{V}$ is not tight with respect to a given finite system model $\mathfrak{S} = \langle S, I, \mathrm{Act}, \twoheadrightarrow \rangle$ and the observation model $\Omega_{\mathrm{Trace}}{\restriction_{\mathrm{OAct}}}$ where $\mathrm{OAct} \subseteq \mathrm{Act}$ is an observable subset of the actions of $\mathfrak{S}$, we can transform $\mathfrak{V}$ such that it becomes tight by building the synchronized product of the VTS $\mathfrak{V}$ and system model $\mathfrak{S}$. Formally, this transformation, coined *tightening*, is the product of the VTS $\mathfrak{V}$ and the system model $\mathfrak{S}$ obtained by synchronizing over the observable actions OAct:

Definition 3.3.7 *Let $\mathfrak{S} = \langle S, I, \mathrm{Act}, \twoheadrightarrow \rangle$ be a TS and $\mathfrak{V} = \langle Q, J, \mathrm{OAct}, \rightarrow, \mathcal{V}, \sqsubseteq, \nu \rangle$ be an applicable VTS with $\mathrm{OAct} \subseteq \mathrm{Act}$. Let $\mathfrak{V}' := \langle S \times Q, \mathrm{Act}, I \times J, T', \mathcal{V}, \nu' \rangle$ be a VTS with $\nu'(\langle s, q \rangle) = \nu(q)$, and where $\langle \langle s, q \rangle, \alpha, \langle s', q' \rangle \rangle \in T'$ iff (1) $\alpha \in \mathrm{OAct}$, $\langle s, s' \rangle \in \twoheadrightarrow$, and $\langle q, q' \rangle \in \twoheadrightarrow'$, or (2) $\alpha \notin \mathrm{OAct}$, $\langle s, s' \rangle \in \twoheadrightarrow$, and $q = q'$.*

The VTS $\mathfrak{V}'$ specializes $\mathfrak{V}$ for the system model $\mathfrak{S}$. As $\mathfrak{V}$ is applicable, the language of the product becomes the language of the TS, i.e., $\mathcal{L}(\mathfrak{V}') = \mathcal{L}(\mathfrak{S})$. Further, the verdicts produced by $\mathfrak{V}'$ are carried over from the original VTS $\mathfrak{V}$, i.e., $\nu'(\omega) = \nu(\omega{\restriction_{\mathrm{OAct}}})$ for each $\omega \in \mathcal{L}(\mathfrak{V}')$. The worst-case time complexity for this tightening construction is $\mathcal{O}(|Q| \cdot |S| \cdot |\twoheadrightarrow| \cdot |\twoheadrightarrow'|)$.

3.4 Provably Accurate Verdicts

Recall that it is an express purpose of the theoretical framework to provide rigorous criteria for what it means for a verdictor to produce accurate verdicts with respect to a given system and observation model (FT1). We assumed that verdicts come with an inherent notion of *specificity*, partially ordering them from least specific to most specific. We want verdicts to at least *correctly* reflect the state of the system at all times. For instance, a runtime monitor should only indicate that a property is satisfied, if the property is indeed satisfied. If a runtime monitor, however, cannot determine whether a property is satisfied or violated, it may also provide the unknown verdict ? (cf. Section 3.1). The challenge lies in producing verdicts that are *as specific as possible*, i.e., without sacrificing correctness. If we can tell that a property is satisfied or violated, that a fault occurred, or that the system is in a certain configuration, then we should indicate so. Otherwise, we still want to be as specific as possible, as a safely approximating answer is always better than none. In the following, we refer to these verdicts as being *most specific*, also implying correctness.

With VTSs as a representation of verdictors and observation models as a tool for describing which runs of a system may generate which observation sequences, we now have everything we need to define what it takes for a verdictor to produce correct verdicts and most specific verdicts. We define both notations with respect to a *verdict oracle* which provides a ground truth for verdicts.

Definition 3.4.1 *Given a verdict domain* $\langle \mathcal{V}, \sqsubseteq \rangle$ *and a transition system* $\mathfrak{S}$*, a* verdict oracle *is a function* $\mathsf{V} : \mathsf{Runs}(\mathfrak{S}) \to \mathcal{V}$ *assigning a verdict to each run.*

A verdict oracle is omniscient in the sense that it has access to the entire run, including the states which have been visited and the transitions which have been taken. Based on all this information, it then assigns a verdict to a given run.

Verdict oracles must be defined according to the intended purpose of a verdictor. For instance, fault diagnosis requires a different verdict oracle than configuration monitoring: Given a run, the former needs to look at occurred faults while the latter needs to look at a system's configuration.

Analogously to VTSs, we call a verdict oracle *monotonic* iff the verdicts it ascribes to runs only ever get more specific as a run progresses.

Definition 3.4.2 *A verdict oracle* $\mathsf{V} : \mathsf{Runs}(\mathfrak{S}) \to \mathcal{V}$ *is* monotonic *iff* $\mathsf{V}(\rho') \sqsubseteq \mathsf{V}(\rho)$ *for all* $\rho' \in \mathsf{Runs}(\mathfrak{S})$ *and* $\rho \in \mathsf{Pref}(\rho')$.

Example 3.7 For an example, recall the VTS for diagnosis of the coffee machine (Example 3.1). In this case, we are interested in faults that certainly occurred. Here, the verdict oracle must collect all the faults that occurred on a given run:

$$\mathsf{V}(\varepsilon) := \varnothing \qquad \mathsf{V}(\rho \diamond \langle s, \alpha, s' \rangle) := \begin{cases} \{\alpha\} \cup \mathsf{V}(\rho) & \text{if } \alpha \in \mathsf{Faults} \\ \mathsf{V}(\rho) & \text{otherwise} \end{cases}$$

Given a run, this verdict oracle provides a ground truth regarding the faults that occurred on that run. This verdict oracle is also monotonic. In itself, a verdict oracle does not yet provide a specification for a verdictor. Such a specification is obtained when instantiating the subsequent definitions with a verdict oracle.

Most Specific Verdicts. Given an observation sequence ω, we know that any of the runs in $\mathsf{Runs}(\omega)$ may be the current ongoing run which generated ω. If we join the verdicts assigned to these runs by the verdict oracle, then we obtain the *most specific* verdict, denoted by $\mathsf{V}(\omega)$, for any given observation sequence ω:

$$\mathsf{V}(\omega) \;:=\; \bigsqcup \{ \mathsf{V}(\rho) \mid \rho \in \mathsf{Runs}(\omega) \} \tag{3.7}$$

Note that the join exists for any observation sequence in the observable language because the verdict domain is a complete join-semilattice and there is at least one run generating each observation sequence in the observable language.

3.4.1 Sound, Complete, and Robust VTSs

With soundness and completeness we now define two criteria for VTSs that together guarantee that a VTS produces the most specific verdict. A VTS is called *provably accurate* iff it has been proven to be sound and complete. A verdictor is provably accurate iff its VTS model has been proven sound and complete.

Soundness. A verdict produced by a VTS for an observation sequence is *correct* iff it is at most as specific as the verdict produced by the verdict oracle for a run that may generate the observation sequence. That is, a correct verdict is a safe over-approximation of each verdict ascribed by the verdict oracle to a run that may generate the sequence. We also call such verdicts *sound*.

Formally, soundness of a VTS is defined as follows:

Definition 3.4.3 *Let $\mathfrak{S}$ be a TS, $\Omega : \mathsf{Runs}(\mathfrak{S}) \to \wp(\mathsf{Obs}^\star)\backslash\{\varnothing\}$ be an observation model, $\mathsf{V} : \mathsf{Runs}(\mathfrak{S}) \to \mathcal{V}$ be a verdict oracle, and $\mathfrak{V} = \langle Q, J, \mathsf{Obs}, \rightarrowtriangle, \mathcal{V}, \sqsubseteq, \nu\rangle$ be an applicable VTS over the verdict domain $\langle \mathcal{V}, \sqsubseteq\rangle$. We call $\mathfrak{V}$* sound *with respect to $\mathfrak{S}$, V, and Ω iff $\mathsf{V}(\rho) \sqsubseteq \nu(\omega)$ for all $\omega \in \mathcal{L}|_\Omega(\mathfrak{S})$ and $\rho \in \mathsf{Runs}(\omega)$.*

Recall that any run in $\mathsf{Runs}(\omega)$ may generate an observation sequence ω.

Example 3.8 The VTS for diagnosing the coffee machine as shown in Figure 3.4 (see Example 3.1) is indeed sound with respect to the system model shown in Figure 2.1 (see Example 2.1), the observation model as defined in Example 3.3, and the verdict oracle as defined above in Example 3.7. In this case, soundness means that a verdict produced by the VTS only contains faults that actually occurred on all runs that may induce an observation sequence, i.e., that certainly occurred.

A verdictor is sound iff it produces a verdict that is not more specific than the most specific verdict. Formally, we obtain the following proposition:

Proposition 3.4.1 *A VTS $\mathfrak{V}$ is sound with respect to $\mathfrak{S}$, V, and Ω, iff $\mathsf{V}(\omega) \sqsubseteq \nu(\omega)$ for all $\omega \in \mathcal{L}|_\Omega(\mathfrak{S})$.*

Proof Sketch. Follows from the join-semilattice properties. □

Verdict domains have a least specific top element $\top$. A VTS always producing this top element is guaranteed to be sound but is not practically useful. For instance, in the case of diagnosis of certain faults, it would always be sound to simply indicate that no faults certainly occurred, as this is the top element of the verdict domain as per Definition 3.1.4. Hence, to arrive at a comprehensive specification, we need to complement soundness with one of the *completeness criteria* presented next.

Completeness. Based on the most specific verdict as per (3.7), we define a notion of completeness complementing soundness. While soundness requires the verdict to be at most as specific as the most specific verdict, completeness generally requires the verdict to be at least as specific as the most specific verdict.

Definition 3.4.4 *Let $\mathfrak{S}$ be a TS, $\Omega : \mathsf{Runs}(\mathfrak{S}) \to \wp(\mathrm{Obs}^\star)\backslash\{\varnothing\}$ be an observation model, $\mathsf{V} : \mathsf{Runs}(\mathfrak{S}) \to \mathcal{V}$ be a verdict oracle, and $\mathfrak{V} = \langle Q, J, \mathrm{Obs}, \rightarrowtriangle, \mathcal{V}, \sqsubseteq, \nu\rangle$ be an applicable VTS over the verdict domain $\langle \mathcal{V}, \sqsubseteq\rangle$. We call $\mathfrak{V}$* complete *with respect to $\mathfrak{S}$, V, and Ω iff $\nu(\omega) \sqsubseteq \bigsqcup\{\mathsf{V}(\rho) \mid \rho \in \mathsf{Runs}(\omega)\}$ for all $\omega \in \mathcal{L}|_\Omega(\mathfrak{S})$.*

It is easy to see that a sound and complete VTS produces exactly the most specific verdict for a given observation sequence. Formally, that is:

Lemma 3.4.1 *A VTS $\mathfrak{V}$ is sound and complete with respect to $\mathfrak{S}$, V, and Ω, iff $\nu(\omega) = \mathsf{V}(\omega)$ for all $\omega \in \mathcal{L}|_\Omega(\mathfrak{S})$.*

Proof Sketch. Follows from the antisymmetry of $\sqsubseteq$. □

Example 3.9 The VTS for diagnosing the coffee machine as shown in Figure 3.4 (see Example 3.1) is sound and complete with respect to the system model shown in Figure 2.1 (see Example 2.1), the observation model as defined in Example 3.3, and the verdict oracle as defined in Example 3.7. Hence, it identifies all and only those faults that certainly occurred. In the corresponding verdict domain, the join $\sqcup$ is set intersection $\cap$. The verdict oracle (see Example 3.7) collects all faults that did occur on a single run in a set. These sets are then intersected to obtain the most specific verdict as per (3.7). Adding a fault to that set would render the verdict unsound as it means that there exists a run where the fault did not occur, i.e., it did not certainly occur. Analogously, removing a fault from the set would render the verdict incomplete, as the fault occurred on all runs, i.e., it did certainly occur.

In terms of the refinement relation (see Section 3.2), a VTS produces most specific verdicts iff it cannot be further refined without becoming unsound.

A significant part of this thesis is devoted to algorithms for implementing or synthesizing verdictors such that the VTSs that represent them are sound and complete for a given system and observation model. To this end, the definitions we just provided serve as the specification and constitute proof obligations.

Δ-Completeness. In case observations are delayed, completeness requires us to speculate about potential missing observations which the ongoing run will inevitably generate. In particular, in the continuous-time setting, such speculation becomes expensive and complicated. Instead of speculating, a verdictor may delay the most specific verdict by some *verdict offset* $\Delta \in \mathbb{R}$. We call this criterion Δ-*completeness*

and formally define it with respect to a distance metric $d : \mathrm{Obs}^\star \times \mathrm{Obs}^\star \to \mathbb{R}$ on observation sequences. For instance, $d(\omega, \omega')$ may be the (discrete) time difference of two observation sequences ω and ω', i.e., $d(\omega, \omega') := |\omega| - |\omega'|$.

Definition 3.4.5 *Let $\mathfrak{S}$ be a TS, $\Omega : \mathrm{Runs}(\mathfrak{S}) \to \wp(\mathrm{Obs}^\star)\backslash\{\varnothing\}$ be an observation model, $\mathsf{V} : \mathrm{Runs}(\mathfrak{S}) \to \mathcal{V}$ be a verdict oracle, and $\mathfrak{V} = \langle Q, J, \mathrm{Obs}, \rightarrowtriangle, \mathcal{V}, \sqsubseteq, \nu \rangle$ be an applicable VTS over the verdict domain $\langle \mathcal{V}, \sqsubseteq \rangle$. Given a verdict offset $\Delta \in \mathbb{R}$ and a distance metric $d : \mathrm{Obs}^\star \times \mathrm{Obs}^\star \to \mathbb{R}$, we call $\mathfrak{V}$* Δ-complete *with respect to $\mathfrak{S}$, V, and Ω iff $\nu(\omega') \sqsubseteq \mathsf{V}(\omega)$ for all $\omega' \in \mathrm{Cont}(\omega)$ as per (3.6) and $\omega \in \mathcal{L}|_\Omega(\mathfrak{S})$ such that $d(\omega, \omega') \geq \Delta$.*

Figure 3.7 visualizes Definition 3.4.5 for some continuation $\omega' \in \mathrm{Cont}(\omega)$ of an observation sequence ω. $\mathsf{V}(\omega)$ is the most specific verdict as per (3.7) for the observation sequence ω. $\nu(\omega')$ is the verdict produced by the VTS for the continuation ω' of ω. The definition requires that for all continuations ω' of an observation sequence ω with a distance of at least Δ (i.e., after a delay of at most Δ) the produced verdict $\nu(\omega')$ is at least as specific as the most specific verdict $\mathsf{V}(\omega)$. Thus, instead of speculating about potentially missing observations arriving later, a Δ-complete verdictor may wait for future observations and only produce the most specific verdict *after* they arrived. The verdict offset Δ serves as a statically known upper bound on the delay with which the most specific verdict is produced. That is, in the worst case, the production of the most specific verdict may be delayed by up to Δ.

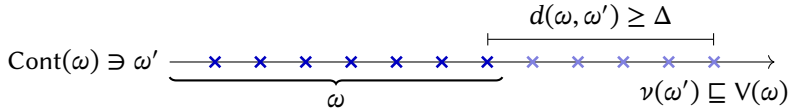


Figure 3.7: Visualization of Definition 3.4.5 for some continuation $\omega' \in \mathrm{Cont}(\omega)$.

Definition 3.4.5 is a generalization of Δ-completeness as defined in the paper on robust diagnosis of real-time systems by the author of this thesis [KH23]. The original definition is specific to diagnosis of certain faults whereas the definition presented here is generic over arbitrary verdict domains.

Note that Δ-completeness is limited to monotonic VTSs and verdict oracles, as it concerns all possible continuations with a distance of at least Δ. Monotonic VTSs occur for configuration monitoring and certain instances of fault diagnosis and runtime verification.

Robustness. We say that a verdictor is *robust* against certain observational imperfections, if it is sound and complete or Δ-complete with respect to a given system model, verdict oracle, and observation model where the observation model models

the respective observational imperfections. That is, the verdictor produces most specific verdicts despite being fed imperfect observations. For instance, we say that a verdictor is robust against losses, if it produces most specific verdicts despite being fed with observations that are subject to losses. We also apply this notion of robustness to the algorithms we develop in Chapter 4 and Chapter 5. Those algorithms enable the synthesis or implementation of robust verdictors given a system model.

3.4.2 VTS Synthesis Problem

Soundness and completeness provide a formal basis for the specification of verdictors. The algorithms contributed in this thesis (Contribution TT) enable the synthesis and implementation of verdictors that are indeed sound and complete—and, we will prove that they do. Conceptually, these algorithms solve various instances of the *VTS Synthesis Problem*, i.e., the problem of finding a VTS that produces most specific verdicts for a given system with respect to an observation model:

VTS Synthesis Problem. Given a system model $\mathfrak{S}$, an observation model Ω, and a verdict oracle V, find a VTS $\mathfrak{V}$ that produces sound and complete or Δ-complete verdicts with respect to $\mathfrak{S}$, Ω, and V.

In the discrete-time setting and given a finite system model $\mathfrak{S}$, the synthesis algorithms presented in Chapter 4 solve certain instances of the VTS Synthesis Problem by explicitly synthesizing VTSs. The algorithms for the continuous-time setting construct and explore a VTS on-the-fly at runtime. They solve the problem on a conceptual level by providing an algorithm implementing a verdictor for a given system and observation model. The VTS Synthesis Problem is meant to provide an abstract description of the problems we are trying to solve. Verdictors and their VTS models solving the VTS Synthesis Problem produce provably accurate verdicts with respect to the involved models and verdict oracle.

Note that the objects appearing in the VTS Synthesis Problem may not be finitely representable and that it is therefore easy to come up with verdict oracles or observation models that are not computable. In this general version, the problem statement does not directly aim at algorithmic solutions. With a few assumptions on these objects we will, however, be able to provide a general solution that applies to all system models, observation models, and verdict oracles satisfying them.

A Naive Solution. Mathematically defining *some* VTS that produces most specific verdicts is indeed straightforward. A VTS can simply track all observations in its state and then produce the most specific verdict $\mathsf{V}(\omega)$ as per (3.7):

Proposition 3.4.2 *Given a system model $\mathfrak{S}$, an observation model Ω, and a verdict oracle V, a VTS $\mathfrak{V}$ producing most specific verdicts is given by*

$$\left\langle \mathrm{Obs}^{\star}, \{\epsilon\}, \mathrm{Obs}, \rightarrowtail, \mathcal{V}, \sqsubseteq, \nu \right\rangle$$

where $\langle \omega, \mathsf{o}, \omega' \rangle \in \rightarrowtail$ iff $\omega' = \omega \diamond \mathsf{o}$ and $\nu(\omega) \coloneqq \mathsf{V}(\omega)$.

This definition works regardless of the system model, observation model, and verdict oracle. As tracking all observations is trivially implemented, the crux of whether we can actually implement this VTS hinges on whether $\mathsf{V}(\omega)$ is computable. Algorithm 1 shows an algorithm for computing $\mathsf{V}(\omega)$. It requires the following assumptions:

- A1 The set $\Omega(\rho)$ of observations generated by a run ρ according to the observation model must be finite and computable, as it is used in lines 8 and 12.
- A2 The set of a *one-step continuations* of a run ρ, denoted by $\textsc{Next}(\rho)$ must be finite and computable. A one-step continuation continues a run by exactly one transition. It is used in line 7.
- A3 The verdict oracle V must be computable. It is used in line 14.
- A4 The join of two verdicts must be computable. It is used in line 14.
- A5 In addition, the observation model must be such that always eventually a new observation is generated, i.e., it should be impossible to continue a run indefinitely without eventually getting a new observation. This property is required to ensure termination of the algorithm.

The algorithm works by first computing the set R of runs that may generate the observation ω, i.e., $R = \mathrm{Runs}(\omega)$. To this end, it explores all the runs of the system model $\mathfrak{S}$ that generate at least one observation sequence that is a prefix of ω (checked in line 9). If a run only generates observation sequences that are not a prefix of ω, then no continuation of this run will generate ω since the observation sequences generated by a run's continuation must extend the observation sequences of the run. This is guaranteed by the definition of observation models (Definition 3.3.1). Hence, a run can be discarded in that case. Together with the assumption A5 that always eventually a new observation will be generated, this exploration will eventually terminate. While all the runs in the set R do generate a prefix of ω, they are not guaranteed to generate ω itself. Hence, we need to filter that set to obtain $\mathrm{Runs}(\omega)$ (line 12). At this point, it remains to compute the most specific verdict for this set of runs, exploiting the fact that V and $\sqcup$ are computable as per A3 and A4.

While Algorithm 1 applies, if all the assumptions are satisfied, it is not practical. To use it, a verdictor needs to keep track of all the observations that ever occurred. This results in an unbounded memory consumption as the system keeps running and new observations arrive. Furthermore, the computation of $\mathsf{V}(\omega)$ itself is very inefficient, requiring an exploration of runs with every new observation. While this inefficiency can be resolved with clever caching to some degree, fundamentally,

Algorithm 1: Computing the most specific verdict $\mathsf{V}(\omega)$ for $\omega \in \mathrm{Obs}^\star$.

Data: an observation sequence $\omega \in \mathrm{Obs}^\star$
Result: the most specific verdict $\mathsf{V}(\omega)$ for ω, if it is defined

```
1  function ComputeV(ω)
2      R := ∅
3      W := {ε}
4      while |W| > 0 do
5          ρ := choose an arbitrary run out of W
6          R, W := R ∪ {ρ}, W \ {ρ}
7          for ρ' ∈ Next(ρ) do
8              for ω' ∈ Ω(ρ') \ R do
9                  if ω' ∈ Pref(ω) then
10                     W := W ∪ {ρ'}
11                     break
12     R := { ρ ∈ R | ω ∈ Ω(ρ) } // = Runs(ω)
13     if |R| > 0 then
14         return ⨆ { V(ρ) | ρ ∈ R }
15     else
16         return undefined // ω ∉ L|_Ω(𝒞)
```

without additional assumptions, the memory consumption will still grow without bounds with every new observation. For instance, when caching the set of runs, such that we do not need to recompute it all the time, the size of the cache grows without bounds because the size of the runs increases with every new observation. In general, we need to store runs in their entirety as the verdict oracle may depend on all the information of a run.

Another shortcoming of Algorithm 1 is that it does not apply to continuous-time systems, e.g., modeled by a timed automaton, as those do not satisfy assumption A2. If time can pass in arbitrary increments, then there is an uncountably infinite number of one-step continuations of a given run. Further, if timing imprecisions, as we consider them in Chapter 5, are involved, then an observation model may produce an uncountably infinite number of observation sequences, where observations are made at slightly different points in time. As a result, in that case, assumption A1 is also violated rendering the algorithm infeasible.

Therefore, in the remainder of this thesis, we develop algorithms that allow the effective implementation or synthesis of verdictors, whose memory consumption does not grow out of bounds and that can handle continuous-time systems as well as timing imprecisions and other observational imperfections. Naturally, these will require different assumptions than Algorithm 1.

3.5 A Unifying Foundation

The theoretical framework we presented is abstract and general. Showcasing its versatility, we now discuss how it generalizes and unifies existing work in the spectrum of automata- and stream-based runtime verification and model-based diagnosis (FT3). To this end, we consider several paradigmatic examples from the relevant literature [BLS07; BLS06b; BLS11; Pin+17a; Sam+95; Sam+96; DAn+05].

3.5.1 Traditional Model-Based Fault Diagnosis

Recall Section 2.6, where we introduced traditional model-based diagnosis [Sam+95; Sam+96]. In Section 3.1, we defined the verdict domain $\langle \wp(\wp(\mathcal{F})), \subseteq \rangle$ for traditional diagnosis (see Definition 3.1.2), where $\mathcal{F}$ is the set of fault classes according to traditional model-based diagnosis. Recall that a diagnoser is a deterministic TS

$$\mathcal{D} = \langle \wp(\mathcal{S} \times \wp(\mathcal{F})), I_{\mathcal{D}}, \mathrm{OAct}, \twoheadrightarrow_{\mathcal{D}} \rangle$$

synthesized from a system model $\langle \mathcal{S}, I, \mathrm{Act}, \twoheadrightarrow \rangle$ where $\mathrm{OAct} \subseteq \mathrm{Act}$ is a set of observable actions. Using the diagnosis function d as per (2.5) mapping states of the diagnoser to diagnoses, we can easily cast the diagnoser into a VTS:

$$\mathfrak{V}_{\mathcal{D}} \;\coloneqq\; \langle \wp(\mathcal{S} \times \wp(\mathcal{F})), I_{\mathcal{D}}, \mathrm{OAct}, \twoheadrightarrow_{\mathcal{D}}, \wp(\wp(\mathcal{F})), \subseteq, \mathrm{d} \rangle$$

The resulting VTS $\mathfrak{V}_{\mathcal{D}}$ may be non-monotonic (e.g., see Figure 2.8). Furthermore, diagnosers are constructed such that $\mathcal{L}(\mathcal{D}) = \mathcal{L}(\mathfrak{S})\restriction_{\mathrm{OAct}}$ [cf. Sam+95]. As a result, the VTS $\mathfrak{V}_{\mathcal{D}}$ is tight with respect to $\mathfrak{S}$. For each trace $\sigma \in \mathcal{L}(\mathfrak{S})$ of the diagnosed system, the VTS $\mathfrak{V}_{\mathcal{D}}$ obtained from $\mathcal{D}$ produces as verdict a diagnosis $\nu(\sigma\restriction_{\mathrm{OAct}})$ indicating which faults occurred, taking only observable actions into account. In terms of the concepts we introduced, the observation model here is $\Omega_{\mathrm{Trace}}\restriction_{\mathrm{OAct}}$.

Soundness and Completeness of Diagnosers. The soundness and completeness criteria we developed also apply to traditional diagnosers. To apply them, we first need to define a verdict oracle. For traditional diagnosis this verdict oracle collects the fault classes of those faults that occurred on a run:

$$\mathsf{V}(\epsilon) \coloneqq \{\varnothing\} \qquad \mathsf{V}(\rho \diamond \langle s, \alpha, s' \rangle) \coloneqq \begin{cases} \{\nabla \mathsf{V}(\rho) \cup \{f\}\} & \text{if } \alpha \in f \text{ for } f \in \mathcal{F} \\ \mathsf{V}(\rho) & \text{otherwise} \end{cases}$$

According to this verdict oracle, a run is mapped to a singleton set containing a set of those fault classes that correspond to the occurred faults. Recall that $\mathcal{F}$ partitions the set of fault actions, hence, there exists at most one fault class $f \in \mathcal{F}$ such that $\alpha \in f$, which is required for the verdict oracle to be well-defined.

Diagnosers obtained with the traditional construction introduced by Sampath et al. [Sam+95] turn out to be indeed sound and complete with respect to this verdict

oracle, i.e., they produce most specific verdicts. To see why this is the case, recall that as per (3.7) the most specific verdict is the verdict obtained by joining the verdicts of the verdict oracle. The join operation $\sqcup$ of the verdict domain of traditional model-based diagnosis (see Definition 3.1.2) is set union $\cup$. Hence, for the verdict oracle defined above, this means that we get the set of sets of fault classes that occurred on all the runs that may induce a given observation sequence:

$$\mathsf{V}(\omega) = \bigcup \{\, \mathsf{V}(\rho) \mid \rho \in \mathsf{Runs}(\omega) \,\}$$

Now, some fault of a fault class f certainly occurred iff it occurred on all those runs, i.e., iff $f \in F$ for all $F \in \mathsf{V}(\omega)$. Some fault of a fault class f possibly occurred iff it occurred on some of those runs, i.e., iff $f \in F$ for some $F \in \mathsf{V}(\omega)$. This is exactly in line with the traditional definitions [cf. Sam+95]. Thus, the theoretical framework presented in this chapter provides a conservative generalization of the core concepts of traditional model-based diagnosis.

3.5.2 LTL Runtime Verification

Recall Section 2.5.1, where we recapitulated LTL runtime monitoring [BLS07; BLS06b; BLS11]. An LTL_3 monitor for an LTL formula φ is a deterministic finite input-enabled transition system $\langle \mathcal{S}, I, \wp(\mathsf{AP}), \twoheadrightarrow \rangle$ together with a state labeling $\langle \mathbb{B}_3, \lambda \rangle$. As for diagnosers, we can easily cast an LTL_3 runtime monitor into a VTS

$$\langle \mathcal{S}, I, \wp(\mathsf{AP}), \twoheadrightarrow, \mathbb{B}_3, \sqsubseteq, \lambda \rangle$$

where $\sqsubseteq$ is the specificity order on $\mathbb{B}_3$ (see Figure 3.1b). The VTS constructed from an LTL_3 monitor is deterministic and monotonic.

In the context of online LTL runtime monitoring, an observation sequence $\omega \in \wp(\mathsf{AP})^\star$ is extended incrementally with new observations from the running system. To this end, a system is usually instrumented in some way to generate observations over the set of atomic propositions AP. In terms of the introduced concepts, such instrumentation may correspond to an observation model which translates the ongoing run of a system to an observation sequence over $\wp(\mathsf{AP})$.

Soundness and Completeness of Runtime Monitors. Traditional LTL runtime verification does not aim to account for the actual future behavior of a concrete system because the correct verdict only depends on the observation sequence and the LTL formula φ. Assuming that the transitions of a system model are labeled with sets of atomic propositions, we may define the following verdict oracle

$$\mathsf{V}(\rho) \;\coloneqq\; [\mathsf{Trace}(\rho) \vDash \varphi]^3_{\mathrm{LTL}} \tag{3.8}$$

based on the LTL_3 semantics as per (2.3). According to this verdict oracle, a run ρ is ascribed a truth verdict depending on whether all infinite continuations of its trace $\mathsf{Trace}(\rho)$ fulfill or violate the given LTL formula φ.

Bauer, Leucker, and Schallhart introduce two maxims for runtime verification: *impartiality* and *anticipation* [BLS07; DLS08]. "*Impartiality* requires that a finite trace is not evaluated to *true* or *false*, if there still exists an (infinite) continuation leading to another [truth] verdict. *Anticipation* requires that once every (infinite) continuation of a finite trace leads to the same [truth] verdict, then the finite trace evaluates to this [truth] verdict" [DLS08, p. 387]. Note that the LTL_3 semantics as per (2.3) adheres to both maxims. Impartiality and anticipation are closely related to soundness and completeness as per Definition 3.4.3 and Definition 3.4.4. With respect to the trace observation model and the verdict oracle as per (3.8), a monitor fulfilling the impartiality maxim is sound and a monitor fulfilling the anticipation maxim is complete for any system model. Thus, a monitor fulfilling both maxims indeed produces most specific truth verdicts and it does so *independently* of the system model. The algorithms presented by Bauer, Leucker, and Schallhart [BLS06b] construct such monitors.

These considerations show that the theoretical framework presented in this chapter also provides a conservative generalization of core concepts found in runtime verification, in particular, LTL runtime verification.

The assumptions made above are typical for LTL runtime verification where it is assumed that sets of atomic propositions can be observed and that the properties are directly expressed over the actually observed sets of atomic propositions. The framework presented here also allows relaxing these assumptions by using different observation models. In case the transitions of a system model are not labeled with sets of atomic propositions, the verdict oracle can also be defined via an additional function that transforms runs to sequences of sets of atomic propositions. Of course, while the theoretical framework allows modeling such scenarios, it does not provide any algorithms to actually construct monitors for them.

RV-LTL Monitoring. Recall that Bauer, Leucker, and Schallhart also introduce RV-LTL monitoring using the truth domain $\mathbb{B}_4 = \{\mathtt{t}, \mathtt{t}^p, \mathtt{f}^p, \mathtt{f}\}$ instead of $\mathbb{B}_3$ to deal with properties like $\square(\mathtt{request} \rightarrow \Diamond \mathtt{response})$ [BLS07]. To accommodate RV-LTL monitoring within the introduced framework, we need to introduce a fifth verdict ? to obtain the verdict domain $\mathbb{B}_5$ depicted in Figure 3.8.

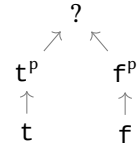


Figure 3.8: Verdict domain $\mathbb{B}_5 = \{\mathtt{t}, \mathtt{t}^p, \mathtt{f}^p, \mathtt{f}, ?\}$ for RV-LTL runtime monitoring.

Without the ? verdict, $\mathtt{t}^{\mathrm{p}}$ and $\mathtt{f}^{\mathrm{p}}$ would not have a least upper bound, which is required to form a verdict domain. Analogously to LTL_3, every RV-LTL monitor can be cast into a deterministic input-enabled VTS over the verdict domain $\mathbb{B}_5$. In contrast to LTL_3, this VTS may be non-monotonic since RV-LTL monitors may toggle between the verdicts $\mathtt{t}^{\mathrm{p}}$ and $\mathtt{f}^{\mathrm{p}}$. While adding ? is required to obtain a verdict domain, RV-LTL monitors constructed with the traditional techniques [BLS07] will never actually produce this verdict. The verdict ? does, however, become relevant when considering existing work on *predictive monitoring*.

In predictive monitoring, knowledge about a concrete system is exploited to produce truth verdicts that take a system's possible future behavior into account [ZLD12; Leu12; Pin+17a; Fer+21]. In particular, Pinisetty et al. consider predictive monitoring for regular timed properties expressible as deterministic timed automata [Pin+17a]. To this end, they also extend the RV-LTL domain with a fifth verdict ?. Here, the verdict ? indicates that the current observation sequence still lacks observations that are required to reach a more specific verdict [Pin+17a, p. 358]. This interpretation fits the specificity ordering on $\mathbb{B}_5$, where ? is the least specific verdict. The online monitoring algorithm developed by Pinisetty et al. implements a verdictor that can be modeled as a VTS observing timed events and producing $\mathbb{B}_5$ verdicts.

3.5.3 Stream-Based Runtime Monitoring

While traditional model-based fault diagnosis and LTL runtime monitoring are both automata-based approaches, stream-based monitoring [DAn+05; Con+18; Bau+20; GS18], is usually not based on automata theory. At least conceptually, we can still make sense of stream-based monitoring within the presented framework. To illustrate how stream-based monitoring fits into the picture, we use the stream-based specification language Lola as an example (recall Section 2.5). To other stream-based runtime verification techniques, similiar considerations apply.

Recall that stream-based monitoring is about computing output streams from input streams. Given a Lola specification $\mathtt{L}$, we will now work towards the definition of a VTS $\langle Q, J, \mathrm{Obs}, \rightarrowtail, \mathcal{V}, \sqsubseteq, \nu \rangle$ that does exactly that.

Let us start with the set Obs of observables. Lola is a *synchronous* stream-based specification language. This means, that the new values for the input streams arrive all at the same time, i.e., synchronously. Let $\{s_i\}_{i=1}^n$ be the family of independent stream variables (inputs) of the Lola specification $\mathtt{L}$ and $\{T_i\}_{i=1}^n$ be the family of their types. As values arrive synchronously, the set of observables is the Cartesian product of the types $\mathrm{Obs} = \times_{i=1}^n T_i$, assuming that a type is nothing more than the set of values that inhabit it. Thus, an observable $\langle v_1, \ldots, v_n \rangle$ contains a new value for all the input streams. The states Q of the VTS and its initial state is whatever the Lola online runtime monitoring algorithm requires. The transition relation is given by the algorithm itself that takes the new values for the input streams and extends

the output streams as necessary. It is not surprising that the algorithm can just be captured by a VTS as VTSs pose no restrictions on states or transitions, so we can simply use any algorithm to instantiate them. The interesting part is how we model the outputs of a Lola runtime monitor as verdicts.

Recall that output streams can have arbitrary data types and we may be interested in the values of those streams. If we are interested in the value of a single output stream of some type T, then we could use the verdict domain $\langle \wp(T) \setminus \{\varnothing\}, \subseteq \rangle$. Here, we may interpret each verdict $v \in \wp(T) \setminus \{\varnothing\}$ as a set of possible values the stream may have. Clearly, a verdict of that kind is less specific than another, if it considers less values possible. Another option could be to use the verdict domain $\langle T \uplus \{?\} \rangle$ where $? \notin T$ is a top verdict introduced to form a join-semilattice. If we are interested in values of multiple streams, then we can either use the product verdict domain or, if the streams share a verdict domain, join the individual verdicts. Recall that the product of complete join-semilattices is again a complete join-semilattice (see Chapter 2). This leaves us with some options to model stream values as verdicts.

It remains to define the verdict function ν of the VTS. The definition of the verdict function is complicated by the fact that output streams may lag behind. For an example, recall Example 2.7, where the current acceleration is computed based on the velocity one step into the future. Here, the acceleration stream is bound to lag one step behind the velocity stream. Thus, in general, we cannot simply produce the verdict corresponding to the current value of a stream as this value may still be unknown. A possible solution to this problem is to produce verdicts that lag similarly behind, i.e., always produce a verdict corresponding to the most-recent known value, and, if such a value does not exist, produce the top element of the verdict domain—which is always a safe over-approximation.

Boolean Lola Specifications. Bozzelli and Sánchez study Boolean stream-based monitoring from a language-theoretic perspective [BS14; BS16]. Their results show that the expressiveness of *Boolean Lola specifications* precisely coincides with the class of regular languages [BS14, p. 66]. In a Boolean Lola specification, all streams are defined over $\mathbb{B}_2 = \{\mathtt{t}, \mathtt{f}\}$, significantly reducing expressiveness.

While producing verdicts that lag behind (as discussed above) may be suitable for some applications, for others it is desirable to try to give a most specific over-approximation of the values a stream may have at the current instant in time. In the Boolean case, this means producing `t` or `f` as soon as they are inevitable as per the anticipation maxim of runtime verification [BLS07]. The original Lola monitoring algorithm does not do that, however, for Boolean Lola specifications Kallwies, Leucker, and Sánchez recently introduced *general anticipatory monitoring* [KLS23]. Their construction yields a Moore machine that produces a $\mathbb{B}_3$ verdict for each output stream such that `?` is indicated iff the current value of a Boolean output stream could be `t` or `f` depending on future inputs. Otherwise, `t` or `f` is indicated depending on

whether the current value will inevitably be `t` or `f`. While Kallwies, Leucker, and Sánchez model the outputs of the Moore machine as functions from stream variables to $\mathbb{B}_3$ truth verdicts, casting the Moore machine into a VTS is straightforward using a product verdict domain for multiple streams as explained above. Extending their approach to general Lola specifications is an interesting avenue for future work on stream-based runtime verification [cf. KLS23]. Verdict domains as a generalization of the Boolean setting and VTSs, may serve as a foundation for such work. Generalizing the anticipation maxim beyond the Boolean setting, would require that a most specific over-approximation of a stream's value is produced.

Soundness and Completeness. As for LTL runtime verification, for stream-based runtime verification, the correct verdict for a given sequence of observations depends only on that sequence and the used specification. It is independent of the system model and the concrete behavior of a system. Still, a verdict oracle can be defined analogously to (3.8) ascribing the correct verdict (with full anticipation) to a given run. Soundness and completeness then apply analogously and correspond to impartiality and anticipation, possibly generalized beyond the Boolean setting.

3.6 Discussion

In this chapter, we established an overarching theoretical framework upon which the subsequent contributions of this thesis built. With verdict domains, verdict transition systems, observation models, and verdict oracles, we introduced all the formal tools to precisely capture what it takes for a verdictor to produce accurate verdicts (FT1): Verdicts should at least be correct and preferably most specific. Technically, this requires verdictors to be sound and complete or Δ-complete.

The verdict domains introduced in Section 3.1 cover the broad range of operational questions discussed in Chapter 1. Verdicts can indicate the satisfaction or violation of properties (Q1), they can indicate the presence of faults (Q2), and they can indicate which configurations a system may have (Q3).

We have also shown that this framework can serve as a unifying foundation for existing work in the spectrum of runtime verification and model-based fault diagnosis (FT3). In general, runtime monitors or fault diagnosers can be cast into the framework if their outputs can be made to form a verdict domain, which should be possible for other approaches [e.g. MB06; AS15; CPS08; Bar+04; Car+13; Mha+17] as well, given the general nature of the theoretical framework and especially verdict domains. As such, the theoretical framework also serves to elucidate the connections between existing work and the contributions made by this thesis.

The benefit of the presented framework lies in the general concepts and the specifications it provides for the subsequent chapters. What it provides, is a frame-

work for modeling verdictors, the observations that are fed to them, as well as the verdicts that they should produce. Following the model-based methodology, we need these models in order to get provable guarantees about the verdicts produced by verdictors regarding a given system and its execution. Practical algorithms exploiting this foundation will be developed in the next two chapters. To this end, VTSs will also serve as a target representation for verdictor synthesis (FT2).

Part II

Generic Verdictor Algorithms

Chapter 4

Modular Discrete-Time Verdictor Synthesis

Enabled by the theoretical framework established in the previous chapter, we now present a generic and modular synthesis approach that solves several significant instances of the VTS Synthesis Problem in the discrete-time setting.

The approach takes the form of a pipeline synthesizing VTSs from system models whose states and transitions have been annotated with verdicts. As we will later show in Chapter 7 and Chapter 8, the pipeline is generic in the sense that it can be used for runtime verification, fault diagnosis, and configuration monitoring alike by choosing appropriate annotations. For presentation purposes, we focus on fault diagnosis in this chapter. The pipeline is also modular in the sense that its building blocks can be combined to synthesize VTSs tailored to the specific needs of an application, e.g., to account for different and orthogonal observational imperfections or to provide predictions based on a system's possible future behavior.

The techniques presented here constitute the first part of Contribution TT.

Synthesis Pipeline. Figure 4.1 depicts the synthesis pipeline. The pipeline takes a *verdict-annotated* finite TS $\mathfrak{S}$ as input and synthesizes an optimized (deterministic and minimal) verdictor implementation `I` from it. It consists of four stages: *annotation tracking*, *lookahead refinement*, *observability adjustment*, and *finalization*.

For the first stage of the pipeline, coined *annotation tracking*, we make the idealized assumption that all actions of the input model $\mathfrak{S}$ are observable. Consequently, this stage constructs a VTS that produces a verdict for each trace of the system model. This verdict *tracks* the annotations of the transitions which may have been taken to produce the respective trace and of the states the system may be in afterwards. As we will show, by choosing appropriate *verdict annotations*, this technique can be used to

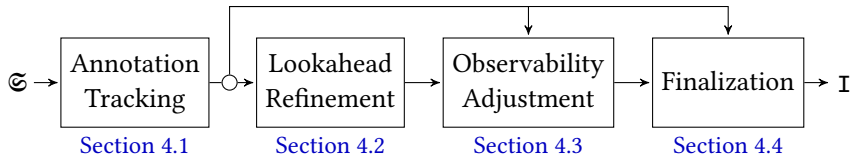


Figure 4.1: Generic and modular model-based VTS synthesis pipeline.

construct VTSs for runtime verification, fault diagnosis, and configuration monitoring alike. After annotation tracking, the *lookahead refinement* stage can be used to transform the VTS to produce most specific predictions concerning the inevitable future. This transformation is entirely optional and requires the additional assumption that the system indeed keeps running. Otherwise, there is no future to obtain predictions for. Note that this assumption is very natural for reactive systems. After annotation tracking and potentially lookahead refinement, the idealized assumption that every action is observable is lifted in the *observability adjustment* stage. This stage consists of VTS transformations accounting for limited observability, delays, losses, and out-of-order arrivals—as they are often unavoidable in practice (recall Chapter 1). Each transformation is optional and may be omitted in case the respective imperfection is not relevant for a certain application. Finally, in the *finalization* stage, our earlier determinization and minimization results (recall Section 3.2) are exploited to obtain a minimal and deterministic VTS optimized towards an efficient implementation in software or hardware. In addition, we also present a *language-relaxing minimization* algorithm which can further reduce the size of a VTS by over-approximating the language of the original VTS.

Notably, the individual algorithmic building blocks of the stages can also be used independently of the pipeline. For instance, a VTS synthesized with some third-party technique can be transformed to provide most specific predictions or to take into account certain observational imperfections. To facilitate such usage, we establish general theorems for the transformations we develop.

Relevant Publications. This chapter is primarily based on the following paper:

> [KDH24]: Maximilian A. Köhl, Clemens Dubslaff, and Holger Hermanns. "Configuration Monitor Synthesis". In: *Automated Technology for Verification and Analysis, ATVA 2024.*

While VTSs have also been part of that work, the greater theoretical framework with observation models and verdict oracles is a novel contribution of this thesis (see Chapter 3). The framing of the contributions in this chapter with these concepts is an unpublished contribution. Furthermore, the transformation for out-of-order arrivals extends the published work.

Chapter Structure. This chapter's structure mirrors the synthesis pipeline. Section 4.1, Section 4.2, and Section 4.3, develop the annotation tracking, lookahead refinement, and observability adjustment stage, respectively (see also Figure 4.1). Section 4.4 discusses the finalization stage and possible implementation techniques for the resulting verdictors. Section 4.5 concludes this chapter by summarizing its contributions and highlighting their wide range of applicability.

4.1 Model-Based Construction

We have previously seen that existing constructions for runtime verification and fault diagnosis can be cast into VTSs (recall Section 3.5). While those constructions are specific to their respective applications, we here present a fully generic construction, coined *annotation tracking*, based on verdict-annotated system models. Annotation tracking can be used for runtime verification, fault diagnosis, and configuration monitoring alike. An in-depth exploration of concrete applications will follow later in Chapter 7 and Chapter 8. Importantly, annotation tracking is not meant to supersede any of the existing techniques. Instead, it complements and extends them by providing a new and generic way to construct VTSs.

4.1.1 Verdict-Annotated System Models

Annotation tracking is based on system models whose transitions and states have been annotated with verdicts—we call such models *verdict-annotated*. Formally, a verdict annotation is a triple of functions assigning verdicts to the initial states, states, and transitions of a transition system modeling a system.

Definition 4.1.1 *Given a TS* $\mathfrak{S} = \langle S, I, \mathrm{Act}, \twoheadrightarrow \rangle$ *and a verdict domain* $\langle \mathcal{V}, \sqsubseteq \rangle$*, a verdict annotation is a triple* $\langle \kappa, \lambda, \gamma \rangle$ *of functions* $\kappa : I \to \mathcal{V}$*,* $\lambda : S \to \mathcal{V}$*, and* $\gamma : \twoheadrightarrow \to \mathcal{V}$ *assigning verdicts to the initial states, states, and transitions of* $\mathfrak{S}$*, respectively.*

Verdict annotations allow us to provide verdicts for when a system starts in a given state (function κ), for when it resides in a given state (function λ), and for when it takes a given transition (function γ). Based on these verdicts, we next define a generic verdict oracle and introduce two examples.

Verdict Oracle. We start with the definition of a generic verdict oracle for verdict-annotated models. Recall that verdict oracles serve as the ground truth for most specific verdicts (Section 3.4). A verdict oracle assigns a verdict to each run of a system. Given a verdict-annotated system model, we now define a verdict oracle that takes into account the annotations of all transitions that have been taken, the

final state the system is in, as well as the state the system started in. This definition will rely on the meet of all these annotations, combining the annotations into the least specific verdict that is at least as specific as the individual verdicts. As verdict domains are join-semilattices, the meet of arbitrary sets of verdicts may be undefined. We will deal with those cases separately later and assume for the following definition that the verdict domain has been extended with a *sentinel bottom verdict* #.

Definition 4.1.2 *Given a TS* $\mathfrak{S} = \langle S, I, \mathrm{Act}, \twoheadrightarrow \rangle$*, a verdict domain* $\langle \mathcal{V}, \sqsubseteq \rangle$*, and a verdict annotation* $\langle \kappa, \lambda, \gamma \rangle$*, we define the* annotation verdict oracle

$$\mathsf{V}_{\kappa,\lambda,\gamma} : \mathsf{Runs}(\mathfrak{S}) \to \mathcal{V} \cup \{\#\}$$

over the verdict domain $\langle \mathcal{V}, \sqsubseteq \rangle$ *extended with the sentinel bottom verdict* $\# \notin \mathcal{V}$ *such that for all* $\rho = \left(\langle s_i, \alpha_i, s_i' \rangle\right)_{i=1}^{n} \in \mathsf{Runs}(\mathfrak{S})$*:*

$$\mathsf{V}_{\kappa,\lambda,\gamma}(\rho) \coloneqq \begin{cases} \bigsqcup \{ \kappa(s) \sqcap \lambda(s) \mid s \in I \} & \text{if } n = 0 \\ \kappa(s_0) \sqcap \left(\bigsqcap \{ \gamma(\mathrm{t}) \mid \langle \cdot, \mathrm{t} \rangle \in \rho \} \right) \sqcap \lambda(s_n') & \text{otherwise} \end{cases}$$

For the empty run, the verdict oracle $\mathsf{V}_{\kappa,\lambda,\gamma}$ returns the most specific verdict subsuming all verdicts of the initial states, i.e., the join of these verdicts. For non-empty runs, it returns the least specific verdict that is at least as specific as the verdicts of all the transitions, the verdict of the initial state, and the verdict of the last state, i.e., the meet of these verdicts. Intuitively, the join combines possibilities, i.e., if multiple verdicts are possible, then their join is the most specific verdict subsuming those possibilities (cf. Section 3.2). In Boolean terms, it corresponds to a disjunction. In contrast, the meet can be seen as combining necessities, i.e., if multiple verdicts are necessary, then their meet is the least specific verdict combining those necessities. In Boolean terms, it corresponds to a conjunction. For concrete examples, we refer the reader to the upcoming examples and Chapter 7 as well as Chapter 8.

As a verdict domain is merely a complete join-semilattice, not a complete lattice, the meet of arbitrary sets of verdicts may not be defined. Hence, we need to introduce the sentinel bottom verdict # thereby completing the lattice. The sentinel verdict is mostly a technical trick as it can or even has to be ignored for all the practical applications we consider in this thesis. Either the meet is in fact guaranteed to be defined for all verdict sets that may appear due to other structural properties of the annotations, or any run with the sentinel verdict can be disregarded as unrealistic. Note that this does not mean that the verdict domains are complete lattices already. In fact, the only application where this is the case is fault diagnosis.

Verdict-annotated system models and their corresponding verdict oracles are highly expressive, since the verdict oracle as per Definition 4.1.2 can incorporate both, information from the present state but also information from the entire history

of the run and the initial state. From a verdict-annotated system model, annotation tracking, explained below, will construct a VTS that is sound and complete with respect to the verdict orale $\mathsf{V}_{\kappa,\lambda,\gamma}$.

Example: State Belief Sets. In planning, it is common to base decisions on *beliefs* regarding the present state of a system [RN10]. For instance, for the standard model of planning under uncertainty, *partially-observable Markov decision processes* (POMDPs), a distribution over states may be tracked [KLC98; Thr02] to inform an agent aiming to maximize a reward [MHC99]. While verdict domains are not quantitative, the support of these distributions, i.e., the set of states that have a non-zero probability, can be seen as an element of the verdict domain $\langle \wp(\mathcal{S}) \setminus \{\varnothing\}, \subseteq \rangle$. Here, each verdict indicates in which states the system may possibly be in. Hence, a verdict v_1 is more specific than a verdict v_2 iff v_1 considers less states possible than v_2, i.e., iff $v_1 \subsetneq v_2$. In the literature, such sets of states are also referred to as *belief states* [RN10] or *state belief sets* [JJS21]. They are known to be sufficient for some applications, e.g., for ensuring almost-sure reachability objectives [JJS21].

Remark. The empty set is indeed not a sensible verdict as the system has to be in some state. This coincides with the fact that the support of any probability distribution over a non-empty set cannot be empty. As a result, the meet of non-overlapping sets is undefined and the verdict domain is not a lattice.

Given that it is useful for planning purposes to know in which states the system may be in, we want to synthesize a VTS over the aforementioned verdict domain, producing verdicts that indicate exactly the states the system may be in. To this end, consider the following verdict annotation over $\langle \wp(\mathcal{S}) \setminus \{\varnothing\}, \subseteq \rangle$:

$$\kappa(s) \coloneqq \mathcal{S} \qquad \lambda(s) \coloneqq \{s\} \qquad \gamma(\mathrm{t}) \coloneqq \mathcal{S} \tag{4.1}$$

Given this verdict annotation, the verdict oracle as per Definition 4.1.2 assigns the set of initial states to the empty run and the singleton set with the last state of the run to each run, respectively. For the empty run, we have:

$$\begin{aligned} & \mathsf{V}_{\kappa,\lambda,\gamma}(\epsilon) \\ = \; & \bigsqcup \{ \kappa(s) \sqcap \lambda(s) \mid s \in I \} \\ = \; & \bigcup \{ \mathcal{S} \cap \{s\} \mid s \in I \} \\ = \; & I \end{aligned}$$

For non-empty runs $\rho = \left(\langle s_i, \alpha_i, s'_i \rangle\right)_{i=1}^{n}$, we have:

$$\begin{aligned}
&\mathsf{V}_{\kappa,\lambda,\gamma}\Big(\left(\langle s_i, \alpha_i, s'_i \rangle\right)_{i=1}^{n}\Big) \\
&= \kappa(s_0) \sqcap \Big(\bigsqcap \big\{\, \gamma\big(\langle s_i, \alpha, s'_i \rangle\big) \mid 1 \leq i \leq n \,\big\}\Big) \sqcap \lambda(s'_n) \\
&= \mathcal{S} \cap \Big(\bigcap \big\{\, \mathcal{S} \mid 1 \leq i \leq n \,\big\}\Big) \cap \{s'_n\} \\
&= \{s'_n\}
\end{aligned}$$

Thus, the verdict assigned by the verdict oracle is the set of states the system may currently be in, assuming that the system can be in any of the initial states initially. As the join on the verdict domain $\langle \wp(\mathcal{S}) \setminus \{\varnothing\}, \subseteq \rangle$ is set union $\cup$, the most specific verdict as per (3.7) for a given observation sequence is simply the set of all states the system may be in after generating the respective observation sequence:

$$\mathsf{V}(\omega) \;=\; \bigcup \big\{\, \mathsf{V}_{\kappa,\lambda,\gamma}(\rho) \,\big|\, \rho \in \mathsf{Runs}(\omega) \,\big\}$$

Therefore, a sound and complete VTS for the verdict oracle as per Definition 4.1.2 based on the verdict annotation as per (4.1) produces most specific verdicts regarding the present state of the system. As we have shown above, the verdict oracle also never returns the sentinel verdict as the system must be in some state.

Example: Fault Diagnosis. We have previously seen that diagnosers constructed with traditional model-based fault diagnosis techniques [Sam+95; Sam+96] can be cast into our theoretical framework (recall Section 3.5.1). Furthermore, we will show in Section 7.2 that traditional diagnosers can also be obtained as a special case with our synthesis pipeline starting with a verdict-annotated model.

For now, let us continue based on Example 3.1: Given the set Faults of fault actions partitioned into a set $\mathcal{F}$ of fault classes, we may annotate the states and transitions of a system model with sets of fault classes that are present in those states or that need to be present in order to take a certain transition. These annotations are elements of the verdict domain $\langle \wp(\mathcal{F}), \supseteq \rangle$ (recall Definition 3.1.3). This verdict domain is a complete lattice and its meet operation $\sqcap$ is set union $\cup$. Hence, with such annotations in place, the verdict oracle as per Definition 4.1.2 collects all the faults that occurred on any of the taken transitions, those of the initial state, and those present in the last state of the system. The resulting verdict oracle is a more general and powerful variant of the verdict oracle defined in Example 3.7 for diagnosis of certain faults, as the annotations allow modeling different kinds of faults:

- The function κ models faults that are already present when the system starts in a certain state and the system cannot recover from them. We call such faults *primordial faults*.
- The function λ models *transient faults* that are only present when the system is in certain states and that go away when the system leaves those states.

– The function γ models *irrecoverable faults* that occur when a certain transition is taken and the system cannot recover from them.

Irrecoverable faults are also called *permanent* in the literature. For a taxonomy of different kinds of faults, we refer to Avizienis et al. [Avi+04]. Avizienis et al. also consider faults that occur as part of the development or deployment process. While they do not call them primordial, primordial faults may correspond to manufacturing errors or occur when a system is deployed, leaving it in a faulty initial state.

In contrast to annotations as just discussed, traditional model-based diagnosis can only handle irrevocable faults [Sam+95]. Models as they are used for traditional model-based diagnosis can be easily transformed into verdict-annotated models. To this end, states are annotated with the empty set and transitions are annotated with the empty set or a singleton set with the respective fault class, depending on whether a transition has been labeled with a fault action or not. Formally, that is:

$$\kappa(s) \coloneqq \varnothing \qquad \lambda(s) \coloneqq \varnothing \qquad \gamma(\langle s, \alpha, s' \rangle) \coloneqq \begin{cases} \{f\} & \text{if } \alpha \in f \\ \varnothing & \text{otherwise} \end{cases}$$

Hence, verdict annotations generalize the fault model of traditional model-based diagnosis.

Example 4.1 Figure 4.2 depicts a verdict-annotated variant of the original model of the coffee machine (recall Figure 2.1). The respective verdict domain is depicted in Figure 3.2. Both fault transitions are annotated with respective singleton sets. All other transitions and the states themselves are annotated with the empty set. The state annotations have been omitted in the graphical representation. A sound and complete VTS for the verdict oracle as per Definition 4.1.2 and this verdict annotation allows diagnosing faults that certainly occurred.

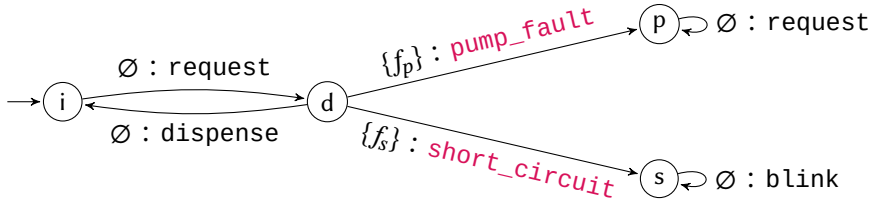


Figure 4.2: Verdict-annotated model of the coffee machine (cf. Figure 2.1).

Relation to Lattice Automata. The careful reader may have noticed that verdict-annotated system models closely resemble lattice automata (see Section 2.2.3). Like lattice automata, they require three functions, two of them assigning verdicts to states and a third assigning verdicts to transitions. Indeed, lattice automata and

their wide range of applications [cf. KL07] serve as the inspiration here. There are, however, two technical differences: First, verdict annotations assign verdicts that are not necessarily elements of a lattice. Second, in contrast to the functions Q_0 and δ as they appear in the definition of lattice automata, κ and γ do not assign verdicts to *all* states and *all* state-action-state triples, but instead only to initial states and transitions, respectively. They annotate an existing transition system encoding some behavior with initial states and transitions. In contrast, lattice automata focus on L-languages necessitating a lattice element for each word (see Section 2.2.3). So, the primary reason to deviate from lattice automata here is that we aim at an *annotative approach* where a transition system independently models the behavior of a system and is then annotated with additional information in the form of verdicts. This annotative approach is commonly pursued for configurable systems [KAK08; Cla+10; Dub19].

4.1.2 Annotation Tracking

With annotation tracking, we now present a construction to obtain a sound and complete VTSs from a verdict-annotated system model under the idealized assumption that all actions can be observed. In technical terms, this assumption entails that the observation model is Ω_{Trace}. This assumption will later be lifted in Section 4.3.

To construct such a VTS from a verdict-annotated model, we exploit results from lattice automata. Given the verdict domain $\langle \mathcal{V}, \sqsubseteq \rangle$, let $\langle \mathcal{V}^{\#}, \sqsubseteq^{\#} \rangle$ denote its extension with the bottom sentinel verdict #. The construction proceeds by first constructing a lattice automaton over $\langle \mathcal{V}^{\#}, \sqsubseteq^{\#} \rangle$, then applying simplification to obtain a simple lattice automaton [KL07, Theorem 6], and finally converting the simple lattice automaton to a VTS according to Lemma 3.2.1. We amalgamate those steps into a single construction:

Definition 4.1.3 *Given a TS $\mathfrak{S} = \langle S, I, \text{Act}, \twoheadrightarrow \rangle$, the verdict domain $\langle \mathcal{V}, \sqsubseteq \rangle$, and verdict annotation $\langle \kappa, \lambda, \gamma \rangle$, we define*

$$\mathfrak{V}^{\#}_{\mathfrak{S},\kappa,\lambda,\gamma} = \left\langle S \times \mathcal{V}, \text{Act}, \{ \langle s, \kappa(s) \rangle \mid s \in I \}, \rightarrow, \mathcal{V}^{\#}, \sqsubseteq^{\#}, \nu \right\rangle$$

where $\nu(\langle s, v \rangle) := v \sqcap \lambda(s)$ and $\langle \langle s, v \rangle, \alpha, \langle s', v' \rangle \rangle \in \rightarrow$ iff there exists a transition $\langle s, \alpha, s' \rangle \in \twoheadrightarrow$ and $v' = v \sqcap \gamma(\langle s, \alpha, s' \rangle)$.

Like simplification of lattice automata [KL07], the underlying principle here is to push the meet over the verdicts of the taken transitions into the state space. To be able to carry out this construction explicitly and use it as part of our synthesis pipeline, the system model needs to be finite.

We will now establish results regarding the tightness of the resulting VTS with respect to the system model, and, most importantly, the soundness and completeness

of the VTS with respect to the verdict oracle as per Definition 4.1.2. Furthermore, we discuss the complexity of the construction. This is followed by an example of the construction for fault diagnosis.

Tightness. The following VTS transformations for observational imperfections and predictions require that VTSs are tight, i.e., capture exactly the possible behavior of the system (recall Definition 3.3.6). Indeed, the VTS $\mathfrak{V}^{\#}_{\mathfrak{S},\kappa,\lambda,\gamma}$ is tight.

Proposition 4.1.1 *The VTS $\mathfrak{V}^{\#}_{\mathfrak{S},\kappa,\lambda,\gamma}$ is tight with respect to $\mathfrak{S}$ and Ω_{Trace}.*

Proof. The observable language of $\mathfrak{S}$ with respect to Ω_{Trace} is its set of traces, i.e., $\mathcal{L}|_{\Omega_{\text{Trace}}}(\mathfrak{S}) = \mathcal{L}(\mathfrak{S})$. The languages of $\mathfrak{V}^{\#}_{\mathfrak{S},\kappa,\lambda,\gamma}$ and $\mathfrak{S}$ are identical, as the transition structure is inherited. Hence, $\mathfrak{V}^{\#}_{\mathfrak{S},\kappa,\lambda,\gamma}$ is tight with respect to $\mathfrak{S}$ and Ω_{Trace}. □

As a result of Proposition 4.1.1, the VTS $\mathfrak{V}^{\#}_{\mathfrak{S},\kappa,\lambda,\gamma}$ produces a verdict for every trace $\sigma \in \mathcal{L}(\mathfrak{S})$ while not accepting words that are not traces of $\mathfrak{S}$.

Soundness and Completeness. While tightness is an important property, the whole point of annotation tracking is that it produces a sound and complete VTS based on the verdict annotations and with respect to the verdict oracle as per Definition 4.1.2 under the idealized assumption that all actions can be observed.

Theorem 4.1.1 *The VTS $\mathfrak{V}^{\#}_{\mathfrak{S},\kappa,\lambda,\gamma}$ is sound and complete with respect to the system model $\mathfrak{S}$, the observation model Ω_{Trace}, and the verdict oracle $\mathsf{V}_{\kappa,\lambda,\gamma}$.*

Proof Sketch. Theorem 4.1.1 is proven based on the fact that the states reached after some trace σ in $\mathfrak{V}^{\#}_{\mathfrak{S},\kappa,\lambda,\gamma}$ correspond to the following set:

$$\left\{\left\langle s', \bigsqcap \{\gamma(t) \mid t \in \rho\}\right\rangle \text{ for some } \rho \in \mathsf{Runs}(\mathfrak{S}) \text{ s.t. } \mathsf{Trace}(\rho) = \sigma\right\}$$

The proof proceeds by proving this correspondence by induction on the length of the trace. For a detailed proof of Theorem 4.1.1, see Appendix A.1.1. □

As annotation tracking gives us a tight, sound, and complete VTS for the verdict oracle as per Definition 4.1.2 and the observation model Ω_{Trace}, we can, e.g., use it to construct a diagnoser based on fault annotations as discussed previously and under the idealized assumption that all actions can be observed.

Complexity. Annotation tracking inherits its complexity from lattice automata simplification [KL07]. The construction leads to a size increase over the system model that is linear in the size of the lattice, i.e., the size increase of the reachable fragment of the resulting VTS lies in $\mathcal{O}(|\mathcal{Q}| \cdot |\mathcal{V}|)$. The worst-case time complexity of

the construction is $\mathcal{O}(|\mathcal{Q}| \cdot |\mathcal{V}| \cdot D \cdot \mathsf{LOpCost}(D))$ where D is the maximal outdegree of $\mathfrak{S}$, i.e., the maximal number of transitions leaving a state. Recall that $\mathsf{LOpCost}(D)$ is the complexity of computing the join/meet over D verdicts.

Example: Fault Diagnosis. In the case of fault diagnosis, the states and transitions of a model are annotated with sets of fault classes (cf. Section 4.1.1). For such models, a VTS constructed by annotation tracking produces most specific diagnoses indicating classes of which faults certainly occurred as per Definition 3.1.3.

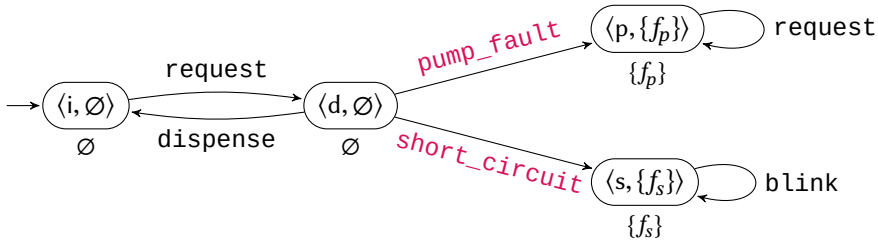


Figure 4.3: VTS constructed from the fault-annotated model of the coffee machine (see Figure 4.2) by annotation tracking.

Example 4.2 Recall Figure 4.2 which depicts a verdict-annotated variant of the coffee machine model (cf. Figure 2.1) for diagnosis purposes. Taking this model with the verdict annotations as a basis, annotation tracking constructs the VTS depicted in Figure 4.3. The verdict assigned to each state is shown next to the state, respectively. This VTS produces diagnoses under the idealized assumption that every action can be observed. Of course, traditionally fault actions are considered unobservable [Sam+95]. We address observational imperfections, such as limited observability, by transforming the VTS in later stages of the pipeline (cf. Figure 4.1).

Sentinel Pruning. As a post-processing step after annotation tracking, we introduce *sentinel pruning*. Recall that the sentinel verdict has been introduced such that the verdict oracle as per Definition 4.1.2 is well-defined. For fault diagnosis, a VTS constructed with annotation tracking actually never produces the sentinel verdict as both diagnosis verdict domains (see Definition 3.1.2 and Definition 3.1.3) are lattices. In general, however, annotation tracking may produce states with the sentinel verdict. For all the applications we consider in this thesis, the sentinel verdict will either not be produced or the states producing it must be discarded as they correspond to unrealistic runs—after all, there is a reason why the verdict domain is not a lattice. For instance, the verdict domain for configuration monitoring as per Definition 3.1.4 is not a lattice as it lacks the empty set. The empty set is not included, as the system is assumed to have some configuration that is among the valid configurations. Now,

to adjust a VTS constructed with annotation tracking in that regard, we remove the states of $\mathfrak{V}^{\#}_{\mathfrak{S},\kappa,\lambda,\gamma}$ that have the sentinel verdict and all transitions that involve such states. We refer to this removal as *sentinel pruning*. The resulting sentinel pruned VTS, denoted by $\mathfrak{V}_{\mathfrak{S},\kappa,\lambda,\gamma}$, still produces most specific verdicts for all realistic traces while not accepting unrealistic traces. We omit the formal exposition of this, as it is straightforward.

4.2 Most Specific Predictions

Being able to identify potential issues early can be highly valuable. For instance, in the case of industrial automation (recall Chapter 1), it is critical to identify potential issues before they escalate to severe problems and bring down production. To identify potential issues early, we develop a VTS transformation that refines verdicts of monotonic states by taking into account future system behaviors starting in each state, respectively. This transformation, coined *lookahead refinement*, transforms a VTS such that it produces *most specific predictions*.

For optimal results, lookahead refinement requires the input VTS to be tight. Furthermore, it comes with the additional assumption that the system indeed keeps running, as otherwise there is no future to obtain predictions for.

At the core of lookahead refinement is a fixpoint construction looking ahead as far as possible. To this end, given a finite input VTS $\mathfrak{V} = \langle Q, J, \mathrm{Obs}, \rightarrowtriangle, \mathcal{V}, \sqsubseteq, \nu \rangle$, we define a *lookahead-refined verdict function* ν_i recursively for all $i \in \mathbb{N}$:

$$\nu_0(q) := \nu(q) \qquad \nu_{i+1}(q) := \begin{cases} \bigsqcup_{q' \in \mathrm{Post}(q)} \nu_i(q') & \text{if } q \text{ is monotonic} \\ \nu(q) & \text{otherwise} \end{cases} \tag{4.2}$$

That is, ν_{i+1} refines the verdict of each monotonic state q by joining the verdicts of q's successors from the previous iteration ν_i. Note that ν_i reaches a fixpoint after at most $|Q|$ iterations, ensuring that verdicts have propagated from a state to all its monotonic predecessors. In essence, potential future verdicts are pushed forward to their predecessors as far as possible, i.e., until a fixpoint is reached. Using this fixpoint, we obtain the *lookahead refined* VTS $\lfloor\mathfrak{V}\rfloor$.

Definition 4.2.1 *Given a finite input VTS $\mathfrak{V} = \langle Q, J, \mathrm{Obs}, \rightarrowtriangle, \mathcal{V}, \sqsubseteq, \nu \rangle$, we define its lookahead refined transformation $\lfloor\mathfrak{V}\rfloor$ by*

$$\lfloor\mathfrak{V}\rfloor := \langle Q, J, \mathrm{Obs}, \rightarrowtriangle, \mathcal{V}, \sqsubseteq, \nu_{|Q|} \rangle$$

where $\nu_{|Q|}$ is the fixpoint of ν_i as per (4.2).

It is easy to see that $\lfloor\mathfrak{V}\rfloor$ indeed refines $\mathfrak{V}$.

Lemma 4.2.1 *For any finite VTS $\mathfrak{V}$, we have $\lfloor\mathfrak{V}\rfloor \trianglelefteq \mathfrak{V}$.*

Proof. Recall from Definition 3.2.3 that a state q is monotonic iff $\nu(q') \sqsubseteq \nu(q)$ for all its successors $q' \in \mathrm{Post}(\{q\}, \mathrm{Obs})$. Observe that $\nu_{i+1}(q) \sqsubseteq \nu_i(q)$ for all states $q \in Q$ since iterating ν_i only refines monotonic states. Hence, $\nu_{\lfloor\mathfrak{V}\rfloor}(\omega) \sqsubseteq \nu_{\mathfrak{V}}(\omega)$ for all $\omega \in \mathcal{L}(\mathfrak{V})$ satisfying condition (RE2) of Definition 3.2.5. Furthermore, $\mathcal{L}(\mathfrak{V}) = \mathcal{L}(\lfloor\mathfrak{V}\rfloor)$ as the transitions remain the same satisfying condition (RE1). Therefore, $\lfloor\mathfrak{V}\rfloor \trianglelefteq \mathfrak{V}$. □

For the same reasons as to why Lemma 4.2.1 holds, lookahead refinement also preserves monotonicity, i.e., $\lfloor\mathfrak{V}\rfloor$ is monotonic iff $\mathfrak{V}$ is monotonic.

Example 4.3 Figure 4.4 shows an example of lookahead refinement for a diagnoser over the verdict domain as per Definition 3.1.3, with a fragment of the input VTS on the left and its refined version on the right. Here, $\varnothing$ is refined to $\{f_2\}$. Assuming that the system continues running, we know that either faults of the classes $\{f_1, f_2\}$ or $\{f_2, f_3\}$ will be diagnosed since α or β will inevitably be observed, producing verdicts $\{f_1, f_2\}$ or $\{f_2, f_3\}$, respectively. This leads to a refinement of the verdict $\varnothing$ to $\{f_2\}$.

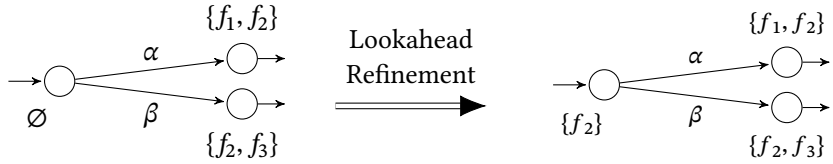


Figure 4.4: An example of lookahead refinement of certain fault class verdicts.

Complexity. Lookahead refinement iterates ν_i at most $|Q|$ times towards a fixpoint and computes the join over at most D verdicts for each state in each iteration, where D is the maximal outdegree of $\mathfrak{V}$. Checking monotonicity can be done in $\mathcal{O}(D \cdot |Q|)$ with $D \leq |Q|$. As monotonicity is preserved in each iteration, it suffices to check it once in the beginning. Hence, the worst-case time complexity lies in $\mathcal{O}(\mathrm{LOpCost}(D) \cdot |Q|^2)$. Recall that $\mathrm{LOpCost}(D)$ is the complexity of computing the join of D verdicts.

Specificity of Predictions. Modifying verdicts only along monotonic states guarantees that verdicts only get more specific. This is the reason behind Lemma 4.2.1 and also allows us to obtain the following corollary for completeness:

Corollary 4.2.1 *If $\mathfrak{V}$ is complete with respect to a system model $\mathfrak{S}$, a verdict oracle V, and an observation model Ω, then $\lfloor\mathfrak{V}\rfloor$ is also complete with respect to the same system model, verdict oracle, and observation model.*

While lookahead refinement preserves completeness, it does not preserve soundness with respect to the original verdict oracle. This is rooted in the very nature of how we defined soundness: Soundness requires that the verdict produced by a VTS is at most as specific as the verdict ascribed by the verdict oracle to each possible run, respectively. Hence, non-trivial predictions that are more specific than the original verdict may not be sound. This does, however, not mean that they are incorrect. If $\mathfrak{V}$ is sound, then $\lfloor\mathfrak{V}\rfloor$ produces verdicts that are eventually correct, i.e., they are correct with respect to some point in time in the future, as expected for predictions. To restore soundness for predictions, a different verdict oracle is required that takes the future into account. We have already seen an example of such a verdict oracle for LTL runtime verification with perfect anticipation (see Section 3.5.2).

For most specific predictions, it is required that the input VTS is tight with respect to the system model. A VTS that is not tight accounts for behavior that cannot actually be realized by a given system. While this is irrelevant for producing verdicts, this spurious behavior may lead to less specific predictions. Furthermore, it is important that lookahead refinement is performed before accounting for any observational imperfections to make sure that it has as much information as possible.

Lookahead refinement is related to the anticipation maxim of runtime verification [BLS07; DLS08]: Lookahead refinement refines verdicts by anticipating possible future observations. In fact, in the Boolean case, lookahead refinement will propagate inevitable `t` and `f` verdicts along monotonic states. Furthermore, lookahead refinement generalizes beyond the Boolean case and truth verdicts. For instance, it can also be used for diagnosis, as we have seen in Example 4.3. Note that lookahead refinement does, however, restrict anticipation to information that is already encoded in the input VTS. As such, it cannot replace specialized construction techniques for anticipatory monitors that do exploit information encoded in some other form, e.g., a Lola specification [KLS23] or an LTL property [BLS07].

To summarize, given a sound, complete, and tight input VTS $\mathfrak{V}$, the lookahead refined VTS $\lfloor\mathfrak{V}\rfloor$ is complete, as verdicts only get more specific, and tight, as the transition structure does not change. It also produces most specific verdicts that are guaranteed to be eventually correct, i.e., it will produce most specific predictions based on the information about possible future observations and verdicts encoded in the input VTS $\mathfrak{V}$.

4.3 Imperfect Observations

Hitherto, we made the idealized assumption that all actions of the system model are observable, and that they are observed exactly once in the order they occurred. However, as already established in the introduction of this thesis, observational imperfections are usually unavoidable. Naturally, when observations are subject to such imperfections, we still want most specific verdicts. In other words, we want verdictors to be robust with respect to these imperfections as much as possible. To this end, we present modular VTS transformations for making VTSs robust to four important and typically unavoidable of observational imperfections: *limited observability*, *delays*, *losses*, and *out-of-order arrivals*.

Recall that limited observability concerns inherent limitations as to what can be observed. Limited observability is often rooted in resource constraints. More observations require more bandwidth in shared networks, deploying additional sensors in a physical system can be expensive, and additional logging to a software system usually harms performance. Delays and losses are necessarily introduced by shared networks between components, e.g., on a manufacturing floor. In the context of industrial automation, the deployed network stacks usually still give some guarantees in terms of bounds on the delay with which a message may arrive and on the number of consecutive messages that may get lost [Fel05; TV99; Di +12; THW94]. In the following, we exploit such guarantees by considering *bounded* delays and losses, in addition to *unbounded* delays and losses. If delays are bounded, then so are possible out-of-order arrivals of observations. Hence, these guarantees can often also be exploited to obtain bounds on possible out-of-order arrivals.

Modular Transformations. Different observational imperfections are usually orthogonal. That is, there can be any combination of limited observability, delays, and losses. Furthermore, it is desirable to be able to flexibly integrate other imperfections in the future. Hence, we take a modular approach to accommodate observational imperfections by VTS transformations. The approach is as follows: Observational imperfections can be characterized by observation model transformers (cf. Section 3.3.1). Now, given an observation model transformer Θ characterizing an observational imperfection and a finite VTS $\mathfrak{V}$ that is sound, complete, and tight with respect to a given system model $\mathfrak{S}$, verdict oracle V, and observation model Ω, we aim to transform $\mathfrak{V}$ such that the transformation is sound, complete, and tight with respect to the same system model $\mathfrak{S}$, the same verdict oracle V, and transformed observation model $\Theta(\Omega)$. Figure 4.5 visualizes the relation between the respective objects. Here, Ξ denotes a *VTS transformer* accounting for the observational imperfections characterized by Θ. This approach allows us to chain the different transformations in order to account for multiple observational imperfections. The correctness of the whole chain then simply follows from the correctness of the individual transformations.

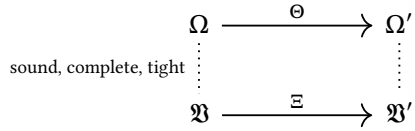


Figure 4.5: VTS transformations accounting for observational imperfections.

Notably, we also want the transformations to be applicable to predictions and to third-party techniques, which do not necessarily produce sound, complete, and tight VTSs. To this end, we prove more general correspondence theorems for the VTS transformations we present in the following. As corollaries of these theorems, we then also obtain that they allow the described modular approach.

In the following, we assume the finite input VTS $\mathfrak{V} = \langle Q, J, \mathrm{Obs}, \rightarrowtriangle, \mathcal{V}, \sqsubseteq, \nu \rangle$ to be implicitly given, unless otherwise indicated.

4.3.1 Limited Observability

As motivated above and in the introduction, in real-world scenarios, there are inherent observability limits. Formally, we can capture those limits by restricting the set of observables Obs of the input VTS $\mathfrak{V}$ to a subset $\mathrm{O} \subseteq \mathrm{Obs}$. We already introduced the respective observation model transformer in Definition 3.3.4. We now present a VTS transformation, coined *observability projection*, that constructs a new VTS such that the set of observables is restricted from Obs to O.

Remark. Conceptually, observability projection shares similarities with *hiding* or *restriction* operators found in typical process calculi [e.g. Mil80; Hoa78]. However, as VTSs have no concept of internal τ*-transitions*, i.e., every transition must correspond to an observable, we have to directly remove transitions with unobservable actions instead of replacing them with τ-transitions.

The idea behind observability projection is as follows: If certain events become unobservable, then we may take the respective transitions in the VTS without an observation. Hence, similarly to lookahead refinement, we must look an unbounded number of transitions in the future. However, unlike lookahead refinement, we must now consider only transitions that became unobservable. To this end, we again use a fixpoint construction. Let $X_i(q)$ be the set of states reachable from $q \in Q$ by taking at most $i \in \mathbb{N}$ unobservable transitions:

$$X_0(q) \coloneqq \{q\} \qquad X_{i+1}(q) \coloneqq X_i(q) \cup \mathrm{Post}\big(X_i(q), \mathrm{Obs} \backslash \mathrm{O}\big) \tag{4.3}$$

It is easy to see that X_i reaches a fixpoint after at most $|Q|$ iterations.

When the input VTS is in a given state q but events outside of O became unobservable, then any of the states in $X_{|Q|}(q)$ must be considered possible as one can get

there without making any observations. Observability projection then updates the transitions and verdict function accordingly.

Definition 4.3.1 *Given a VTS* $\mathfrak{V} = \langle \mathcal{Q}, J, \mathrm{Obs}, \rightarrowtriangle, \mathcal{V}, \sqsubseteq, \nu \rangle$ *and a set* $\mathrm{O} \subseteq \mathrm{Obs}$ *of observable actions, we define the observability projection of* $\mathfrak{V}$ *by*

$$\mathfrak{V}{\restriction}_{\mathrm{O}} := \langle \mathcal{Q}, J, \mathrm{O}, \rightarrowtriangle', \mathcal{V}, \sqsubseteq, \nu' \rangle \quad \textit{with} \quad \nu'(q) := \bigsqcup\nolimits_{q' \in X_{|\mathcal{Q}|}(q)} \nu(q') \tag{4.4}$$

and $\langle q, \mathrm{o}, q'' \rangle \in \rightarrowtriangle'$ *iff* $\langle q', \mathrm{o}, q'' \rangle \in \rightarrowtriangle$ *for some state* $q' \in X_{|\mathcal{Q}|}(q)$*. Here,* $X_{|\mathcal{Q}|}$ *is the fixpoint of* X_i *as per* (4.3).

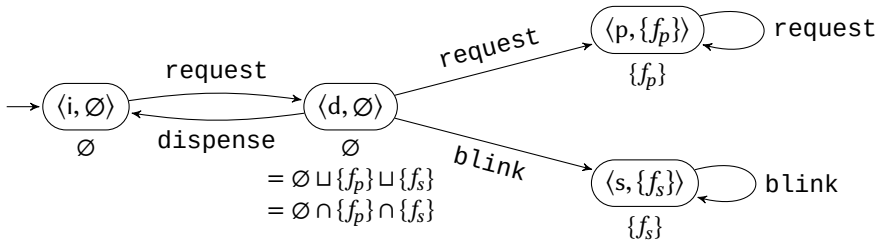


Figure 4.6: Observability projection of the VTS constructed by annotation tracking from the model of the coffee machine (see Figure 4.3).

Example 4.4 Figure 4.6 depicts the observability projection of the VTS constructed by annotation tracking from the verdict-annotated model of the coffee machine (see Figure 4.3). Here, we assumed that all actions except the fault actions are observable, i.e., $\mathrm{O} = \{\texttt{request}, \texttt{dispense}, \texttt{blink}\}$. The resulting VTS allows diagnosing faults of the coffee machine. It is equivalent to the VTS depicted in Figure 3.4, which we presented as an example for a VTS for diagnosis purposes. It is also structurally identical to the diagnoser constructed with traditional techniques depicted in Figure 2.8. However, it does not yet produce the same verdicts as the diagnoser. Note that the verdict assigned to the state $\langle \mathrm{d}, \varnothing \rangle$ has been updated with the join as per (4.4). As the join is the set intersection, the verdict remains unchanged.

Correctness Theorem. When the observables are restricted to O, then two observation sequences $\omega_1, \omega_2 \in \mathcal{L}(\mathfrak{V})$ of the input VTS become observationally indistinguishable iff they share the same O-projection, i.e., iff $\omega_1{\restriction}_{\mathrm{O}} = \omega_2{\restriction}_{\mathrm{O}}$. The VTS $\mathfrak{V}{\restriction}_{\mathrm{O}}$ produces most specific verdicts subsuming the verdicts produced for observation sequences with the same O-projection by the input VTS $\mathfrak{V}$, i.e., observation sequences that are indistinguishable when only observing observables in O. In more technical terms, for any observation sequence $\omega' \in \mathcal{L}(\mathfrak{V}{\restriction}_{\mathrm{O}})$ the verdict $\nu_{\mathfrak{V}{\restriction}_{\mathrm{O}}}(\omega')$ produced by

$\mathfrak{V}{\restriction}_{\mathrm{O}}$ is the most specific verdict subsuming all verdicts produced by $\mathfrak{V}$ for observation sequences whose O-projection is ω'.

Theorem 4.3.1 *Given a VTS $\mathfrak{V} = \langle \mathcal{Q}, J, \mathrm{Obs}, \rightarrowtriangle, \mathcal{V}, \sqsubseteq, \nu \rangle$ and a set $\mathrm{O} \subseteq \mathrm{Obs}$, we have (i) $\mathcal{L}(\mathfrak{V}{\restriction}_{\mathrm{O}}) = \mathcal{L}(\mathfrak{V}){\restriction}_{\mathrm{O}}$ and (ii) for all $\omega' \in \mathcal{L}(\mathfrak{V}{\restriction}_{\mathrm{O}})$:*

$$\nu_{\mathfrak{V}{\restriction}_{\mathrm{O}}}(\omega') = \bigsqcup \left\{ \nu(\omega) \;\middle|\; \omega \in \mathcal{L}(\mathfrak{V}) \text{ s.t. } \omega' = \omega{\restriction}_{\mathrm{O}} \right\}$$

Proof Sketch. Theorem 4.3.1 is proven by induction on the length of ω', similar to the proof of Theorem 4.1.1 (see Appendix A.1.1) and exploiting the fact that:

$$\bigcup \{ X_{|\mathcal{Q}|}(q) \mid q \in \mathsf{After}_{\mathfrak{V}{\restriction}_{\mathrm{O}}}(\omega') \} \;=\; \bigcup \{ \mathsf{After}_{\mathfrak{V}}(\omega) \mid \omega \in \mathcal{L}(\mathfrak{V}) \wedge \omega' = \omega{\restriction}_{\mathrm{O}} \}$$

□

Theorem 4.3.1 establishes a general relation between the input VTS $\mathfrak{V}$ and its projection $\mathfrak{V}{\restriction}_{\mathrm{O}}$. We also obtain the following corollary if the input VTS is sound, complete, and tight, e.g., when it has been constructed with annotation tracking.

Corollary 4.3.1 *If $\mathfrak{V}$ is sound, complete, and tight with respect to a system model $\mathfrak{S}$, a verdict oracle V, and an observation model Ω, then $\mathfrak{V}{\restriction}_{\mathrm{O}}$ is also sound, complete, and tight with respect to the same system model, the same verdict oracle, and the transformed observation model $\Omega{\restriction}_{\mathrm{O}}$ as per Definition 3.3.4.*

As the input VTS is sound and complete, the verdict $\nu(\omega)$ it produces for a given observation sequence ω is the most specific verdict $\mathsf{V}_{\Omega}(\omega)$, i.e., $\nu(\omega) = \mathsf{V}_{\Omega}(\omega)$ (recall Lemma 3.4.1). As it is tight, there is no spurious behavior that cannot actually be realized by the given system model. Hence, the fixpoint as per (4.3) only takes into account unobservable behaviors that may actually be realized by the system. As a result, Theorem 4.3.1 carries through to the most specific verdicts $\mathsf{V}_{\Omega{\restriction}_{\mathrm{O}}}(\omega')$ according to the transformed observation model.

Complexity. For constructing $\mathfrak{V}{\restriction}_{\mathrm{O}}$, the fixpoint computation as per (4.3) has to be carried out for each of the $|\mathcal{Q}|$ states before at most $|\mathcal{Q}|$ verdicts are joined as per (4.4). Hence, the worst-case time complexity lies in $\mathcal{O}(\mathsf{LOpCost}(|\mathcal{Q}|) \cdot |\mathcal{Q}| + |\mathcal{Q}|^2)$.

Relation to Belief State Constructions. In planning, it is common to ask for the set of possible states a system may be in under the assumption of partial observability, which we call limited observability here [Rin04; RN10]. Using appropriate verdict annotations, the techniques developed here can also be used to obtain such sets of possible system states (cf. discussion in Section 4.1.1). Observability projection as per Definition 4.3.1 is closely related to standard *belief state constructions* for tracking

possible states of a system after producing some observations [Rin04; RN10]. We defer the usual exponential blowup to the finalization stage of the pipeline (cf. Figure 4.1) by introducing non-determinism in the construction. The primary innovation over standard constructions lies in a general exploitation of the join-semilattice properties of verdict domains, of which state belief sets are a specialized instance. At its very core, observability projection tracks the possible states a VTS may be in. So, we can indeed reframe observability projection as a belief state construction about VTS states where a verdict for a state belief set is obtained by joining the verdicts assigned to the individual states. The constructions we present in the following for delays, losses, and out-of-order arrivals can be framed in a similar way.

4.3.2 Delays

When observations are made over a shared network, delays are typically unavoidable. If observations are delayed, then verdictors run the risk of producing outdated verdicts that are neither sound nor complete.

Example 4.5 Consider the VTS depicted in Figure 4.7. This VTS diagnoses faults based on the traditional diagnosis verdict domain (see Definition 3.1.2). So, after observing α, it produces $\{\{f\}\}$, indicating that a fault of fault class f certainly occurred. Before observing α, it produces $\{\emptyset\}$, indicating that no fault occurred. In case observations are delayed, it may be that α did occur, however, it has not been observed yet. As it has not been observed yet, the VTS still produces $\{\emptyset\}$, which is unsound. It indicates that the system functions nominally, when, in fact, a fault of fault class f may have occurred. The transformation we present in the following, allows transforming a VTS in order to account for such delays.

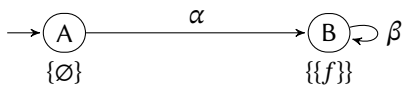


Figure 4.7: A VTS for fault diagnosis based on the traditional diagnosis verdict domain.

Unbounded Delays. Let us consider unbounded delays first. If observations can be delayed by an unbounded time, then instead of observing an observation sequence ω, a verdictor may observe any prefix of it. Any observations not on the specific prefix will then need to be considered delayed and only arrive eventually. Formally, this corresponds to the following observation model transformer Delay_∞:

$$\mathrm{Delay}_\infty(\Omega)(\rho) \coloneqq \bigcup \{\, \mathrm{Pref}(\omega) \mid \omega \in \Omega(\rho) \,\}$$

For instance, in case of Example 4.5, the actual observation sequence may be $\alpha\,\beta\,\beta$, however, with unbounded delays it is possible that the verdictor observed nothing yet, just α, the full sequence, or anything in between.

Now, to account for such delays with a VTS transformation, we use a similar fixpoint construction to limited observability and lookahead refinement. If observations can be arbitrarily delayed, then we must look into the indefinite future and consider all verdicts that may eventually be produced possible. To this end, let $Y_i(q)$ be the set of states reachable from $q \in Q$ by taking at most $i \in \mathbb{N}$ transitions:

$$Y_0(q) := \{q\} \qquad Y_{i+1}(q) := Y_i(q) \cup \mathrm{Post}\big(Y_i(q), \mathrm{Obs}\big) \tag{4.5}$$

Again, it is clear that Y reaches a fixpoint after $|Q|$ iterations. The only difference to (4.3) is that any transitions may be taken and not just unobservable ones. Intuitively, if observations are delayed by an unbounded time, then the verdicts of any of the states in $Y_{|Q|}(q)$ may be the correct verdict for the current state of the system. This insight gives rise to the following VTS transformation for unbounded delays.

Definition 4.3.2 *Given a VTS $\mathfrak{V} = \langle Q, J, \mathrm{Obs}, \rightarrow, \mathcal{V}, \sqsubseteq, \nu\rangle$, we define the* unbounded delay transformation $\mathrm{Delay}_\infty(\mathfrak{V})$ *of $\mathfrak{V}$ by*

$$\mathrm{Delay}_\infty(\mathfrak{V}) := \langle Q, J, \mathrm{Obs}, \rightarrow, \mathcal{V}, \sqsubseteq, \nu'\rangle \tag{4.6}$$

with $\nu'(q) := \bigsqcup_{q' \in Y_{|Q|}(q)} \nu(q')$ and $Y_{|Q|}$ as per (4.5).

We omit any further formal exposition of the unbounded case, as it is simpler but analogous to the bounded case which we consider next.

Bounded Delays. In case we can justify a bound on the possible delay, we can afford a construction that leads to more specific verdicts than for the unbounded case. Such bounds can often be justified by guarantees provided by networks, e.g., those typically used to connect manufacturing equipment [Fel05; TV99; Di +12; THW94]. Let us again first define an observation model transformer introducing a delay of up to $B \in \mathbb{N}$ observations:

$$\mathrm{Delay}_B(\Omega)(\rho) := \{\, \omega[1\,..\,|\omega| - \mathrm{D}] \mid \omega \in \Omega(\rho) \wedge 0 \leq \mathrm{D} \leq B \,\} \tag{4.7}$$

For $B = 0$, there is no delay and the transformed model is identical to the original. For $B > 0$, the observation model transformer introduces nondeterminism as a run may give rise to different observation sequences depending on the actual delay D. Note that the delay is considered to be discrete, as we are in the discrete-time setting. Continuous-time delays will be a topic of the next chapter.

Intuitively, if observations are delayed by up to B, then the verdict of any of the states in $Y_B(q)$ as pre (4.5) may be the correct verdict for the current state of the system.

This is analogous to the unbounded case, however, instead of using the fixpoint of Y, we just look B observations ahead. This is also exactly how many observations we need to look ahead, as observations may be delayed by at most B. The transformation of the input VTS then also proceeds analogously to Definition 4.3.2:

Definition 4.3.3 *Given a VTS* $\mathfrak{V} = \langle Q, J, \mathrm{Obs}, \rightarrow, \mathcal{V}, \sqsubseteq, \nu \rangle$ *and a delay bound* B*, we define the* B-delay transformation $\mathrm{Delay}_B(\mathfrak{V})$ *of* $\mathfrak{V}$ *by*

$$\mathrm{Delay}_B(\mathfrak{V}) := \langle Q, J, \mathrm{Obs}, \rightarrow, \mathcal{V}, \sqsubseteq, \nu' \rangle \tag{4.8}$$

with $\nu'(q) := \bigsqcup_{q' \in Y_B(q)} \nu(q')$ *and* Y_B *as per* (4.5).

Example 4.6 Figure 4.8 shows the VTS obtained by the delay transformation from the VTS depicted in Figure 4.7 for a delay bound of $B = 1$. Recall that the VTS is supposed to detect the occurrence of faults using the traditional fault diagnosis domain. Considering that observations may be delayed by up to one step, the transformed VTS now produces $\{\varnothing, \{f\}\}$ instead of $\{\varnothing\}$ in the state A, as the action α may have already occurred but not been observed yet. This verdict now indicates that it is possible that a fault of class f occurred and that it is also possible that no fault occurred, which is indeed correct. In contrast, the original VTS produced incorrect verdicts when faced with delayed observations (cf. Example 4.5). When the observation of α finally arrives, we still transition to the right state and indicate that now a fault of class f certainly occurred, which is also correct.

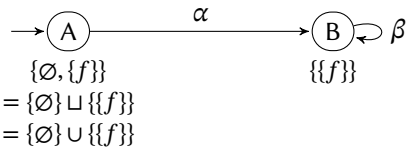


Figure 4.8: VTS obtained from the VTS shown in Figure 4.7 by the transformation for bounded delays with a bound of $B = 1$.

Correctness Theorem. The verdict produced for some observation sequence ω' by the transformed VTS $\mathrm{Delay}_B(\mathfrak{V})$ is the most specific verdict subsuming the verdicts produced by $\mathfrak{V}$ for those observation sequences out of which ω' may arise by a delay of up to B steps. Formally, we obtain the following correspondence theorem:

Theorem 4.3.2 *For a VTS $\mathfrak{V} = \langle \mathcal{Q}, J, \mathrm{Obs}, \rightarrowtriangle, \mathcal{V}, \sqsubseteq, \nu \rangle$ and delay bound B:*

(i) $\mathcal{L}\big(\mathrm{Delay}_B(\mathfrak{V})\big) = \{\, \omega[1\,..\,|\omega| - \mathsf{D}] \mid \omega \in \mathcal{L}(\mathfrak{V}) \wedge 0 \leq \mathsf{D} \leq B \,\}$

(ii) For all observation sequences $\omega' \in \mathcal{L}\big(\mathrm{Delay}_B(\mathfrak{V})\big)$, we have:

$$\nu_{\mathrm{Delay}_B(\mathfrak{V})}(\omega') = \bigsqcup \Big\{ \nu(\omega) \;\Big|\; \text{for } \omega \in \mathcal{L}(\mathfrak{V}) \text{ s.t. } \exists \wedge 0 \leq \mathsf{D} \leq B : \omega' = \omega[1\,..\,|\omega| - \mathsf{D}] \Big\}$$

Proof Sketch. Theorem 4.3.2 is proven by induction on the length of ω', similar to the proof of Theorem 4.1.1 (see Appendix A.1.1) and exploiting the fact that:

$$\begin{aligned} & \bigcup \{\, Y_B(q) \mid q \in \mathsf{After}_{\mathfrak{V}}(\omega') \,\} \\ = \quad & \bigcup \{\, \mathsf{After}_{\mathfrak{V}}(\omega) \mid \omega \in \mathcal{L}(\mathfrak{V}) \wedge \exists 0 \leq \mathsf{D} \leq B : \omega' = \omega[1\,..\,|\omega| - \mathsf{D}] \,\} \end{aligned} \tag{4.9}$$

□

From Theorem 4.3.2, we obtain the following corollary by a similar reasoning as for Corollary 4.3.1 based on the fact that $\nu(\omega) = \mathsf{V}(\omega)$.

Corollary 4.3.2 *If $\mathfrak{V}$ is sound, complete, and tight with respect to a system model $\mathfrak{S}$, a verdict oracle V, and an observation model Ω, then $\mathrm{Delay}_B(\mathfrak{V})$ is also sound, complete, and tight with respect to the same system model, the same verdict oracle, and the transformed observation model $\mathrm{Delay}_B(\Omega)$ for any B.*

For the unbounded case a theorem analogously to Theorem 4.3.2 applies where D is considered unbounded. Since Y_i as per (4.5) reaches a fixpoint after $|\mathcal{Q}|$ iterations, the transformation as per Definition 4.3.3 yields the same VTS for all bounds $B \geq |\mathcal{Q}|$ and collapses to Definition 4.3.2, i.e., when the bound exceeds $|\mathcal{Q}|$ it does no longer matter and we obtain a VTS for the unbounded case.

4.3.3 Losses

When observations are made over a shared network, they may not only be delayed but they can also get lost. For instance, there may be transmission errors or messages may be overwritten by higher priority messages [THW94]. If observations are lost, then verdictors run the risk of producing incorrect verdicts or getting stuck.

Example 4.7 Consider the VTS depicted in Figure 4.7 again. If, instead of being merely delayed, the observation of α gets lost while the VTS is in the state A, then analogously

to Example 4.5, the VTS does produce an incorrect verdict. It will still indicate $\{\varnothing\}$, despite the occurrence of α. Furthermore, when a subsequent observation of β is made, this observation cannot be fed into the VTS, i.e., the VTS is no longer applicable with respect to an observation model with losses.

Unbounded Losses. Like for delays, let us consider the unbounded case first and define an observation model transformer for it. If an unbounded number of observations may be lost, then any observation might be removed from an observation sequence that is generated for a given run:

$$\mathrm{Loss}_\infty(\Omega)(\rho) \coloneqq \{\, \mathrm{Word}(\omega') \mid \omega \in \Omega(\rho) \wedge \omega' \subseteq \omega \,\} \tag{4.10}$$

Recall that Word takes a semiword that may contain gaps and transforms it into a proper word by removing any gaps (recall Section 2.1).

The difference between losses and delays is that when observations are lost, then they will never arrive. To accommodate losses, we follow a similar strategy as for delays. However, this time, it is not sufficient to simply look at possible future verdicts. We also must adapt the transition relation as observations may never arrive and we may thus need to skip them. These considerations give rise to the following VTS transformation to account for unbounded losses:

Definition 4.3.4 *Given a VTS* $\mathfrak{V} = \langle Q, J, \mathrm{Obs}, \rightarrowtriangle, \mathcal{V}, \sqsubseteq, \nu \rangle$*, we define the* unbounded loss transformation $\mathrm{Loss}_\infty(\mathfrak{V})$ *of* $\mathfrak{V}$ *by*

$$\mathrm{Loss}_\infty(\mathfrak{V}) \coloneqq \langle Q, J, \mathrm{Obs}, \rightarrowtriangle', \mathcal{V}, \sqsubseteq, \nu' \rangle \tag{4.11}$$

with $\langle q, \mathsf{o}, q'' \rangle \in \rightarrowtriangle'$ *iff* $\langle q', \mathsf{o}, q'' \rangle \in \rightarrowtriangle$ *for some* $q' \in Y_{|Q|}(q)$*,*

$$\nu'(q) \coloneqq \bigsqcup\nolimits_{q' \in Y_{|Q|}(q)} \nu(q') \,,$$

and $Y_{|Q|}$ *as per* (4.5)*.*

The difference to Definition 4.3.2 lies in the fact that the transition relation is also changed allowing observations to be skipped. As for unbounded delays, we omit a further formal exposition as it is simpler but analogous to the bounded case which we consider next.

Bounded Losses. As for delays, we may assume that there is an upper bound $B \in \mathbb{N}$ on the number of observations that may get lost consecutively. Such a bound can often be derived from the guarantees of the network.

For instance, it is well-known that in communication networks packets get lost in bursts. This insight lead to the established Gilbert-Elliott channel model [Gil60;

Ell63]. Following the Gilbert-Elliott model, one can compute a probability that not more than B packets are lost consecutively. A bound B on consecutive losses can then be chosen such that this probability is sufficiently low. For further details on the Gilbert-Elliott channel model and the computation of the revelant probabilities, we refer to the existing literature [Gil60; Ell63].

To formally model such *bounded losses* in terms of an observation model transformer, we follow a model established in the literature on *weakly-hard real-time systems* for consecutive deadline misses [BBL01; PM22]:

Definition 4.3.5 *Let $\varpi \in \{\mathsf{L}, \mathsf{A}\}^\star$ be a finite sequence over the set $\{\mathsf{L}, \mathsf{A}\}$ where L indicates that an observation gets lost and A indicates that it arrives. The word ϖ satisfies the constraint of at most $B \in \mathbb{N}$ consecutive losses iff:*

$$\forall 1 \leq i \leq j \leq |\varpi| : \varpi(i) = \varpi(j) = \mathsf{L} \wedge j - i > B \Longrightarrow \exists i \leq k \leq j : \varpi(k) = \mathsf{A}$$

For $B \in \mathbb{N}$, we denote the set of such words by LA_B.

That is, between any two losses that are more than B apart, at least one observation arrives. For a given observation sequence ω and a word $\varpi \in \mathsf{LA}_B$ such that $|\omega| = |\varpi|$, we define a *loss projection* that removes all lost observations:

$$\omega \downarrow_\varpi := \mathsf{Word}(\{\, \langle i, \mathsf{o}\rangle \in \omega \mid \varpi(i) = \mathsf{A} \,\})$$

Using LA_B and the loss projection, we then define an observation model transformer:

$$\mathsf{Loss}_B(\Omega)(\rho) := \{\, \omega' \mid \exists \omega \in \Omega(\rho), \varpi \in \mathsf{LA}_B \text{ s.t. } |\omega| = |\varpi| : \omega\downarrow_\varpi = \omega' \,\} \tag{4.12}$$

The resulting observation model may induce up to B consecutive losses in the original observation sequences. As in the unbounded case, a VTS robust to at most B consecutive losses is synthesized in a similar way as for bounded delays with the addition that observations may be skipped.

Definition 4.3.6 *Given a VTS $\mathfrak{V} = \langle Q, J, \mathsf{Obs}, \rightarrowtriangle, \mathcal{V}, \sqsubseteq, \nu\rangle$ and a loss bound B, we define the B-*loss transformation $\mathsf{Loss}_B(\mathfrak{V})$ *of $\mathfrak{V}$ by*

$$\mathsf{Loss}_B(\mathfrak{V}) := \langle Q, J, \mathsf{Obs}, \rightarrowtriangle', \mathcal{V}, \sqsubseteq, \nu'\rangle \tag{4.13}$$

with $\langle q, \mathsf{o}, q''\rangle \in \rightarrowtriangle'$ iff $\langle q', \mathsf{o}, q''\rangle \in \rightarrowtriangle$ for some $q' \in Y_B(q)$,

$$\nu'(q) := \bigsqcup\nolimits_{q' \in Y_B(q)} \nu(q')\,,$$

and Y_B as per (4.5).

Remark. The observation model for up to B bounded losses implicitly also accounts for up to B bounded delays. Formally, we have:

$$\forall \rho \in \mathsf{Runs}(\mathfrak{S}) : \quad \mathsf{Delay}_B(\Omega)(\rho) \subseteq \mathsf{Loss}_B(\Omega)(\rho) \tag{4.14}$$

Assume given an observation sequence $\omega \in \Omega(\rho)$ of the original observation model, i.e., without any losses or delays. Now, a delay of up to B may cut up to B observations off of ω as per (4.7). The resulting observation sequence $\omega[1\,..\,|\omega| - \mathsf{D}]$ for $0 \leq \mathsf{D} \leq B$ is identical to the sequence obtained when the last D observations are lost as per (4.12). Hence, we have (4.14). As a result, if there are bounded losses and delays, it suffices to apply the transformation for bounded losses, which then transparently also handles bounded delays. An analogous argument can be made for the respective unbounded cases. These considerations are also reflected in the fact that Definition 4.3.2 and Definition 4.3.4, as well as Definition 4.3.3 and Definition 4.3.6 share the definition of the transformed verdict function ν', respectively.

Example 4.8 Figure 4.9 shows the VTS obtained by the loss transformation from the VTS depicted in Figure 4.7 for a loss bound of $B = 1$. The transformation fixes both issues described in Example 4.7. First of all, similar to Example 4.6, the left state is now $\{\varnothing, \{f\}\}$, indicating that it is possible that a fault of class f occurred and that it is possible that the system functions nominally. This is correct as the observation α may have been lost. In addition, in cases where α is lost, and β is observed next, the added transition to the right state can be taken thereby preventing the verdictor from getting stuck. Taking this transition, then also correctly leads to a state indicating that a fault of class f certainly occurred.

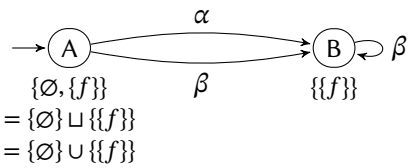


Figure 4.9: VTS obtained from the VTS shown in Figure 4.7 by the transformation for losses with a bound of $B = 1$.

Correctness Theorem. The verdict produced for some observation sequence ω' by the transformed VTS $\mathsf{Loss}_B(\mathfrak{V})$ is the most specific verdict subsuming the verdicts produced by $\mathfrak{V}$ for those observation sequences out of which ω' may arise by up to B consecutive losses. Formally, we establish the following general theorem:

Theorem 4.3.3 *For a VTS* $\mathfrak{V} = \langle \mathcal{Q}, J, \mathrm{Obs}, \rightarrowtriangle, \mathcal{V}, \sqsubseteq, \nu \rangle$ *and loss bound B:*

(i) $\mathcal{L}\big(\mathrm{Loss}_B(\mathfrak{V})\big) = \{\, \omega\downarrow_{\varpi} \mid \omega \in \mathcal{L}(\mathfrak{V}), \varpi \in \mathsf{LA}_B, |\varpi| = |\omega| \,\}$

(ii) For all observation sequences $\omega' \in \mathcal{L}\big(\mathrm{Loss}_B(\mathfrak{V})\big)$*, we have:*

$$\nu_{\mathrm{Loss}_B(\mathfrak{V})}(\omega') = \bigsqcup \Big\{ \nu(\omega) \;\Big|\; \text{for } \omega \in \mathcal{L}(\mathfrak{V}) \text{ s.t. } \exists \varpi \in \mathsf{LA}_B, |\varpi| = |\omega| : \omega' = \omega\downarrow_{\varpi} \Big\}$$

Proof Sketch. Theorem 4.3.3 is proven by induction on the length of ω', similar to the proof of Theorem 4.1.1 (see Appendix A.1.1) and exploiting the fact that:

$$\begin{aligned} & \bigcup \{\, Y_B(q) \mid q \in \mathsf{After}_{\mathrm{Loss}_B(\mathfrak{V})}(\omega') \,\} \\ = \; & \bigcup \{\, \mathsf{After}_{\mathfrak{V}}(\omega) \mid \omega \in \mathcal{L}(\mathfrak{V}) \wedge \exists \varpi \in \mathsf{LA}_B, |\varpi| = |\omega| : \omega' = \omega\downarrow_{\varpi} \,\} \end{aligned} \tag{4.15}$$

□

From Theorem 4.3.3, we obtain the following corollary by a similar reasoning as for Corollary 4.3.1 based on the fact that $\nu(\omega) = \mathsf{V}(\omega)$.

Corollary 4.3.3 *If* $\mathfrak{V}$ *is sound, complete, and tight with respect to a system model* $\mathfrak{S}$*, a verdict oracle* V*, and an observation model* Ω*, then* $\mathrm{Loss}_B(\mathfrak{V})$ *is also sound, complete, and tight with respect to the same system model, the same verdict oracle, and the transformed observation model* $\mathrm{Loss}_B(\Omega)$ *for any B.*

For the unbounded case a theorem analogously to Theorem 4.3.3 applies where losses are now unbounded. Since Y_i as per (4.5) reaches a fixpoint after $|\mathcal{Q}|$ iterations, the transformation as per Definition 4.3.6 yields the same VTS for all bounds $B \geq |\mathcal{Q}|$ and collapses to Definition 4.3.4, i.e., when the bound exceeds $|\mathcal{Q}|$ it does no longer matter and we obtain a VTS for the unbounded case.

4.3.4 Bounded Out-of-Order Arrivals

For delays, as previously introduced, we assumed that observations are delayed en block, i.e., without changing their order. This assumption can be justified, e.g., if all observations originate from a single sender on a shared network. For instance, the single sender may assign sequence numbers to observations such that they can be ordered, even if the underlying packets on the network carrying the observations may arrive out-of-order. For distributed systems where observations may originate from multiple senders, however, this assumption is no longer justified. Here, observations may be delayed individually and there may be no way to totally order them upon arrival. In such a case, verdictors have to account for out-of-order arrivals of

observations. In the following, we restrict our considerations to *bounded out-of-order arrivals*.[9]

Technically, out-of-order arrivals lead to permuted observation sequences. As for bounded delays and losses, we assume that there is an upper bound B constraining possible permutations. Formally, we define *bounded permutations* as follows:

Definition 4.3.7 *For a bound $B \in \mathbb{N}$ and $N \in \mathbb{N}$, a* bounded permutation *$\xi : \{1 \,..\, N\} \to \{1 \,..\, N\}$ is a bijection such that $|n - \xi(n)| \leq B$ for all $1 \leq n \leq N$. Let $\Xi_{\langle B,N \rangle}$ be the set of bounded permutations for bound B and N.*

For a given observation sequence ω and bounded permutation $\xi \in \Xi_{\langle B,|\omega|\rangle}$, we denote the corresponding permutation of ω by $\omega \downharpoonright \xi$. Formally, we define:

$$(\omega \downharpoonright \xi)(i) \coloneqq \omega(\xi(i))$$

Based on these definitions, we define the following observation model transformer Reord_B for bounded out-of-order arrivals of observations:

$$\begin{aligned} & \mathrm{Reord}_B(\Omega)(\rho) \\ \coloneqq \; & \left\{ \omega' \;\middle|\; \exists \omega \in \Omega(\rho), \xi \in \Xi_{\langle B,|\omega|\rangle}, 0 \leq \mathrm{D} \leq B+1 : (\omega \downharpoonright \xi)[0..|\omega| - \mathrm{D}] = \omega' \right\} \end{aligned} \tag{4.16}$$

Notably, the resulting observation models also include a delay of up to $B + 1$ observations. As discussed above, out-of-order arrivals are fundamentally caused by observations that are delayed individually. This delay is inherent to out-of-order arrivals. Without the delay, an observation needs to be directly observed preventing any permutations. From a technical perspective, dropping the delay from (4.16) would violate the condition for observation models as per Definition 3.3.1. Take Example 4.5 as an example again and assume that α occurs and is required to be observed immediately, i.e., without a delay. Then, α would be the only possible observation sequence. Now, if β occurs next, it cannot be made to arrive before α as this would not continue the previous observation sequence α. Recall that it is a fundamental requirement of observation models that future observation sequences continue previous ones. Hence, the delay in (4.16) is justified from a practical perspective and this practical perspective is also reflected in the technical definition of observation models.[10]

[9] While bounded and unbounded delays and losses can be handled similarly, unbounded out-of-order arrivals are more complex to handle than bounded out-of-order arrivals. In particular, the construction we present in the following only works for the bounded case.

[10] A delay of B instead of $B + 1$ would be sufficient. However, this would complicate the VTS transformation we present in the following. This transformation is based on speculating about the next $B + 1$ observations in order to account for any permutations. The transformation is straightforwardly adapted to account for a delay of just B by tracking a history of $B + 1$ states and using the last state of the history (if it exists) instead of the speculated state to determine the verdict.

VTS Transformation. The VTS transformation for out-of-order arrivals is based on the idea to first speculate about future observations (which may be delayed) and then handle them in which ever order they are made, when they are made. The states of the transformed VTS will be pairs $\langle q, \aleph \rangle$ where q is a state of the input VTS and $\aleph$ is a sequence of *speculative observations*. Formally, we have

$$\langle q, \aleph \rangle \in Q \times (\mathrm{Obs} \cup \{\#\})^{B+1}$$

where $(\mathrm{Obs} \cup \{\#\})^{B+1}$ is the set of finite sequences of length $B+1$ over $\mathrm{Obs} \cup \{\#\}$ and $\# \notin \mathrm{Obs}$ is a *marker* used to indicate the absence of a specific observable. The sequence $\aleph$ contains speculative observations which have not been made yet. For instance, $\langle \mathsf{B}, \alpha\, \beta \rangle$ encodes that the state B has been reached by speculating about future observations of α and β. When β is observed, then it is set to #, leading to the state $\langle \mathsf{B}, \alpha\, \# \rangle$. Likewise, when α is observed, then it is set to #, leading to the state $\langle \mathsf{B}, \#\, \beta \rangle$. As out-of-order arrivals are bounded, it suffices to speculate about at most $B+1$ observations. When a speculative observation sequence $\aleph$ starts with #, this means that the first speculative observation has been made and we can thus continue speculating. To this end, we define a function Shift as follows:

$$\mathsf{Shift}(\langle q, \aleph \rangle) := \{\, \langle q', \aleph[2..] \diamond \mathsf{o} \rangle \mid \langle q, \mathsf{o}, q' \rangle \in \rightarrow \,\}$$

This function discards the first element of $\aleph$ and extends $\aleph$ with an observation that may be made in state q and lead to state q'. In case of Example 4.5, we may speculate that first α and then β occurs, resulting in the state $\langle \mathsf{B}, \alpha\, \beta \rangle$. Now, if α is observed, we end up in $\langle \mathsf{B}, \#\, \beta \rangle$ and can speculate further resulting in $\langle \mathsf{B}, \beta\, \beta \rangle$.

To account for observations that we have speculated about, we define

$$\mathsf{Apply}(\langle q, \aleph \rangle, \mathsf{o}) := \{\, \langle q, \aleph' \rangle \mid \exists 1 \leq i \leq |\aleph| \,:\, \aleph(i) = \mathsf{o} \wedge \aleph' = \mathsf{Mark}(\aleph, i) \,\}$$

where Mark replaces a speculative observation with #:

$$\mathsf{Mark}(\aleph, i)(j) := \begin{cases} \aleph(j) & \text{if } j \neq i \\ \# & \text{otherwise} \end{cases}$$

The set $\mathsf{Apply}(\langle q, \aleph \rangle, \mathsf{o})$ contains possible states that may result from observing o. In general, there may be multiple speculative observations of o and we cannot know to which thereof a particular observation of o corresponds. Hence, we must consider the entire set $\mathsf{Apply}(\langle q, \aleph \rangle, \mathsf{o})$. This set contains all states that may result from replacing a speculative observation of o at some position i with #.

As a final building block for the VTS transformation, we need a way to speculate about as much observations as required for the bound B. We achieve this by yet another fixpoint construction. To this end, given a set $X \subseteq Q \times (\mathrm{Obs} \cup \{\#\})^{B+1}$ of states as discussed above, let S_i be inductively defined as follows:

$$\mathsf{S}_0(X) := X$$

$$\mathsf{S}_{i+1}(X) \coloneqq \mathsf{S}_i(X) \cup \bigcup \{\, \mathsf{Shift}(\langle q, \aleph \rangle) \mid \langle q, \aleph \rangle \in \mathsf{S}_i(X) \;\text{ s.t. }\; \aleph(1) = \# \,\}$$

It is easy to see that S_i reaches a fixpoint after at most $B+1$ iterations as Shift only adds concrete observables and we only apply it to those states $\langle q, \aleph \rangle$ where $\aleph(1)$ is not yet a concrete observable. Based on this fixpoint, we define the following VTS transformation for bounded out-of-order arrivals of observations:

Definition 4.3.8 *Given a VTS $\mathfrak{V} = \langle Q, J, \mathsf{Obs}, \rightarrowtriangle, \mathcal{V}, \sqsubseteq, \nu \rangle$ and a recording bound B, we define the B-*reordering transformation $\mathsf{Reord}_B(\mathfrak{V})$ *of $\mathfrak{V}$ by*

$$\mathsf{Reord}_B(\mathfrak{V}) \coloneqq \left\langle Q \times (\mathsf{Obs} \uplus \{\#\})^{B+1}, J', \mathsf{Obs}, \rightarrowtriangle', \mathcal{V}, \sqsubseteq, \nu' \right\rangle \tag{4.17}$$

where

$$J' \coloneqq \mathsf{S}_{B+1}\left(\left\{ \langle q, \aleph \rangle \;\middle|\; q \in J \wedge \aleph \in \{\#\}^{B+1} \right\}\right)$$

$$\nu'(\langle q, \cdot \rangle) \coloneqq \nu(q)$$

and with $\langle \langle q, \aleph \rangle, \mathsf{o}, \langle q', \aleph' \rangle \rangle \in \rightarrowtriangle'$ iff:

$$\langle q', \aleph' \rangle \in \mathsf{S}_{B+1}(\mathsf{Apply}(\langle q, \aleph \rangle, \mathsf{o}))$$

The initial states J' are defined by speculating about observations starting in the initial states of the input VTS. The transitions $\rightarrowtriangle'$ are defined by first accounting for an observation based on the speculations and then speculating further. Notably, this VTS transformation exploits non-determinism to handle all possible speculative observations, relying on Definition 3.2.2 to join their verdicts.

Example 4.9 Figure 4.10 shows the VTS obtained by the out-of-order transformation from the VTS depicted in Figure 4.7 for a bound of $B = 1$. Figure 4.11 shows the determinization of this VTS as per Definition 3.2.7. As for the transformations for delays and losses (see Figure 4.8 and Figure 4.9, respectively), the verdict of the initial state is $\{\varnothing, \{f\}\}$, indicating that the system may function nominally or that a fault of class f occurred. As in the case of losses, β is a valid observation in the initial state. In contrast to the transformation for losses, however, this observation must then be followed by α. Still, a fault of class f is diagnosed as soon as β is observed, as this indicates that we should already be in state B of the original VTS.

Correctness Theorem. The verdict produced for some observation sequence ω' by the transformed VTS $\mathsf{Reord}_B(\mathfrak{V})$ is the most specific verdict subsuming the verdicts produced by $\mathfrak{V}$ for those observation sequences out of which ω' may arise by out-of-order arrivals bounded by B as per (4.16). Formally, we establish the following general theorem about the transformation:

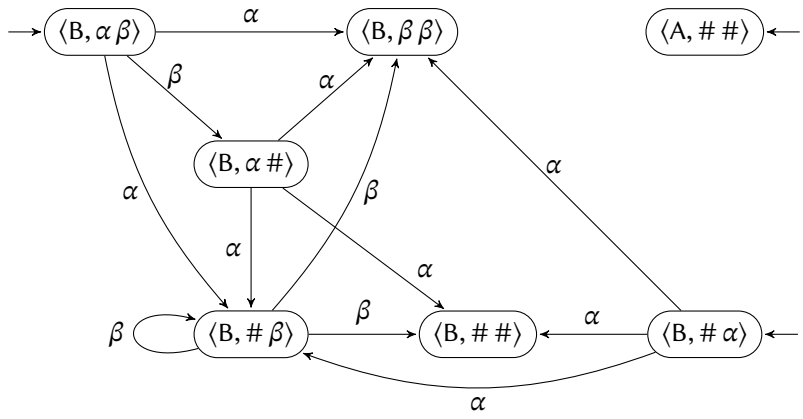


Figure 4.10: VTS obtained from the VTS shown in Figure 4.7 by the transformation for out-of-order-arrivals with a bound of $B = 1$. Verdicts have been omitted from the presentation. They are inherited from the original VTS.

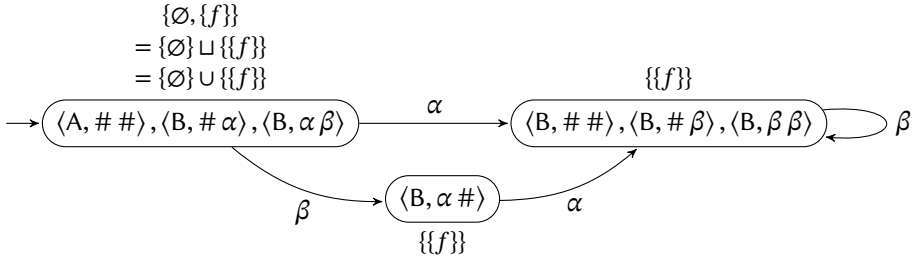


Figure 4.11: Determinization of the VTS shown in Figure 4.10.

Theorem 4.3.4 *For a VTS* $\mathfrak{V} = \langle Q, J, \mathrm{Obs}, \rightarrow, \mathcal{V}, \sqsubseteq, \nu \rangle$ *and reorder bound B:*

(i) $\mathcal{L}\big(\mathrm{Reord}_B(\mathfrak{V})\big) = \big\{ (\omega \downarrow \xi)[0..|\omega| - \mathrm{D}] \;\big|\; \omega \in \Omega(\rho), \xi \in \Xi_{\langle B, |\omega| \rangle}, 0 \leq \mathrm{D} \leq B + 1 \big\}$

(ii) For all observation sequences $\omega' \in \mathcal{L}\big(\mathrm{Reord}_B(\mathfrak{V})\big)$*, we have:*

$$\nu_{\mathrm{Reord}_B(\mathfrak{V})}(\omega') = \bigsqcup \big\{ \nu(\omega) \;\big|\; \omega \in \mathcal{L}(\mathfrak{V}) \wedge \exists \xi \in \Xi_{\langle B, |\omega| \rangle}, 0 \leq \mathrm{D} \leq B + 1 : (\omega \downarrow \xi)[0..|\omega| - \mathrm{D}] = \omega' \big\}$$

Proof Sketch. Theorem 4.3.3 is proven by induction on the length of ω', similar to the

proof of Theorem 4.1.1 (see Appendix A.1.1) and exploiting the fact that:

$$\begin{aligned} & \bigcup \{ q \mid \langle q, \cdot \rangle \in \mathsf{After}_{\mathsf{Reord}_B(\mathfrak{B})}(\omega') \} \\ = \; & \bigcup \left\{ \mathsf{After}_{\mathfrak{B}}(\omega) \;\middle|\; \begin{array}{l} \omega \in \mathcal{L}(\mathfrak{B}) \wedge \exists \xi \in \Xi_{\langle B, |\omega| \rangle}, 0 \leq \mathsf{D} \leq B+1 : \\ (\omega \lfloor \xi)[0..|\omega| - \mathsf{D}] = \omega' \end{array} \right\} \end{aligned}$$

□

From Theorem 4.3.3, we obtain the following corollary by a similar reasoning as for Corollary 4.3.1 based on the fact that $\nu(\omega) = \mathsf{V}(\omega)$.

Corollary 4.3.4 *If $\mathfrak{B}$ is sound, complete, and tight with respect to a system model $\mathfrak{S}$, a verdict oracle V, and an observation model Ω, then $\mathsf{Reord}_B(\mathfrak{B})$ is also sound, complete, and tight with respect to the same system model, the same verdict oracle, and the transformed observation model $\mathsf{Reord}_B(\Omega)$ for any B.*

4.3.5 Possibility Lifting

As a common theme of the introduced transformations, when faced with observational imperfections, certain verdicts become indistinguishable. We deal with those indistinguishable verdicts by subsuming them into a most specific verdict (cf. join in Definition 4.3.1, Definition 4.3.3, Definition 4.3.6, and Definition 4.3.8). For instance, in case of Example 4.4, we combined multiple sets of fault classes by joining them, which corresponds to the intersection in case of the verdict domain as per Definition 3.1.3. As a result, we obtained a diagnoser that is simplified and lacks in capabilities compared to the traditional construction shown in Figure 2.8. Instead of combining the verdicts by joining them, we can also retain them as individual *possibilities*. To this end, we introduce *possibility lifting*. Possibility lifting replaces the verdict domain $\langle \mathcal{V}, \sqsubseteq \rangle$ of a VTS with $\langle \wp(\mathcal{V}), \subseteq \rangle$, where the individual verdicts in $\mathsf{V} \in \wp(\mathcal{V})$ represent possible verdicts. As the join now is set union, the individual verdicts are collected in a set. Possibility lifting is applied before applying the transformations for observational imperfections. Formally, it is defined as follows:

Definition 4.3.9 *Let $\mathfrak{B} = \langle Q, J, \mathsf{Obs}, \rightarrowtail, \mathcal{V}, \sqsubseteq, \nu \rangle$ be a VTS. The possibility lifting $\lceil \mathfrak{B} \rceil$ of $\mathfrak{B}$ is defined as follows:*

$$\lceil \mathfrak{B} \rceil := \langle Q, J, \mathsf{Obs}, \rightarrowtail, \wp(\mathcal{V}), \subseteq, \nu' \rangle \quad \textit{with} \quad \nu'(q) := \{\nu(q)\}$$

Example 4.10 Figure 4.12 shows the result of possibility lifting applied to the VTS constructed by annotation tracking (see Figure 4.3) from the coffee machine model (see Figure 4.2). Recall that without possibility lifting, the VTS constructed from this VTS by observability projection did not yet produce the same verdicts as the

diagnoser constructed with traditional techniques (cf. Example 4.4). If we apply possibility lifting before the observability projection, we actually obtain a VTS that produces exactly the same verdicts as the diagnoser constructed with traditional techniques. To this end, compare the VTS depicted in Figure 4.13 with the traditional diagnoser depicted in Figure 2.8. In general, we obtain the traditional construction as a special case of our synthesis pipeline by annotation tracking, possibility lifting, and subsequent observability projection.

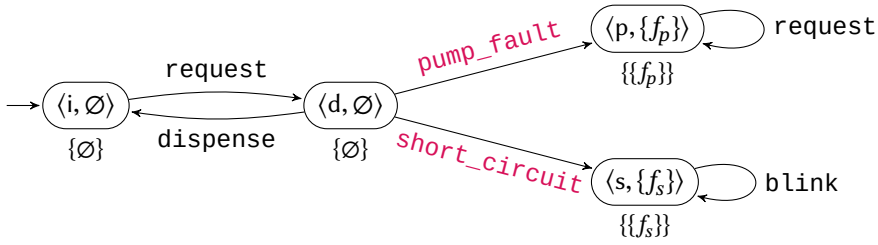


Figure 4.12: VTS obtained by observability lifting from the VTS constructed by annotation tracking (see Figure 4.3) from the coffee machine model (see Figure 4.2).

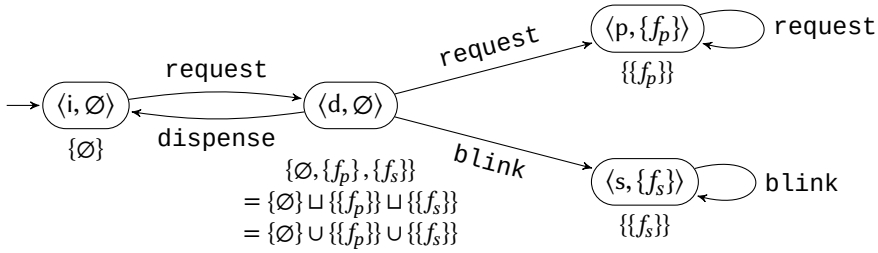


Figure 4.13: VTS obtained from the verdict-annotated model of the coffee machine (see Figure 4.2) by applying annotation tracking (see Figure 4.3) followed by possibility lifting (see Figure 4.12), and observability projection.

4.4 Finalization

Using the presented stages of the pipeline enables the synthesis of a VTS, starting from a verdict-annotated system model. The VTS serves as an explicit representation of a verdictor and can be transformed to produce predictions and to account for various observation imperfections. To efficiently implement a VTS, whether in software or hardware, it is desirable for it to be deterministic and minimal (cf. Section 3.2.2). If it is deterministic, then we only need to keep track of a single state at runtime and can

use an efficient lookup table to implement the transition relation. If it is minimal, then we require the least amount of space for storing its state and its implementation, e.g., in the form of a transition lookup table. Note that the presented VTS transformations, e.g., for limited observability and losses, potentially introduce nondeterminism, even if the input VTS was deterministic, as they modify the transition relation. Having a deterministic and minimal VTS representation is particularly crucial for environments with space limitations, like embedded devices or FPGAs [Bod+04; Zha+22]. While determinization generally increases the size, minimization may reduce it.

Our earlier determinization and minimization results for VTSs (see Section 3.2.2) can be directly exploited to obtain deterministic and minimal VTSs. In addition, we present an additional technique called *language-relaxing minimization*. VTSs can then be finalized towards an efficient implementation by determinization followed by either classical minimization or language-relaxing minimization.

4.4.1 Language-Relaxing Minimization

While traditional minimization preserves the language of a VTS, i.e., the observation sequences it accepts, language-relaxing minimization is more liberal and allows the VTS to accept additional observation sequences. At runtime, we may not care about detecting observation sequences that are not possible according to a VTS, e.g., as we may assume that the system may indeed only generate observation sequences that the VTS accepts. In those cases, only the verdicts produced by a VTS matter but not its language. Given a VTS $\mathfrak{V}$, language-relaxing minimization constructs a $\mathfrak{V}'$ with $\nu'(\sigma) = \nu(\sigma)$ for all $\sigma \in \mathcal{L}(\mathfrak{V})$ and where $\mathfrak{V}'$ may also accept additional observation sequences and produce arbitrary verdicts for them, i.e., $\mathcal{L}(\mathfrak{V}') \supseteq \mathcal{L}(\mathfrak{V})$. The VTS $\mathfrak{V}'$ can be even smaller than the minimization of $\mathfrak{V}$. Finding the smallest such $\mathfrak{V}'$ is an interesting challenge on its own, which we leave for future work. As a first step towards language-relaxing minimization, we adapt the minimization algorithm developed by Valmari and Lehtinen [VL08], which itself is an adaptation of Hopcroft's original minimization algorithm [Hop71].

Algorithm. At its core, Hopcroft's original minimization algorithm maintains a partition of the states of an automaton. This partition is then successively refined until no further refinement is necessary to obtain Myhill-Nerode equivalence classes of states, i.e., sets of states from which the same language is accepted [Hop71; Myh57]. To refine an equivalence class, the algorithm considers so called *splitters*. We will illustrate the idea on a high-level using VTS terminology. Given a deterministic and finite input VTS $\mathfrak{V} = \langle Q, J, \mathrm{Obs}, \rightarrow, \mathcal{V}, \sqsubseteq, \nu \rangle$ and a partition $\mathcal{U} = \{\mathcal{C}_i\}_{i=1}^{n}$ of its states Q, a splitter is a pair $\langle \mathcal{C}, \mathsf{o} \rangle$ where $\mathcal{C} \in \mathcal{U}$ is a class of the partition $\mathcal{U}$ and $\mathsf{o} \in \mathrm{Obs}$ is an observable. A splitter is used to refine the partition $\mathcal{U}$. Traditionally, a class

$\mathcal{C}' \in \mathcal{U}$ is split by a splitter into two sets of states:

$$Q_{\text{in}} \coloneqq \{\, q \in \mathcal{C}' \mid \exists q' \in \mathcal{C} : \langle q, \mathsf{o}, q' \rangle \in \rightarrow \,\}$$
$$Q_{\text{miss}} \coloneqq \mathcal{C}' \setminus Q_{\text{in}}$$

The set Q_{in} contains all states of $\mathcal{C}'$ from which an observation of o leads to a state in $\mathcal{C}$. Recall that the input VTS is deterministic, hence, if such a transition exists, then there is no other transition that would lead to some other state not in $\mathcal{C}$. We say that those states *hit inside* the splitter. Now, the remaining states of $\mathcal{C}'$ are collected in Q_{miss}, those do not hit inside the splitter, i.e., they *miss*. If both sets are non-empty, then we would need to refine the class $\mathcal{C}'$ into states that hit and states that miss the splitter. Clearly, those states should not belong to the same equivalence class of states, as an observation of o leads to states in different classes. Traditional automata are input-enabled, i.e., in each state there exists a transition for each symbol. In contrast, VTSs are not input-enabled. As a result, we may consider a third set of states:

$$Q_{\text{out}} \coloneqq \{\, q \in \mathcal{C}' \mid \exists q' \in \mathcal{Q} \setminus \mathcal{C} : \langle q, \mathsf{o}, q' \rangle \in \rightarrow \,\}$$

The states in Q_{out} are those states that hit *outside* of the splitter. From those states an observation of o will actually lead to a state that is not in $\mathcal{C}$. For Q_{miss} this is not generally the case, as for those states there may also not exist a transition for the observable o at all. If the set Q_{out} is empty, then all the states in Q_{miss} just lack a transition for o. In this case and if we do not care about the language, we may thus just add transitions that lead to $\mathcal{C}$. As a result, we do avoid the need to split $\mathcal{C}'$ by accepting additional observation sequences.

In a nutshell, for language-relaxing minimization, we modify the minimization algorithm by Valmari and Lehtinen such that classes are only split by a splitter if states hit inside and outside of the splitter. To this end, their algorithm is modified to check this condition before splitting. Note that the resulting algorithm, in contrast to the original, is non-deterministic: The outcome depends on the order in which splitters are considered, which is left unspecified. For a detailed exposition of their algorithm and our modifications to it, we refer to the original paper [VL08] and our implementation (AT1), which also concretizes all data structures.

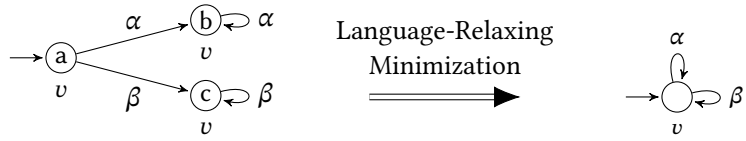


Figure 4.14: An example of language-relaxing minimization.

Example 4.11 Figure 4.14 shows an example of language-relaxing minimization. Assume that all the states of the VTS on the left produce the same verdict υ, i.e., they

initially belong to the same class. Traditionally, this class would need to be split into three, as the first state allows us to decide between sequences of just α and sequences of just β. Let's call this class of states $\mathcal{C}$. For the splitter $\langle \mathcal{C}, \alpha \rangle$, we obtain:

$$Q_{\text{in}} \coloneqq \{\text{a}, \text{b}\}$$
$$Q_{\text{miss}} \coloneqq \{\text{c}\}$$
$$Q_{\text{out}} \coloneqq \varnothing$$

Thus, following the traditional algorithm, we would have to split $\mathcal{C}$, however, as Q_{out} is empty, we can avoid splitting. For the splitter $\langle \mathcal{C}, \beta \rangle$, the argument applies analogously. Hence, we do not need to split $\mathcal{C}$ leading to the language-relaxed minimization on the right. Now, this VTS accepts additional observation sequences, namely any combinations of α and β, however, it will still produce the same verdicts for the original sequences of just α or of just β.

Impact on Applicability and Tightness. Language-relaxing minimization may enlarge but will never shrink the language accepted by a VTS, i.e., every observation sequence accepted by the original VTS will also be accepted after language-relaxing minimization. Therefore, a VTS applicable with respect to a given system and observation model will stay applicable. However, a VTS which has been tight with respect to a given system and observation model, may no longer be tight after language-relaxing minimization due to it accepting additional observation sequences which have not been accepted by the original VTS and are not in the observable language of the system model with respect to the given observation model.

Impact on Soundness and Completeness. Language-relaxing minimization has no impact on the soundness or completeness of a VTS with respect to a given system model and observation model. This is the case since soundness and completeness are defined over the observable language of the system model, which is completely independent of the VTS and its language, and the verdict generated for a particular observation sequence is not changed.

4.5 Discussion

We now have all the building blocks of the generic synthesis pipeline for VTS synthesis (see Figure 4.1). In the first step, a VTS is constructed based on a system model, either by annotation tracking or by some other means. Annotation tracking takes a system model annotated with verdicts, e.g., a featured transition system, and produces a VTS tracking these annotations. Optionally, lookahead refinement can be applied to enable most specific predictions. Then, to account for limited observability, delays, losses, and out-of-order arrivals, the presented transformations can be applied to

obtain a VTS robust against these imperfections. Note that the transformations can trivially be cascaded to obtain a VTS that accounts for multiple imperfections in a most specific manner. Soundness, completeness, and tightness is preserved along any such cascade independently of the verdict oracle. Lastly, the VTS can be determinized and minimized towards an efficient implementation. We have exemplified how the techniques can be used to synthesize traditional diagnosers (recall Section 4.3.5). We will explore further concrete use cases in Chapter 7 and Chapter 8 and evaluate the techniques empirically as part of Chapter 8 on configuration monitoring.

Composing Transformations. The presented VTS transformations are, in general, neither associative nor commutative, i.e., the order in which they are applied matters. As observational imperfections typically require concessions with respect to the specificity of verdicts, lookahead refinement should always be applied before applying any of the transformations for observational imperfections. That way, as much information as possible is exploited for predictions before it is watered down by observational imperfections. As the transformations for losses and out-of-order arrivals also account for delays, it typically makes no sense to cascade them with the transformation for delays. When composing the transformations for losses and out-of-order arrivals, it typically makes sense to first account for out-of-order arrivals and then for losses. This corresponds to a lossy channel, e.g., a shared network between components, over which observations are sent and where reordering occurs due to multiple senders being connected to the verdictor via a shared medium.

Timed Automata. The generic synthesis pipeline also applies to timed automata by first constructing their region graph or abstracting it with a zone graph [AD91]. Both the region and zone graph are finite transition systems, thus the pipeline can be directly applied to those. Notably, this will not take into account explicit timing information that may be attached to observations. In the next chapter, we deal with timed observations and imprecisions on the timing of observations explicitly. In addition to timed automata, the synthesis pipeline also generalizes to other infinite state systems of which finite abstractions exist in the form of transition systems.

Third-Party Techniques. Facilitated by the general theorems shown for the VTS transformations, the presented techniques can be used in tandem with third-party techniques, for instance, runtime monitors constructed with traditional runtime verification techniques [BLS06b]. Note that our techniques require the VTS to be tight, as otherwise the transformations would take spurious behavior into account and the resulting verdicts may not be as specific as possible. Should a third-party technique not synthesize a tight VTS, the VTS can be tightened before applying the respective transformations (see Definition 3.3.7).

Comparison to the Naive Algorithm. Compared to the naive algorithm (recall Section 3.4.2), the synthesis pipeline yields VTSs that can be efficiently implemented and only require bounded resources. Concretely, only a single state of the VTS needs to be stored at runtime and a lookup table can be used to implement transitions in constant time. Furthermore, synthesized VTSs are finite and can thus be analyzed effectively, e.g., to answer the question whether a certain verdict may ever be generated (which is an example of a simple VTS reachability property).

Chapter 5

Robust Continuous Time Verdictor Algorithm

Real-time systems are characterized by the fact that the timing of events plays a central role in their behavior. For instance, a prolonged or shortened delay between two events may be indicative of a fault in such a system, thus, requiring timing information to diagnose the fault. As an example, consider the timed model of the coffee machine (cf. Figure 2.2). Here, the timing of events is decisive to determine that the pump is faulty. So far, we considered observations without an explicit time component. Complementing the previous chapter, we now turn to observations with an explicit time component. By giving verdictors access to timing information, this information can be exploited for verdict generation. As in the previous chapter, the approach is generic and we focus on diagnosis for our examples.

While the timing of events plays a central role for real-time systems, its accurate assessment is usually hindered by *timing imprecisions* such as varying latency between events and their observation. We here consider *varying latency*, *varying clock drift*, and *unknown clock offsets* as well as thereby induced out-of-order observations. Those timing imprecisions are typically unavoidable when multiple components are connected over a shared network. For instance, imagine multiple controllers that make up a complex machine on a manufacturing floor. For such cases, we develop a verdictor algorithm conceptually solving significant instances of the VTS Synthesis Problem in the continuous-time setting. Verdicts obtained with this algorithm are *robust* against the aforementioned timing imprecisions, i.e., they are guaranteed to be correct and most specific despite those timing imprecisions. As for the discrete-time case, the algorithm can be used for runtime verification, fault diagnosis, and configuration monitoring alike by using different verdict domains.

The continuous-time setting is particularly challenging as we have to deal with a

continuum of possible observation times, rendering the explicit techniques taken previously infeasible. For this reason, and in contrast to the synthesis pipeline presented in the last chapter, we will not construct or transform VTSs explicitly. The approach taken here is monolithic, providing a single algorithm to handle all the involved imprecisions. The algorithm is still model-based and will take a system model in terms of a timed automaton (TA) as input. In addition, it requires a specification of the involved timing imprecisions. Conceptually, the algorithm gives rise to an infinite state VTS and we will prove it correct with respect to the theoretical framework. To this end, we will also develop an observation model.

The techniques presented here constitute the second part of Contribution FT.

Motivating Example: Industrial Automation. To concretize the intricacies that make up the problem domain, we consider a small excerpt of a manufacturing plant: Imagine a system where individual items are placed on a conveyor belt over which they are to be sorted into different processing stations. Variants of this sorting example have already been studied in a multitude of different settings ranging from real-time requirements specification [Buc+10] to formal verification [Ive+00] to teaching manufacturing systems [LFM20]. The setup we consider comprises two sensors and two grippers for identifying and then physically moving items to their respective stations. Figure 5.1 depicts such a sorting system. Here, the sensors are used to determine the size of each item on the belt.

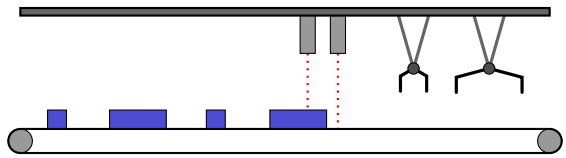


Figure 5.1: Schematic drawing of a system sorting items by size.

The conveyor belt and grippers are mechanically moving. They are therefore prone to faults caused by wear. For instance, a breakage of a ball bearing may lead to a sudden increase in friction and reduced velocity of the conveyor belt thereby slowing down production. Detecting such a fault before it results in a complete system failure is crucial to avoid unscheduled and costly downtime of production. The reduced velocity of the belt also has an impact on the timing of various events, e.g., the time difference between the first and second sensor detecting an item on the belt, and this phenomenon is intertwined with clock imprecisions and communication delays.

As an instance of the VTS Synthesis Problem, consider the problem of diagnosing faults within such kinds of systems by passively and centrally observing their behavior, e.g., by listening to messages exchanged on a shared network. The advantage of this passive approach is that it does not require any special instrumentation

and thereby is guaranteed to *not* adversely interfere with the system in any way. Active techniques would otherwise come, for instance, with additional load on the shared network, changing the timing of events (cf. Chapter 1). This advantage is inherited from traditional works on diagnosis following the same approach [e.g. Tri02; Sam+95; BCD05; Car+13; ALH06; TYG08]. Clearly, the observations obtained by passive observation are subject to timing imprecisions.

Technical Basis. For the purposes of this chapter, we assume:

(A1) The system is modeled as a timed automaton

$$\mathfrak{T} = \langle L, I, \mathsf{Act}, \mathbb{C}, E, \mathsf{Inv} \rangle$$

annotated with verdicts $\lambda_E : E \to \mathcal{V}$ of a verdict domain $\langle \mathcal{V}, \sqsubseteq \rangle$.

(A2) A subset $\mathsf{OAct} \subseteq \mathsf{Act}$ of the actions Act is observable.

(A3) Observation times are rationals as they are discretized by the verdictor.

(A4) Every occurrence of an observable action within the system constitutes an event and causes exactly one corresponding observation.

(A5) As long as the system keeps running, always eventually an observable action will occur. Formally, the region graph [see AD91] of $\mathfrak{T}$ must not contain any cycles that are free of observable actions.

As in the discrete-time case, we start with an annotated system model. Here, this model is a timed automaton instead of a transition system (TS). Just annotating transitions ensures monotonicity and suffices for fault diagnosis, without transient faults, and for configuration monitoring. We later discuss how the assumption (A1) can be relaxed and how state labels can be handled at the price of weakened guarantees.

Assumption (A2) is again motivated by observability limitations inherent to most systems. Assumption (A3) is rooted in the fact that the resolution of hardware clocks is limited and, within a verdictor, we always need to represent timestamps in some discretized manner. Assumption (A4) means that no observations are lost and also no spurious observations are produced. We make this assumption because the continuous-time setting is already challenging even if lost or spurious observations cannot occur. Assumption (A5) is required to guarantee termination of the presented algorithm—it is also a realistic assumption in practice.

As for the discrete-time case, we define a verdict oracle based on the annotations of the system model. Note that the theoretical framework is defined in terms of a TS system model, not a TA. Therefore, we define the verdict oracle for the TS semantics $[\![\mathfrak{T}]\!]$ of the TA $\mathfrak{T}$. To this end, we can then reuse the definition for the

verdict annotations in the discrete-time setting (recall Definition 4.1.1), by defining the three functions κ, λ, and γ, as follows:

$$\kappa(s) \coloneqq \top \qquad \lambda(s) \coloneqq \top \qquad \gamma(\langle\langle l, \eta\rangle, a, \langle l', \eta'\rangle\rangle) \coloneqq \lambda_E(l, a, l') \tag{5.1}$$

The transitions $\langle\langle l, \eta\rangle, a, \langle l', \eta'\rangle\rangle$ of the transition system semantics of the TA system model are annotated based on the annotations of the corresponding edges as per assumption (A1). For now, states are simply annotated with the least specific verdict $\top$ of $\mathcal{V}$, as we restrict annotations to edges. As already hinted at above, this restriction to edge verdict labels will later be lifted. For now, the verdict oracle for the continuous-time case is given by $\mathsf{V}_{\kappa,\lambda,\gamma}$ as per Definition 4.1.2. It is easy to see that this verdict oracle is monotonic—which will become important later.

Proposition 5.0.1 *The verdict oracle $\mathsf{V}_{\kappa,\lambda,\gamma}$ based on (5.1) is monotonic.*

Proof Sketch. Proven by induction on the run. □

Note that the verdict oracle is defined over the verdict domain extended with the sentinel bottom verdict #. As in the discrete-time case, the sentinel verdict is either not produced or it corresponds to observation sequences considered unrealistic.

Example 5.1 For the industrial automation example discussed above, we may consider two observable actions, `trigger_0`, `trigger_1` $\in$ OAct, indicating when an item entered the field of view of the first and second sensor respectively, and an action `fault_bearing` $\notin$ OAct corresponding to a bearing fault whereafter the conveyor belt experiences increased friction. Figure 5.2 shows a timed automaton modeling the system in question. Here, a fault of the ball bearings may happen at any time and influence the time between `trigger_0` and `trigger_1` events (normally: 3 ms to 6 ms; faulty: 6 ms to 12 ms). In both cases, items enter the field of view of the first sensor 10 ms to 200 ms after the previous item entered the field of view of the second sensor. For the purposes of diagnosis, we annotate edges with sets of fault classes as per Definition 3.1.3. Following traditional model-based diagnosis, these annotations are analogous to the discrete-time setting (recall Section 4.1.1): Edges that do not correspond to a fault are annotated with the empty set and edges that do correspond to a fault are annotated with the respective singleton set of the fault's class. For the example, we denote the fault class of a bearing fault by f_B.

Relevant Publications. This chapter generalizes work done by the author of this thesis specifically for diagnosis. It is largely based on the following paper:

> [KH23]: Köhl and Hermanns (2023), *Model-Based Diagnosis of Real-Time Systems: Robustness Against Varying Latency, Clock Drift, and Out-of-Order Observations*

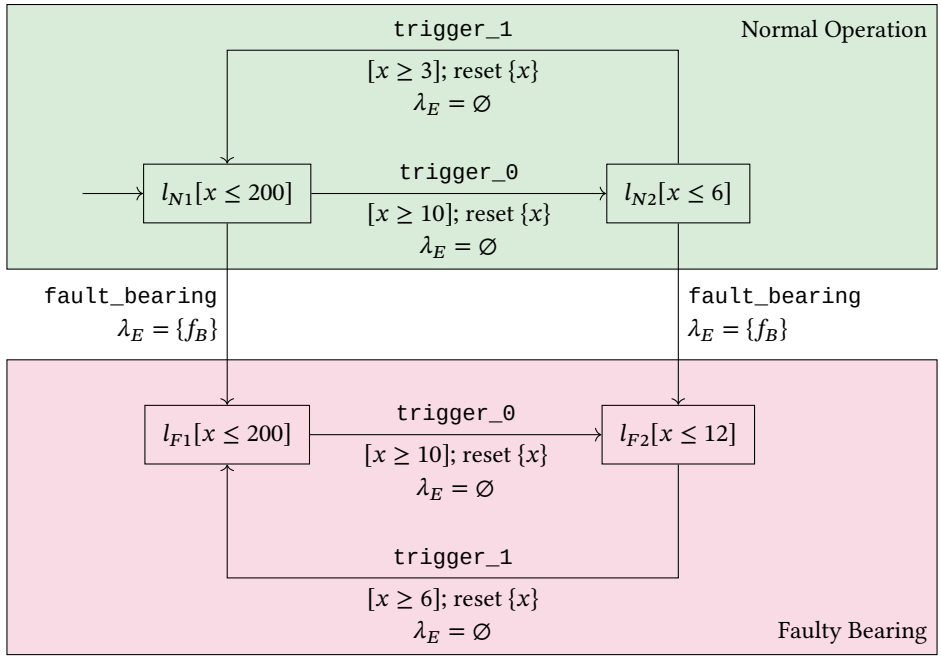


Figure 5.2: Example for a timed automaton model of the excerpt of the industrial automation system with a bearing fault as described in Example 5.1, one clock variable x, and edge clock guards and invariants in brackets. Time is given in ms. Verdict annotations as per λ_E are indicated with $\lambda_E = v$ as part of the edges.

Exceeding the contributions of the original paper, this thesis generalizes the ideas and concepts to harvest them within the more general theoretical framework underlying this thesis. In particular, this opens up the usage of the techniques for other applications than diagnosis such as runtime verification or configuration monitoring.

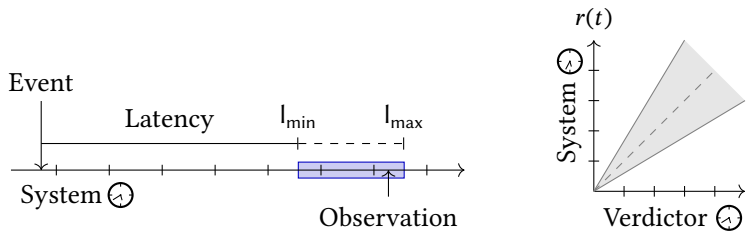
Chapter Structure. Section 5.1 formalizes the timing imprecisions mentioned above. Section 5.2 presents an observation model for the continuous-time case with timing imprecisions. Section 5.3 then develops basic building blocks towards the verdictor algorithm. Finally, Section 5.4 presents the verdictor algorithm as well as an approximation that only requires a bounded amount of memory and discusses how the algorithm can be generalized beyond edge annotations. Section 5.5 concludes this chapter and summarizes its contributions.

5.1 Timing Imprecisions

For real-time systems, the accurate assessment of the timing of events within a system is usually hindered by three well-known and orthogonal phenomena: *varying latency*, *varying clock drift*, and *unknown clock offsets*.

Latency is the time it takes after an event occurs until it is observed. The variability of latency is called *jitter*. In our industrial automation example, the shared network introduces a varying latency between events and their observation. Figure 5.3a visualizes the time interval (shaded rectangle) for making an observation, relative to the system's clock. It is the result of accounting for any possible *but bounded* latency within the interval $[l_{min}, l_{max}]$. The interval captures possible bounded jitter. Such *latency bounds* are justified by the hard timing guarantees of the shared network and can be obtained via standard timing analyses [e.g. THW94; Pop+06; SCT10].

Clock drift refers to the unavoidable imprecisions concerning the speed of hardware clocks. For instance, quartz-based hardware clocks suffer from inaccuracies due to temperature variations.[11] Usually, manufacturers guarantee a certain upper bound on inaccuracy for their products. As a result, starting with some *initial offset*, the clocks of the verdictor and the system may *drift* apart up to some specified bound as indicated by the shaded cone in Figure 5.3b. Note that while from the initial offset any point in the cone can be reached, this does not mean that from any point in the cone all future points in the cone can be reached. Instead, every point in the cone represents an offset of both clocks from which a new cone emerges. The dashed line represents perfect but unrealistic clock synchronicity.



(a) Delayed observation of events.

(b) Drifting clocks.

Figure 5.3: Timing imprecisions when observing real-time systems.

Model-based diagnosis in the real-time setting based on timed automata has been addressed in the existing literature before [Tri02; BCD05; Mha+17]. By considering the aforementioned timing imprecisions, we relax the strict assumption that the occurrence times of events within the system can be precisely observed. For a realistic

[11] https://web.archive.org/web/20240226014014/https://blog.bliley.com/crystal-oscillator-stability

observation model, we consider the superimposition of all these phenomena just discussed. To this end, we first model them formally in isolation.

Varying Latency. We assume events to be observed with varying latency, as in Figure 5.3a. For the set OAct of observable actions, we assume given a *minimal latency* function $\mathsf{l}_{\min} : \mathrm{OAct} \to \mathbb{Q}$ and a *maximal latency* function $\mathsf{l}_{\max} : \mathrm{OAct} \to \mathbb{Q}$ such that the minimal latencies are less than the corresponding maximal latencies, respectively, i.e., $\mathsf{l}_{\min}(a) < \mathsf{l}_{\max}(a)$ for all $a \in \mathrm{OAct}$. These functions capture varying but bounded latency per observable action relative to the system's clock. Hence, it takes at least $\mathsf{l}_{\min}(a)$ and at most $\mathsf{l}_{\max}(a)$ time units after an occurrence of $a \in \mathrm{OAct}$ until it is observed. Note that latencies are rationals. This is justified as observation times themselves are also rationals (cf. assumption (A3)).

Clock Drift and Offsets. In addition to varying latency, we also consider clock drift and offsets. Clock drift has already been extensively studied in the timed automata literature [Pur98; SFK08; Wul+08]. We follow the established model and capture the accuracy guarantee provided by manufacturers of hardware clocks as a parameter $\delta \geq 0$ bounding the amount a clock may run faster or slower than the actual time, in our case the system's clock [SFK08]. We encapsulate the clock drift *and* initial offset of both clocks in an unknown *time-remapping function* $r : \mathbb{R} \to \mathbb{R}$ mapping verdictor time t to system time $r(t)$ (see Figure 5.3b). That is, at a given time t of the verdictor's clock, the value of the system's clock is $r(t)$. As the speed deviation between the clocks is bounded by δ, the slope of the function r must be restricted. Formally, these considerations give rise to the following definition:

Definition 5.1.1 *Given a* drift parameter $\delta \in \mathbb{Q}_{\geq 0}$*, a* time-remapping function $r : \mathbb{R} \to \mathbb{R}$ *is a strictly monotone continuous bijection with a slope bounded by:*

$$\forall t' > t : \frac{r(t') - r(t)}{t' - t} \in \left[\frac{1}{1+\delta}, 1+\delta\right] \tag{5.2}$$

Timed automata with drifting clocks have been introduced by Puri [Pur98] whose general model of slope restriction we follow here. Instead of restricting the slope to $[1-\epsilon, 1+\epsilon]$, we follow Swaminathan et al. [SFK08] and restrict it according to (5.2). Note that we do not require r to be differentiable, hence, the amount of clock drift can change discretely at any point in time. We furthermore only restrict the speed deviation of the system's clock by (5.2) but not it's absolute difference to the verdictor's clock. In addition, we also make no assumption about the initial offset of both clocks as any such assumption would be ad-hoc.

Note that events occurring at well-defined moments in time, e.g., at system startup, can be used for synchronization of both clocks up to the actual latency of the event

and its observation. The to-be-developed techniques will transparently take into account the timing information carried by observations of such synchronization events. Active synchronization, on the other hand, would violate our goal that a verdictor shall be completely passive, an assumption made to avoid any adverse interferences with the system (cf. Chapter 1).

In the following, we will use ${}^{\uparrow}t$ to denote the time an event occurred relative to the system's clock and ${}^{\downarrow}t$ to denote the time an event is observed relative to the verdictor's clock.

Example 5.2 Recall Example 5.1 and imagine that the sensors are connected to a shared CAN bus. A shared CAN bus with a background load may realistically introduce a latency between 1 ms and 3 ms [BBR02]. Hence, it takes between 1 ms and 3 ms from when an item enters the field of view until the respective action is observed on the bus. Formally, let $l_{\text{min}}(\texttt{trigger\_i}) \coloneqq 1\,\text{ms}$ and $l_{\text{max}}(\texttt{trigger\_i}) \coloneqq 3\,\text{ms}$ for $\texttt{i} \in \{0, 1\}$. Now, consider that a `trigger_0` and a `trigger_1` event take place precisely 5 ms apart at time ${}^{\uparrow}t_0 = 12\,\text{ms}$ and ${}^{\uparrow}t_1 = 17\,\text{ms}$ relative to the system's clock, respectively. As these events occurred 5 ms apart, a verdictor with precise access to this timing information may correctly conclude that the conveyor belt functions nominally (at least up to the point of the `trigger_1` event).

Now, consider that the times a verdictor observes are subject to the timing imprecisions discussed previously. As a result, the events may be observed, e.g., at time ${}^{\downarrow}t_0 = 212\,\text{ms}$ and ${}^{\downarrow}t_1 = 219\,\text{ms}$ relative to the verdictor's own clock, respectively. Recall that we make no assumptions about the initial offset of the system's and the verdictor's clock. The time ${}^{\downarrow}t_0 = 212\,\text{ms}$, at which the first event is observed, is the result of the initial offset of both clocks, the drift which has occurred since then, and the latency. For instance, disregarding drift and assuming that the actual latency has been 1 ms, the initial offset must have been 199 ms because $12\,\text{ms} + 1\,\text{ms} + 199\,\text{ms} = 212\,\text{ms}$.[12] While the events are observed 7 ms apart, which is more than the 6 ms for a normally-functioning conveyor belt, the verdictor must not conclude that there has been a fault because the additional delay between both observations can be accounted for by considering the latency of observations in the system and, indeed, the system is not faulty: The first event may have been observed with a latency of 1 ms and the second with a latency of 3 ms explaining the additional 2 ms.

In contrast, imagine a `trigger_0` and a `trigger_1` event that take place precisely 12 ms apart[13] at time ${}^{\uparrow}t'_0 = 12\,\text{ms}$ and ${}^{\uparrow}t'_1 = 24\,\text{ms}$, respectively, i.e., the

[12] We assume that neither the initial offset nor the actual drift nor the actual latency is directly known to the verdictor. The verdictor can only indirectly infer information about them from the observations made. The theory and verdictor algorithm we present enable precisely that.

[13] The reader is invited to play through other cases where the time difference is not as extreme. The theory we present here will enable us to understand precisely when a verdictor can and must conclude from observations that a fault occurred and when this is not the case. For simplicity, this example also disregards clock drift and out-of-order observations which we address subsequently.

conveyor belt must be experiencing increased friction caused by a faulty bearing. In this case, even if both events are observed with the minimal latency of 1 ms at ${}^{\downarrow}t'_0 = 212\,\text{ms}$ and ${}^{\downarrow}t'_1 = 224\,\text{ms}$, respectively, the verdictor should be able to conclude that the bearings failed after observing `trigger_1`. In this case, the large time difference between both observations cannot be accounted for by considering the latency of observations. The observations can only be explained (based on the timing imprecisions and system model) with faulty bearings.

Example 5.2 shows that timing imprecisions must be taken into account by a verdictor to produce correct (and most specific) verdicts, e.g., not indicating a fault even though the time difference between the observations of `trigger_0` and `trigger_1` is above 6 ms. Instantiating the theoretical framework established in Chapter 3, it remains to define an observation model based on the presented timing imprecisions. In the example, we already considered an intuitive correspondence between the timing of events and their observations. The observation model established in the following, will capture such a correspondence formally.

5.2 Observation Model

To obtain guarantees according to our model-based methodology with respect to the theoretical framework, we need to define an observation model capturing the introduced timing imprecisions. At their core, observation models relate events within the system with possible observations. As observation times are discretized (A3) and observations are occurrences of observable actions (A4), observables naturally are pairs $\langle {}^{\downarrow}t, a\rangle$ where ${}^{\downarrow}t \in \mathbb{Q}$ is the *observation time* and $a \in \text{OAct}$ is an observable action. We call such pairs *timed observables*. A *timed observation sequence* is a finite sequence of timed observables being observed one after the other.

Definition 5.2.1 *A* timed observation sequence *over a set* OAct *of observable actions is a finite sequence* $\omega = \left(\langle {}^{\downarrow}t_i, a_i\rangle\right)_{i=1}^{n} \in (\mathbb{Q} \times \text{OAct})^{\star}$ *such that* ${}^{\downarrow}t_i \leq {}^{\downarrow}t_{i+1}$ *for all* $1 \leq i < n$.

A *timed observation* is a tuple $\theta = \langle j, \langle {}^{\downarrow}t, a\rangle\rangle \in \omega$ in a timed observation sequence ω, where j is the position of the timed observable $\langle {}^{\downarrow}t, a\rangle$ in ω. Recall that sequences are partial functions which are sets of pairs (cf. Section 2.1).

Traditionally, *timed words* have been defined as sequences over $\mathbb{R}_0^+ \times \text{Act}$ [AD91]. For timed observation sequences, we follow the definition of Tripakis [Tri02] and use rationals instead of reals. This is rooted in the fact that the resolution of any hardware clock is limited and the clock will thus tick in discrete time steps, leading to the assignment of discrete timestamps to observations (A3). Notably, within the system, events may still happen at non-rational points in time. For instance, if a

timed automaton is used to model physical processes, then the occurrence times of events are not bound to the discrete nature of hardware clocks.

As a counterpart to timed observation sequences, we further define *timed event sequences* as follows:

Definition 5.2.2 *A* timed event sequence *over a set* OAct *of observable actions is a finite sequence* $\varrho = \left(\langle {}^{\uparrow}t_i, a_i\rangle\right)_{i=1}^{n} \in \left(\mathbb{R}_0^+ \times \mathrm{OAct}\right)^\star$ *such that* ${}^{\uparrow}t_i \leq {}^{\uparrow}t_{i+1}$ *for all* $1 \leq i < n$.

Each run $\rho = \left(\langle s_i, \mathrm{a}_i, s_i'\rangle\right)_{i=1}^{n}$ of the TS semantics $[\![\mathfrak{T}]\!]$ of the system model $\mathfrak{T}$ induces a timed event sequence $\varrho(\rho)$, inductively defined as follows:

$$\varrho(\rho) := \epsilon \qquad \varrho(\rho \diamond \langle s, \mathrm{a}, s'\rangle) := \begin{cases} \varrho(\rho) \diamond \langle \mathrm{Dur}(\rho), \mathrm{a}\rangle & \text{if a} \in \mathrm{OAct} \\ \varrho(\rho) & \text{otherwise} \end{cases} \tag{5.3}$$

That is, whenever an observable action occurs, the time associated with the respective event is the time that has passed so far on the run. Analogously to timed observations, a *timed event* is a tuple $e = \langle i, \langle {}^{\uparrow}t, a\rangle\rangle \in \varrho$ in a timed event sequence ϱ, where i is the position of $\langle {}^{\uparrow}t, a\rangle$ in ϱ and ${}^{\uparrow}t$ is the event's *occurrence time*.

Example 5.3 Coming back to Example 5.2, we are now in a position to formally describe the first situation as follows: Two trigger events, $e_0 = \langle 0, \langle 12\,\mathrm{ms}, \texttt{trigger\_0}\rangle\rangle$ and $e_1 = \langle 1, \langle 17\,\mathrm{ms}, \texttt{trigger\_1}\rangle\rangle$, took place precisely 5 ms apart. Each event led to an observation, $\theta_0 = \langle 0, \langle 212\,\mathrm{ms}, \texttt{trigger\_0}\rangle\rangle$ and $\theta_1 = \langle 1, \langle 219\,\mathrm{ms}, \texttt{trigger\_1}\rangle\rangle$, respectively. The observations are 7 ms apart and give rise to a timed observation sequence ω which has been observed by the verdictor. Analogously, the events are part of a timed event sequence ϱ. We are aiming for the technical tools to answer the following question with mathematical rigor: Can the verdictor conclude only from the observation sequence ω, i.e., without having any additional information about the actual events, that a bearing fault `fault_bearing` has occurred? While we already know intuitively that the observations in ω may be caused by two events, namely e_0 and e_1, which are not indicative for a fault when accounting for latency as specified in Example 5.2, a mathematical correspondence capturing this is needed. The second situation where events take place precisely 12 ms apart is captured analogously. However, in this case, the verdictor can conclude only from the set of observations that a bearing fault `fault_bearing` must have occurred.

5.2.1 Occurrence and Observation Times

As the first step towards an observation model, we need to relate observation times with the occurrence times of events. As per assumption (A4), each timed observation $\theta = \langle j, \langle {}^{\downarrow}t, a\rangle\rangle \in \omega$ is caused by a corresponding occurrence of a timed event $e =$

$\langle i, \langle {}^{\uparrow}t, a\rangle\rangle \in \wp$ in the system. In the previous section, we introduced the unavoidable timing imprecisions consisting of (a) the clock drift and offset of the verdictor's clock relative to the system's clock according to some unknown time-remapping function r satisfying (5.2), and (b) the maximal and minimal latency functions $\mathsf{l}_{\max}$ and $\mathsf{l}_{\min}$ assigning a varying latency to each observable action. Combining these imprecisions and assuming r to be given, a timed event $\langle i, \langle {}^{\uparrow}t, a\rangle\rangle$ may cause a timed observation $\langle j, \langle {}^{\downarrow}t, a\rangle\rangle$ at any time ${}^{\downarrow}t$ in the *observation time interval*:

$$ {}^{\downarrow}T_r(\langle i, \langle {}^{\uparrow}t, a\rangle\rangle) \coloneqq \left[r^{-1}({}^{\uparrow}t + \mathsf{l}_{\min}(a)), r^{-1}({}^{\uparrow}t + \mathsf{l}_{\max}(a))\right] \tag{5.4} $$

This is because for an event $\langle i, \langle {}^{\uparrow}t, a\rangle\rangle$ occurring at time ${}^{\uparrow}t$ it takes between $\mathsf{l}_{\min}(a)$ and $\mathsf{l}_{\max}(a)$ relative to the system's clock until it is observed. Thus it will be observed at some time in the interval $\left[{}^{\uparrow}t + \mathsf{l}_{\min}(a), {}^{\uparrow}t + \mathsf{l}_{\max}(a)\right]$. This interval is relative to the system's clock and does not yet account for clock drift and offsets. Assuming that we know the time-remapping function r, we can convert system time to verdictor time by applying its inverse r^{-1}. As r is a bijection, its inverse r^{-1} must exist. Thereby, we obtain the interval as defined in (5.4). Figure 5.4 visualizes this definition.

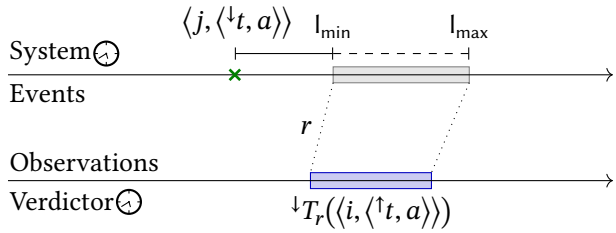


Figure 5.4: The observation time interval ${}^{\downarrow}T_r(\langle i, \langle {}^{\uparrow}t, a\rangle\rangle)$ of an event $\langle i, \langle {}^{\uparrow}t, a\rangle\rangle$ is the time interval in which the event must be observed when accounting for unknown varying latency together with known clock drift and offset as given by the function r.

Analogously to (5.4), we also obtain an *occurrence time interval* for each timed observation $\langle j, \langle {}^{\downarrow}t, a\rangle\rangle$, capturing the time interval in which the event causing the observation may have occurred:

$$ {}^{\uparrow}T_r(\langle j, \langle {}^{\downarrow}t, a\rangle\rangle) \coloneqq \left[r({}^{\downarrow}t) - \mathsf{l}_{\max}(a), r({}^{\downarrow}t) - \mathsf{l}_{\min}(a)\right] \tag{5.5} $$

The definition (5.5) is essentially the inverse of (5.4). Figure 5.5 visualizes the idea behind it. While (5.4) takes the perspective of the system and asks when a certain event may be observed, (5.5) takes the perspective of the verdictor and asks when an event causing a certain observation may have occurred.

Given that observation and occurrence times are different sides of the same coin, we establish the following relation between them:

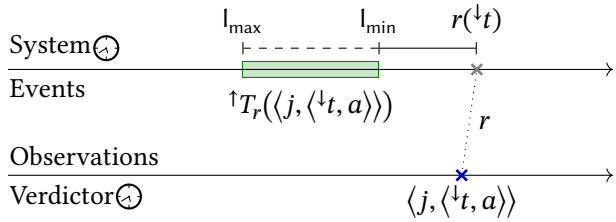


Figure 5.5: The occurrence time interval $^{\uparrow}T_r(\langle j, \langle ^{\downarrow}t, a\rangle\rangle)$ of an observation $\langle j, \langle ^{\downarrow}t, a\rangle\rangle$ is the time interval in which the event causing the observation must have occurred when accounting for unknown varying latency together with known clock drift and offset as given by the function r.

Lemma 5.2.1 *For all observable actions $a \in \mathrm{OAct}$, observation times $^{\downarrow}t \in \mathbb{Q}$, occurrence times $^{\uparrow}t \in \mathbb{R}$, indices $j, i \in \mathbb{N}$, and time-remapping functions r as per Definition 5.1.1, the following condition holds:*

$$^{\uparrow}t \in {}^{\uparrow}T_r(\langle j, \langle ^{\downarrow}t, a\rangle\rangle) \iff {}^{\downarrow}t \in {}^{\downarrow}T_r(\langle i, \langle ^{\uparrow}t, a\rangle\rangle) \tag{5.6}$$

For a detailed proof, see Appendix A.2.1.

Difference Bound. The intervals we introduced capture how occurrence and observation times relate to each other. In the following, we will also need two further technical notations, *difference bounds* and a partial order $\prec$ on observations.

Using the occurrence time interval, we obtain an upper bound on the time difference between the events causing two timed observations θ and θ':

$$\mathrm{MaxD}(\theta, \theta') := \max_r \left(\max {}^{\uparrow}T_r(\theta) - \min {}^{\uparrow}T_r(\theta')\right) \tag{5.7}$$

Here, $\max_r$ ranges over *all* time-remapping functions as per Definition 5.1.1. By maximizing over all time remapping functions, an upper bound independent of any specific r is obtained, reflecting that we cannot assume to know the actual function r. For any two timed observations θ and θ', the *difference bound* $\mathrm{MaxD}(\theta, \theta')$ is an upper bound on the difference $^{\uparrow}t - {}^{\uparrow}t'$ between (i) the occurrence time $^{\uparrow}t$ of an event $\langle ^{\uparrow}t, a\rangle$ causing θ and (ii) the occurrence time $^{\uparrow}t'$ of an event $\langle ^{\uparrow}t', b\rangle$ causing θ'. Despite the fact that there are uncountably many time-remapping functions, we can still compute MaxD due to the slope restriction (5.2). The difference bound $\mathrm{MaxD}(\theta, \theta')$ can be computed for any pair $\langle \theta, \theta'\rangle$ of observations:

Lemma 5.2.2 *Given the drift parameter δ (recall Definition 5.1.1), the following equation holds for any two timed observations $\langle j, \langle {}^{\downarrow}t, a\rangle\rangle$ and $\langle j', \langle {}^{\downarrow}t', a'\rangle\rangle$:*

$$\begin{aligned} &\mathrm{MaxD}\Big(\langle j, \langle {}^{\downarrow}t, a\rangle\rangle, \langle j', \langle {}^{\downarrow}t', a'\rangle\rangle\Big) \\ = \;& \mathsf{I}_{\max}(a') - \mathsf{I}_{\min}(a) + \begin{cases} (1+\delta)({}^{\downarrow}t - {}^{\downarrow}t') & \text{if } {}^{\downarrow}t \geq {}^{\downarrow}t' \\ (1+\delta)^{-1}({}^{\downarrow}t - {}^{\downarrow}t') & \text{otherwise} \end{cases} \end{aligned}$$

Proof Sketch. The proof of Lemma 5.2.2 relies on the slope restriction (5.2) which enables the computation of extrema, in particular, the maximum over all time remapping functions as in (5.7), by considering the drift bounds. For a detailed proof see Appendix A.2.2. □

Using MaxD, we further define a partial order $\prec$ on observations:

$$\theta \prec \theta' \text{ if and only if } \mathrm{MaxD}(\theta, \theta') < 0 \tag{5.8}$$

For any two observations θ and θ', we have $\theta \prec \theta'$ if and only if the event e' causing θ' must have occurred after the event e causing θ. Recall that $\mathrm{MaxD}(\theta, \theta')$ is an upper bound on the difference ${}^{\uparrow}t - {}^{\uparrow}t'$ of the occurrence times of the events causing θ and θ', respectively. Now, if $\mathrm{MaxD}(\theta, \theta') < 0$, then ${}^{\uparrow}t - {}^{\uparrow}t' < 0$ and, thus, ${}^{\uparrow}t < {}^{\uparrow}t'$, i.e., the event e' causing θ' must have occurred after the event e causing θ.

Example 5.4 Consider the observations $\theta_0 = \langle 0, \langle 212\,\mathrm{ms}, \texttt{trigger\_0}\rangle\rangle$ and $\theta_1 = \langle 1, \langle 219\,\mathrm{ms}, \texttt{trigger\_1}\rangle\rangle$ as defined in Example 5.3. Further, recall from Example 5.2 that $\mathsf{I}_{\min}(\texttt{trigger\_i}) = 1\,\mathrm{ms}$ and $\mathsf{I}_{\max}(\texttt{trigger\_i}) = 3\,\mathrm{ms}$ for $i \in \{0, 1\}$. Assume that the clock accuracy is $\pm 3\,\%$, i.e., $\delta = 0.03$. Applying Lemma 5.2.2, we obtain:

$$\begin{aligned} &\mathrm{MaxD}(\theta_0, \theta_1) \\ =\;& \mathsf{I}_{\max}(\texttt{trigger\_1}) - \mathsf{I}_{\min}(\texttt{trigger\_0}) + (1+\delta)^{-1} \cdot (212\,\mathrm{ms} - 219\,\mathrm{ms}) \\ \approx\;& 3\,\mathrm{ms} - 1\,\mathrm{ms} - 6.796\,\mathrm{ms} \\ =\;& -4.796\,\mathrm{ms} \end{aligned}$$

Hence, the difference ${}^{\uparrow}t_0 - {}^{\uparrow}t_1$ of the occurrence times ${}^{\uparrow}t_0$ and ${}^{\uparrow}t_1$ of the events causing θ_0 and θ_1, respectively, is less than $-4.796\,\mathrm{ms}$. In other words, the event e_0 causing θ_0 occurred at least 4.796 ms before the event e_1 causing θ_1. Indeed, in Example 5.3, the event e_0 occurred $5\,\mathrm{ms} \geq 4.796\,\mathrm{ms}$ before the event e_1.

5.2.2 Consistency of Events and Observations

The intervals introduced previously relate occurrence times of events with possible observation times of the observations they cause. If we look at a system run, then it induces multiple timed events and each of these events must cause an observation

such that the time of the observation is in the respective interval (recall assumption (A4)). Note that the actual latency may vary independently for each observation, however, the time-remapping function should be the same for all observations, as all observations are timestamped with the verdictor's clock.

These considerations give rise to a notion of *consistency* between timed observation sequences and timed event sequences. Consistency requires that events and observations must match up one-to-one such that their times align according to the intervals we introduced with respect to some common time-remapping function. Formally, we define consistency as follows:

Definition 5.2.3 *A timed event sequence* $\varrho = \left(\left\langle {}^{\uparrow}t_i, a_i \right\rangle\right)_{i=1}^{n}$ *and a timed observation sequence* $\omega = \left(\left\langle {}^{\downarrow}t_j, b_j \right\rangle\right)_{j=1}^{n}$ *are* consistent *iff there exists a time-remapping function* r *and a bijection* $\mathcal{R} : \varrho \to \omega$ *mapping events to observations such that* $a = b$ *and* ${}^{\downarrow}t \in {}^{\downarrow}T_r(\langle {}^{\uparrow}t, a \rangle)$ *for all* $\left\langle \cdot, \left\langle {}^{\uparrow}t, a \right\rangle \right\rangle \in \varrho$ *where* $\left\langle \cdot, \left\langle {}^{\downarrow}t, b \right\rangle \right\rangle = \mathcal{R}\left(\left\langle \cdot, \left\langle {}^{\uparrow}t, a \right\rangle \right\rangle\right)$.

In the following, we denote consistency of ϱ with ω by $\varrho \rhd \omega$.

Note that Definition 5.2.3 rests on the core assumption (A4) that every event will eventually cause exactly one observation. This assumption is enshrined in the bijection $\mathcal{R}$ which maps each event to the observation it causes.

In words, consistency requires that there exists a single time-remapping function capturing clock drift and offsets of the system's and verdictor's clock over time such that the observation times of all observations are within the observation time interval of the events causing the respective observations.

Leveraging (5.3), it is easy to lift Definition 5.2.3 to runs of the TS semantics $[\![\mathfrak{T}]\!]$ of the timed automaton system model $\mathfrak{T}$. A run ρ is then consistent with a timed observation sequence ω iff $\varrho(\rho) \rhd \omega$.

Example 5.5 Recall Example 5.3, and consider the following timed event sequence induced by the earlier discussed run containing the two events e_0 and e_1:

$$\varrho = \underbrace{\langle 12\,\text{ms}, \texttt{trigger\_0} \rangle}_{e_0} \diamond \underbrace{\langle 17\,\text{ms}, \texttt{trigger\_1} \rangle}_{e_1}$$

Recall that those events gave rise to the following observation sequence:

$$\omega = \underbrace{\langle 212\,\text{ms}, \texttt{trigger\_0} \rangle}_{\theta_0} \diamond \underbrace{\langle 219\,\text{ms}, \texttt{trigger\_1} \rangle}_{\theta_1}$$

In this case, the bijection mapping e_0 to θ_0 and e_1 to θ_1 together with the time-remapping function $r(t) = 0.99t - 196.81\,\text{ms}$ witnesses consistency between ϱ and ω. To see that this is the case, let us first map the observation times to the system time: $r(212\,\text{ms}) = 13.07\,\text{ms}$ and $r(219\,\text{ms}) = 20\,\text{ms}$. Based on this, we now compute

the occurrence time intervals as per (5.5):

$$
\begin{aligned}
{}^{\uparrow}T_r(\theta_0) &= [r(212\,\text{ms}) - 3\,\text{ms}, r(212\,\text{ms}) - 1\,\text{ms}] = [10.07\,\text{ms}, 12.07\,\text{ms}] \ni 12\,\text{ms} \\
{}^{\uparrow}T_r(\theta_1) &= [r(219\,\text{ms}) - 3\,\text{ms}, r(219\,\text{ms}) - 1\,\text{ms}] = [17\,\text{ms}, 19\,\text{ms}] \ni 17\,\text{ms}
\end{aligned}
$$

Consistency of ω and ϱ tells us that the observations ω may indeed have been caused by the events ϱ. In contrast, ϱ is *not* consistent with the second example where there have been 12 ms between the two trigger observations, i.e., where:

$$
\omega' = \underbrace{\langle 212\,\text{ms}, \texttt{trigger\_0} \rangle}_{\theta'_0} \diamond \underbrace{\langle 224\,\text{ms}, \texttt{trigger\_1} \rangle}_{\theta'_1}
$$

This is because there exists no bijection $\mathcal{R}$ and time-remapping function instantiating Definition 5.2.3: The large delay between the observations does not allow for a match with the events on ϱ, even when taking the timing imprecisions into account.

Naive Observation Model. A naive definition of an observation model may now simply define the set of timed observation sequences induced by a given run ρ as those timed observation sequences that are consistent with ρ:

$$
\Omega(\rho) \coloneqq \left\{ \omega \in (\mathbb{Q} \times \text{OAct})^{\star} \mid \varrho(\rho) \rhd \omega \right\} \tag{5.9}
$$

Unfortunately, (5.9) has several problems. First of all, it is not very realistic as it assumes that all events of a run have been observed, when in fact, there may still be observations missing due to latency. For instance, consider a run where first a occurs and then, after 5 s, b occurs. Now, according to (5.9), both events, a and b, must have been observed, when in fact, b may only arrive later due to its latency.

Related to this problem is the fact that the observation model as per (5.9) does not satisfy the condition required as per Definition 3.3.1. Take the same example as before and assume that the run continues such that c happens immediately after b. Assume further that c has a maximal latency of 3 ms while b has a minimal latency of 6 ms. Now, we cannot continue the original observation sequence which required us to observe a and b because c needs to be observed before b.

We encountered a similar problem in the discrete-time setting when considering out-of-order observations (see Section 4.3.4) caused by observations that are delayed individually. In contrast to the discrete-time setting, we now have concrete possible latencies. To tackle these problems, the observation model must capture all possible out-of-order observations induced by such latencies. In the discrete-time setting, we simply handled out-of-order observations by bounded permutations and delays, implicitly assuming that the possible (discrete) latencies of all observables are identical. Handling the varying, individual latencies in the continuous-time setting is more challenging and, thus, requires a more complex formal machinery.

5.2.3 Out-of-Order Observations

As the result of varying latency, observations may be made in a different order than the events causing them ocurred in. Figure 5.6 visualizes this phenomenon. Here, the event e_1 ocurred before the event e_2, however, at time t of the verdictor's clock, the verdictor made the observations ω, including the observation of e_2 but lacking the observation of e_1. Only later will the verdictor observe e_1, thereby extending ω to ω'. Thus, in this example, ω is an *incomplete prefix* in the sense that it lacks certain observations of events which occurred before already observed events. Due to this incompleteness, it is also not consistent with the ongoing run. It may not even be consistent with any run, e.g., when e_1 must always happen before e_2.

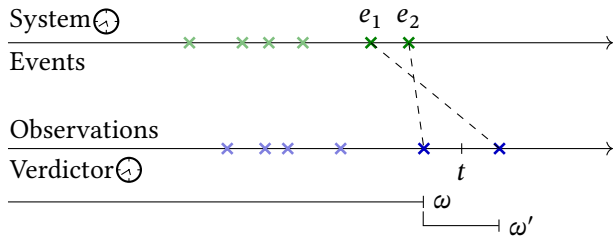


Figure 5.6: Latency induced out-of-order observations leading to an incomplete prefix.

Example 5.6 Assume for the sake of this example, that the latency of trigger actions in the industrial automation example varies between 1 ms and 50 ms (instead of 1 ms and 3 ms). Now, take the event sequence as introduced in Example 5.3:

$$\varrho = \underbrace{\langle 12\,\text{ms}, \texttt{trigger\_0}\rangle}_{e_0} \diamond \underbrace{\langle 17\,\text{ms}, \texttt{trigger\_1}\rangle}_{e_1}$$

The ongoing run ρ inducing these events as per (5.3) may start as follows and continue in some arbitrary way:

$$\rho = \langle s_1, 12\,\text{ms}, s_2\rangle \underbrace{\langle s_2, \texttt{trigger\_0}, s_3\rangle}_{e_0} \langle s_3, 5\,\text{ms}, s_4\rangle \underbrace{\langle s_3, \texttt{trigger\_1}, s_4\rangle}_{e_1} \cdots \quad (5.10)$$

Due to the highly varying latency, it is possible that e_1 is observed with a latency of only 1 ms at time ${}^{\downarrow}t_1 = 25\,\text{ms}$, while the observation of e_0 is made much later with a latency of 40 ms at time ${}^{\downarrow}t_0 = 59\,\text{ms}$.[14] This leads to the following observation sequence:

$$\omega' = \underbrace{\langle 25\,\text{ms}, \texttt{trigger\_1}\rangle}_{\theta'_1} \diamond \underbrace{\langle 59\,\text{ms}, \texttt{trigger\_0}\rangle}_{\theta'_0}$$

[14] For instance, the time-remapping function $r(t) = t - 7$ witnesses consistency because ${}^{\downarrow}t_0 = 12 + 40 + 7 = 59$ and ${}^{\downarrow}t_1 = 17 + 1 + 7 = 25$. Hence, assuming an initial clock offset of 7 ms and no clock drift, the observation and occurrence times are indeed consistent.

Therefore, between 25 ms and 59 ms the observations made by the diagnoser contain only an observation of `trigger_1` but none of `trigger_0`, i.e., for any $25\,\text{ms} < t < 59\,\text{ms}$, the observation sequence is just:

$$\omega = \underbrace{\langle 25\,\text{ms}, \texttt{trigger\_1} \rangle}_{\theta_1'}$$

However, no timed event sequence induced by a run is consistent with this observation sequence because any run inducing a `trigger_1` event must also induce a `trigger_0` event occurring before (see the timed automaton in Example 5.1).

Necessary Observations. To tackle out-of-order observations, we define an observation model according to which all events which *must* have been observed will have been observed. A timed event $\langle i, \langle {}^{\uparrow}t, a \rangle \rangle$ must have been observed iff more than the maximal latency $\mathsf{l}_{\max}(a)$ of the observable action has passed since the event occurred, i.e., iff $\mathsf{Dur}(\rho) > {}^{\uparrow}t + \mathsf{l}_{\max}(a)$. Let $\mathsf{MustObs}(\rho)$ be the set containing all events of a run ρ that must have been observed:

$$\mathsf{MustObs}(\rho) \coloneqq \left\{ \langle i, \langle {}^{\uparrow}t, a \rangle \rangle \in \varrho(\rho) \mid \mathsf{Dur}(\rho) > {}^{\uparrow}t + \mathsf{l}_{\max}(a) \right\} \tag{5.11}$$

Using (5.11), we further define a set $\Pi(\rho)$ that contains all sets of events such that the respective sets contain all events that must have been observed and may or may not contain all other events that may or may not have been observed:

$$\Pi(\rho) \coloneqq \{ \varsigma \subseteq \varrho(\rho) \mid \mathsf{MustObs}(\rho) \subseteq \varsigma \} \tag{5.12}$$

Each set $\varsigma \in \Pi(\rho)$ corresponds to a timed event sequence $\mathsf{Word}(\varsigma)$. With that, we define a second candidate for an observation model, requiring that there exists some set $\varsigma \in \Pi(\rho)$, which by definition contains all the observations that must have been made, such that the timed event sequence $\mathsf{Word}(\varsigma)$ corresponding to this set is consistent with the observations that have been made:

$$\Omega(\rho) \coloneqq \left\{ \omega \in (\mathbb{Q} \times \mathsf{OAct})^{\star} \mid \exists \varsigma \in \Pi(\rho) : \mathsf{Word}(\varsigma) \rhd \omega \right\} \tag{5.13}$$

Now, (5.13) fixes the problems of the naive model (5.9). In particular, it allows for incomplete prefixes by requiring events that must have been observed to actually be observed while not mandating the remaining events to be observed.

Unfortunately, (5.13) is unrealistic as it does not guarantee that $\omega \in \Omega(\rho)$ can be continued to be consistent with the run ρ. It should be possible to extend an observation sequences such that it becomes consistent with the run as the observations missing due to latency will eventually be made. Adding this condition, we obtain our final observation model.

Definition 5.2.4 *We define the* continuous-time observation model *by:*

$$\Omega(\rho) \coloneqq \left\{ \omega \in (\mathbb{Q} \times \mathrm{OAct})^\star \mid \exists \varsigma \in \Pi(\rho), \omega' \geq \omega : \mathrm{Word}(\varsigma) \rhd \omega \wedge \varrho(\rho) \rhd \omega' \right\}$$

According to Definition 5.2.4, a run may induce an observation sequence which (a) accounts for all events that must have been observed and optionally events that may or may not have been observed, and (b) can be continued such that it becomes consistent with the run.

Terminated Runs. Consider that a run ρ does not progress any further because the system terminated. Realistically, time cannot actually halt, so we must assume that it keeps progressing even though the run does not continue. Thus, delayed observations will still eventually arrive (after their minimal latency) at the verdictor regardless of the fact that the run does not continue. In particular, an observation sequence $\omega \in \omega(\rho)$ induced by the run ρ as per Definition 5.2.4 may include observations of events $\langle i, \langle {}^{\uparrow}t, a \rangle \rangle$ with ${}^{\uparrow}t + \mathsf{l}_{\mathsf{min}}(a) > \mathrm{Dur}(\rho)$. That is, for these events, the minimal latency has not passed *relative to the duration of the run*, yet they may have been observed because the time kept progressing after the system terminated. If those observation sequences are observed at runtime, then they may allow a verdictor to detect termination.

Observation Prefixes. We will now establish a relation between consistency as per Definition 5.2.3 and the observation model. This relation will play a pivotal role for the correctness of the verdictor algorithm.

Lemma 5.2.3 *Assume given an observation sequence ω' consistent with some run ρ', i.e., $\varrho(\rho') \rhd \omega'$. For all prefixes $\omega \in \mathrm{Pref}(\omega')$ of the observation sequence ω' there exists a prefix $\rho \in \mathrm{Pref}(\rho')$ of the run ρ' such that $\omega \in \Omega(\rho)$ where Ω is the observation model as per Definition 5.2.4.*

For a detailed proof see Appendix A.2.3.

5.3 Verdictor Building Blocks

The verdictor algorithm for continuous-time systems will be based on two central building blocks, *active prefixes* and *bound consistency*, that we are going to introduce in the following. Together these render the problem algorithmically tractable.

The correctness proof of the verdictor algorithm will at its core also rely on theorems that we prove for active prefixes and bound consistency. The algorithm we

present here is Δ-complete (recall Definition 3.4.5) for

$$\Delta \coloneqq (1+\delta)\cdot(\max_{a\in\mathrm{OAct}} \mathsf{l}_{\max}(a) - \min_{a\in\mathrm{OAct}} \mathsf{l}_{\min}(a)) \tag{5.14}$$

with respect to the following distance function that returns the time difference of two non-empty timed observation sequences and 0 if either sequence is empty:

$$d(\epsilon,\cdot) \coloneqq 0 \qquad d(\cdot,\epsilon) \coloneqq 0 \qquad d(\omega \diamond \langle {}^{\downarrow}t, a\rangle, \omega \diamond \langle {}^{\downarrow}t', a'\rangle) \coloneqq {}^{\downarrow}t' - {}^{\downarrow}t \tag{5.15}$$

The choice of Δ is such that if more than Δ time passed after a timed observation θ, then any observations being made will be caused by events that occurred after the event causing the observation θ. This property is exploited by the subsequent definitions and the verdictor algorithm.

By delaying the verdict by an offset of Δ, we can avoid speculating about missing observations. Recall that in the discrete-time setting, we accounted for out-of-order observations by speculation (see Section 4.3.4). In the continuous-time setting, speculating about concrete possible future observations is not feasible, as there are infinitely many possible observation times.

5.3.1 Active Prefix Verdictor

The verdictor algorithm will be based on the concept of *active prefixes*. An active prefix is a certain subset of the observations that have been made. These subsets are chosen such that they contain all observations for which it is guaranteed that observations arriving in the future are caused by events that occurred after the events causing the observations. We call observations with that property *settled*. At any given time t, observations made at time t or later cannot be caused by events that occurred arbitrarily in the past because the latency of observations is bounded. Intuitively, it must be possible to account for all settled observations without speculating about future observations, as any future observations are guaranteed to be caused by events that happened later. Observations become settled if more than Δ time as per (5.14) has passed after they have been made:

$$\mathrm{Settled}(\omega) \coloneqq \{\langle j, \langle {}^{\downarrow}t, a\rangle\rangle \in \omega \mid {}^{\downarrow}t + \Delta < \mathrm{Dur}(\omega)\} \tag{5.16}$$

Here, $\mathrm{Dur}(\omega)$ is the time of the last observation in ω.

Active Subset Verdictor. A verdictor should account for at least all settled observations because observations made in the future must correspond to events that have occurred or will occur after the events causing the settled observations. We call those subsets of an observation sequence ω that contain at least all settled observations *active subsets*. The set of active subsets is defined as follows:

$$A(\omega) \coloneqq \{\varsigma \subseteq \omega \mid \mathrm{Settled}(\omega) \subseteq \varsigma\} \tag{5.17}$$

We prove that a VTS based on active subsets is sound and Δ-complete for Δ as per (5.14). To this end, let $\mathfrak{V}$ be a VTS maintaining the active subsets and then producing a verdict by considering all runs that are consistent with some active subset:

$$\nu(\omega) = \bigsqcup \{ \mathsf{V}(\rho) \mid \exists \varsigma \in A(\omega) : \varrho(\rho) \rhd \mathrm{Word}(\varsigma) \} \tag{5.18}$$

Here, V is the verdict oracle defined in the introduction of this chapter based on the verdict annotations of the edges of the timed automaton as per (5.1).

Considering active subsets avoids the need for speculation, because we simply look at the active subsets instead of any possible continuations as would be required by the observation model defined in Definition 5.2.4. We establish soundness and Δ-completeness for every VTS satisfying (5.18).

Theorem 5.3.1 *Every VTS $\mathfrak{V}$ satisfying (5.18) is sound.*

Proof Sketch. According to Definition 3.4.3, we must prove that $\mathsf{V}(\rho') \sqsubseteq \nu(\omega')$ for all $\omega' \in \mathcal{L}\restriction_\Omega([\![\mathfrak{T}]\!])$ and $\rho' \in \mathrm{Runs}(\omega')$. The proof rests on the fact that there exists a prefix $\rho \in \mathrm{Pref}(\rho')$ and an active subset $\varsigma \in A(\omega)$ such that ρ is consistent with $\mathrm{Word}(\varsigma)$, i.e., $\rho \rhd \mathrm{Word}(\varsigma)$. As this prefix exists, we have $\mathsf{V}(\rho) \sqsubseteq \nu(\omega)$. Further, due to monotonicity of V (see Proposition 5.0.1), we have $\mathsf{V}(\rho') \sqsubseteq \mathsf{V}(\rho)$. Therefore, $\mathsf{V}(\rho') \sqsubseteq \nu(\omega)$. For a detailed proof, including the existence of the prefix, see Appendix A.2.4. □

Theorem 5.3.2 *Every VTS $\mathfrak{V}$ satisfying (5.18) is Δ-complete for Δ as per (5.14).*

Proof Sketch. According to Definition 3.4.5, we must prove that $\nu(\omega') \sqsubseteq \mathsf{V}(\omega)$ for all $\omega \in \mathcal{L}\restriction_\Omega([\![\mathfrak{T}]\!])$ and $\omega' \in \mathrm{Cont}(\omega)$ such that $d(\omega', \omega) > \Delta$. The proof rests on the fact that after Δ time passed, all observations in ω become settled, i.e., $\omega \subseteq \mathrm{Settled}(\omega')$. The crux of the proof then lies in Lemma 5.2.3, which implies that there exists a prefix $\rho \in \mathrm{Pref}(\rho'')$ such that $\omega \in \Omega(\rho)$ for all runs ρ'' consistent with $\mathrm{Word}(\varsigma)$ where ς is some active subset of ω', i.e., $\varsigma \in A(\omega')$ and $\rho'' \rhd \mathrm{Word}(\varsigma)$. Analogously to the proof of Theorem 5.3.1, the proof exploits the fact that such a prefix exists and that V is monotonic (see Proposition 5.0.1). For a detailed proof, see Appendix A.2.6. □

Active Prefix Verdictor. While a VTS satisfying (5.18) is already sound and Δ-complete, we can refine the set $A(\omega)$, for the sake of further reducing the number of sets to be considered by the verdictor. An *active prefix* of a timed observation sequence ω is an active subset such that if it contains an observation θ, then it also contains all observations that are guaranteed to be caused by events that occurred before the event causing the observation θ. Formally, an active prefix is an active subset that is downward-closed with respect to $\prec$. The set of downward-closed subsets of ω with respect to $\prec$ is defined as follows:

$$\mathrm{Pref}_\prec(\omega) \coloneqq \{ \varsigma \subseteq \omega \mid \forall \theta \in \varsigma : \forall \theta' \in \omega : \theta' \prec \theta \Longrightarrow \theta' \in \varsigma \} \tag{5.19}$$

In the following, we refer to members of the set $\mathrm{Pref}_{\prec}(\omega)$ as $\prec$-*prefixes* of ω. The set of active prefixes is then given by:

$$\mathrm{Pref}_A(\omega) := \mathrm{Pref}_{\prec}(\omega) \cap A(\omega) \tag{5.20}$$

If $\theta \prec \theta'$ then a verdictor taking θ' into account should also take θ into account because the event causing θ must have occurred before the event causing θ' (cf. the discussion around (5.8)). A $\prec$-prefix has precisely the property that if it contains an observation θ', then it also contains all observations θ caused by events that must have occurred before the event causing θ'. This idea is formally captured by (5.20). Analogously to (5.18), let $\mathfrak{V}$ be a VTS such that:

$$\nu(\omega) = \bigsqcup \{ \mathsf{V}(\rho) \mid \exists \varsigma \in \mathrm{Pref}_A(\omega) : \varrho(\rho) \rhd \mathrm{Word}(\varsigma) \} \tag{5.21}$$

A VTS satisfying (5.21) is sound and Δ-complete.

Theorem 5.3.3 *Every VTS $\mathfrak{V}$ satisfying (5.21) is sound.*

For a detailed proof see Appendix A.2.5.

Corollary 5.3.1 *Every VTS $\mathfrak{V}$ satisfying (5.21) is Δ-complete for Δ as per (5.14).*

Proof. As $\mathrm{Pref}_A(\omega) \subseteq A(\omega)$, we have:

$$\left(\bigsqcup \{ \mathsf{V}(\rho) \mid \exists \varsigma \in \mathrm{Pref}_A(\omega) : \varrho(\rho) \rhd \mathrm{Word}(\varsigma) \}\right) \sqsubseteq \left(\bigsqcup \{ \mathsf{V}(\rho) \mid \exists \varsigma \in A(\omega) : \varrho(\rho) \rhd \mathrm{Word}(\varsigma) \}\right)$$

Thus, Theorem 5.3.2 directly implies Corollary 5.3.1. □

The verdictor algorithm will be based on maintaining the set of active prefixes and then producing a verdict as per (5.21). For this, we need a way to decide $\rho \rhd \mathrm{Word}(\varsigma)$ for a given active prefix ς. To this end, we introduce *bound consistency*.

5.3.2 Bound Consistency

In addition to active prefixes, the second concept necessary for algorithmic tractability is called *bound consistency*. Bound consistency is equivalent to consistency, but does not refer to an unknown time-remapping function, of which there are uncountably many. Instead, it requires that events and observations can be matched such that the time difference between the occurrence times of any two events is bounded by the difference bound as per (5.7) between the observations that caused them. Formally, we define bound consistency as follows:

Definition 5.3.1 *A timed event sequence* $\varrho = \left(\langle {}^{\uparrow}t_i, a_i\rangle\right)_{i=1}^{n}$ *and a timed observation sequence* $\omega = \left(\langle {}^{\downarrow}t_j, b_j\rangle\right)_{j=1}^{n}$ *are* bound-consistent *iff there exists a bijection* $\mathcal{R} : \varrho \to \omega$ *mapping events to observations such that*

(i) $a = b$ *for all* $\langle \cdot, \langle {}^{\uparrow}t, a\rangle\rangle \in \varrho$ *where* $\langle \cdot, \langle {}^{\downarrow}t, b\rangle\rangle = \mathcal{R}(\langle \cdot, \langle {}^{\uparrow}t, a\rangle\rangle)$*, and*

(ii) ${}^{\uparrow}t - {}^{\uparrow}t' \leq \mathsf{MaxD}(\mathcal{R}(e), \mathcal{R}(e'))$ *for all* $e, e' \in \varrho$*.*

Bound consistency and consistency are equivalent:

Theorem 5.3.4 *A timed event sequence* ϱ *and a timed observation sequence* ω *are consistent iff they are bound-consistent.*

Proof Sketch. To prove Theorem 5.3.4, one chooses the same bijection $\mathcal{R}$ for both consistency and bound consistency. It is then rather easy to see that if ϱ is consistent with ω, then it is also bound-consistent with ω because MaxD is an upper bound on the difference of the occurrence times. The proof of the other direction, however, is quite involved and rests on the construction of a piecewise-linear time-remapping function from the fact that the differences of the occurrence times between any two events is bounded by MaxD. For a detailed proof, we refer to Appendix A.2.7. □

5.4 Verdictor Algorithm

Having established its basic building blocks, we now present the verdictor algorithm for the continuous-time setting with timing imprecisions. The algorithm is based on active prefixes and produces a verdict as per (5.21), which we have already shown to be sound and Δ-complete for Δ as per (5.14) (see Theorem 5.3.3 and Corollary 5.3.1). To reason and track the verdicts of the individual runs that are consistent with the active prefixes, the algorithm relies on bound consistency and exploits abstraction techniques from timed automata reachability analysis, in particular the zone abstraction to represent sets of system states [AD91; BY03].

Conceptually, the algorithm implements an infinite state and deterministic VTS. The states of this VTS are pairs $\langle \omega, \mathbb{V}\rangle$ where ω is the sequence of observations made so far and $\mathbb{V}$ is a set of *verdict states* representing the states the system may be in, the observations the system has generated to get there, and the verdicts collected to get there. Formally, a verdict state is a triple $\langle s^{\#}, \varsigma, \upsilon\rangle$ where $s^{\#}$ is an *abstract system state*, $\varsigma \subseteq \omega$ is a set of *matched observations*, and υ is a verdict. An abstract system state $s^{\#}$ is a pair $\langle l, \eta^{\#}\rangle$ where l is a location of the timed automaton and $\eta^{\#}$ is a set of clock constraints. An abstract system state represents a set of system states:

$$[\![\langle l, \eta^{\#}\rangle]\!] \;\coloneqq\; \{\langle l, \eta \restriction_{\mathbb{C}}\rangle \mid \eta \vDash \eta^{\#}\}$$

In response to new observations, the algorithm updates the set of verdict states such that each verdict state $\langle s^\#, \varsigma, \upsilon \rangle \in \mathbb{V}$ abstractly represents a set of runs that generate the observations ς, that end up in some state represented by $s^\#$, and that are assigned the verdict υ by the verdict oracle. Thereby the algorithm enables computing the verdict as per (5.21) by joining the verdicts of all verdict states whose set of matched observations is an active prefix and whose set of states is non-empty:

$$\nu(\langle \omega, \mathbb{V} \rangle) \;\coloneqq\; \bigsqcup \{\, \upsilon \mid \exists \langle s^\#, \varsigma, \upsilon \rangle \in \mathbb{V} : \varsigma \in \mathrm{Pref}_A(\omega) \wedge [\![s^\#]\!] \neq \varnothing \,\} \tag{5.22}$$

In the following, we will state the exact invariants required for correctness formally, present the core algorithm for updating verdict states, and describe how the invariants are initially established and maintained by using this algorithm.

Correctness Invariants. To inherit soundness and Δ-completeness from the verdict defined via the active prefixes, we must ensure that the verdict as per (5.22) is identical to the verdict as per (5.21). To this end, the following correctness invariants are required for the states $\langle \omega, \mathbb{V} \rangle$:

(IV1) For each verdict state $\langle s^\#, \varsigma, \upsilon \rangle \in \mathbb{V}$, ς is a downward closed subset of ω with respect to $\prec$, i.e., ς is a $\prec$-prefix as per (5.19).

(IV2) For all $\prec$-prefixes $\varsigma \in \mathrm{Pref}_\prec(\omega)$ and verdicts $\upsilon \in \mathcal{V}$, it holds that:

$$\begin{aligned} & \bigcup \{\, [\![s^\#]\!] \mid \langle s^\#, \varsigma, \upsilon \rangle \in \mathbb{V} \,\} \\ = \; & \bigcup \{\, \mathrm{After}(\rho) \mid \rho \in \mathrm{Runs}([\![\mathfrak{T}]\!]) \text{ s.t. } \mathsf{V}(\rho) = \upsilon \wedge \varrho(\rho) \rhd \mathrm{Word}(\varsigma) \,\} \end{aligned}$$

In particular invariant (IV2) ensures that the verdict states $\mathbb{V}$ cover all the states the system may be in after generating the respective observations. Based on these invariants, it is easy to see that the verdict as per (5.22) is indeed the verdict produced by an active prefix verdictor as per (5.21).

Clock Zones. To *canonically* and efficiently represent verdict states, the verdictor algorithm relies on *clock zones*. Clock zones are typically used for reachability analysis of timed automata [BY03; AD91; BLR05]. Mathematically, a clock zone is a set of clock valuations that can be represented by a set $\eta^\#$ of clock constraints. Given a set of clock constraints $\eta^\#$, we will denote the zone it represents by $[\![\eta^\#]\!]$. Formally:

$$[\![\eta^\#]\!] \;\coloneqq\; \{\, \eta \in \mathbb{C} \to \mathbb{R}_0^+ \mid \eta \vDash \eta^\# \,\}$$

It is well-known that clock zones can be canonically represented by *Difference Bound Matrices* (DBMs), a matrix data structure that stores a *bound* for each pair $\langle x, y \rangle$ with $x, y \in \mathbb{C} \cup \{0\}$. A bound constrains the difference $x - y$ and can be either ∞ or $\sim c$ where $\sim \in \{<, \leq\}$ and $c \in \mathbb{Q}$ [BY03]. A DBM corresponds to a canonical

set of clock constraints $x - y \sim c$ for each pair $\langle x, y \rangle$ and bound of the form $\sim c$. Notably, any set of clock constraints can be represented canonically like that. For the verdictor algorithm, we assume that sets of clock constraints are always represented canonically, e.g., in the form of a DBM. In addition to canonicity, DBMs also enable efficient operations on clock zones. The verdictor algorithm will make use of standard operations[15] on clock zones, namely (i) $\texttt{Reset}(\eta^\#, \mathsf{R})$ which resets the clocks R in $\eta^\#$ to zero and (ii) $\texttt{Future}(\eta^\#)$, which drops the upper bounds on all clock variables but retains the clock differences. Formally, they have the following semantics:

$$[\![\texttt{Reset}(\eta^\#, \mathsf{R})]\!] = \{\, \eta \downarrow \mathsf{R} \mid \eta \in [\![\eta^\#]\!] \,\}$$

$$[\![\texttt{Future}(\eta^\#)]\!] = \left\{\, \eta \oplus \Delta t \mid \eta \in [\![\eta^\#]\!], \Delta t \in \mathbb{R}_0^+ \,\right\}$$

For the implementation of these operations using DBMs, we refer to Bengtsson and Yi [BY03]. To combine multiple sets of clock constraints, we use $\cup$ and implicitly assume that the result is made canonical.

Updating Verdict States. To maintain the invariants, the verdictor algorithm needs to update the set of verdict states when new observations arrive. At the heart of the algorithm is an exploration procedure EXPLORE taking a verdict state $\mathsf{v} = \langle s^\#, \varsigma, \upsilon \rangle$ and an observation $\theta \in \omega \setminus \varsigma$. It then extends v with θ to produce successor verdict states $\langle s^{\#\prime}, \varsigma \cup \{\theta\}, \upsilon' \rangle$. This procedure is displayed in Algorithm 2 . Algorithm 2 is a variation of the reachability analysis algorithm for zone graphs of timed automata [BY03; AD91; BLR05] extended by adding clocks for observations and constraining their difference according to Definition 5.3.1 (see line 11 and 12), and combining the verdict annotations as per (5.1) and Definition 4.1.2 (line 16).

Invariant (IV2) implies that a verdict state $\langle s^\#, \varsigma, \upsilon \rangle \in \mathbb{V}$ represents a set of states $[\![s^\#]\!]$ of the LTS semantics of the system model and that those states are reached after runs consistent with ς and which induce the verdict υ. Now, the exploration procedure computes all the successor verdict states $\langle s^{\#\prime}, \varsigma \cup \{\theta\}, \upsilon' \rangle$ such that the states $[\![s^{\#\prime}]\!]$ are reachable from some state in $[\![s^\#]\!]$ after runs consistent with $\varsigma \cup \{\theta\}$ and which induce the verdict υ'.

In Algorithm 2, the input observation θ_x is optional. If not present, it is denoted by $\perp$. The latter case occurs as a result of a recursive step, where the observation provided initially has already been matched with an event (in line 13, θ'_x is set to $\perp$). This is handled (line 3-4) by adding the verdict state to the set S to be returned as result (line 17). Regardless of the provision of an observation, the algorithm explores possible successor states by iterating over those edges of the system model whose action is either unobservable or may be matched with the observation if provided (line 5). Here, $\texttt{Action}(\theta_x)$ denotes the action associated with θ_x if provided, and

[15] Note that `Future` is sometimes called UP in the literature.

Algorithm 2: Recursive procedure for exploring verdict states (`Explore`).

Data: a verdict state $\langle s^\#, \varsigma, \upsilon \rangle$ and an optional observation $\theta_x = \langle j, \langle {}^{\downarrow}t, a \rangle \rangle$
Result: a set of verdict states

```
1  function Explore(⟨⟨l, η#⟩, ς, υ⟩, θx)
2      S := ∅
3      if θx = ⊥ then
4          S := {⟨⟨l, η#⟩, ς, υ⟩}
5      for ⟨l, G, α, R, l'⟩ ∈ E where α ∉ OAct ∨ α = Action(θx) do
6          η#s' := η# ∪ G
7          if [[η#s']] ≠ ∅ then
8              ς', θ'x := ς, θx
9              if α = Action(θx) then
10                 η#s' := Reset(η#s', {θx})
11                 for θy ∈ ς do
12                     η#s' := η#s' ∪ {θx − θy ≤ MaxD(θy, θx), θy − θx ≤ MaxD(θx, θy)}
13                 ς', θ'x := ς ∪ {θx}, ⊥
14             η#s' := Future(Reset(η#s', R)) ∪ Inv(l')
15             if [[η#s']] ≠ ∅ then
16                 υ' := υ ⊓ T(l, α, l')
17                 S := S ∪ Explore(⟨⟨l', η#s'⟩, ς', υ'⟩, θ'x)
18     return S
```

$\bot \notin$ Act otherwise. For each edge, the following is executed: First, the zone of the verdict state is constrained by the guard to check whether the edge is enabled (line 6-7). If so, and if the action agrees with the observation (line 9), a clock for the observation is introduced and initialized to zero (line 10). Subsequently, the definition of bound consistency (recall Definition 5.3.1) is echoed by constraining the difference between clock θ_x and the clocks recording the time passed since already matched observations (line 11-12). The observation has thus been matched and is added to the set of matched observations. For subsequent recursive calls, the observation is then set to $\bot$ (line 13). The next step computes the successor zone (line 14) in the usual manner, resetting the clocks as specified by the edge, then applying the `Future` operator, and finally constraining the obtained zone by the invariant of the successor location. If the successor zone is non-empty (line 15), then the set S is extended by calling the exploration procedure recursively using the obtained successor location, zone, matched observations, verdict, and observation (line 17). To this end, the verdict has been updated with the annotation of the taken edge (line 16). The procedure terminates because we assumed that there can be no cycles free of observables (A5) and Algorithm 2 allows only one observable corresponding to θ_x to occur.

Establishing the Invariants. The set of verdict states $\mathbb{V}$ is initialized with

$$\mathbb{V}_0 \coloneqq \bigcup \left\{ \texttt{Explore}\left(\left\langle \langle l, \eta^{\#}_{0,l} \rangle, \varnothing, \mathsf{T} \right\rangle, \bot\right) \mid l \in I \right\} \tag{5.23}$$

where $\eta^{\#}_{0,l}$ is the initial clock zone obtained by initializing all clocks to zero, then applying `Future`, and finally constraining the result with the zone $\mathsf{Inv}(l)$. Initially, there are no observations, i.e., $\omega = \epsilon$, hence, $\varnothing$ is its only $\prec$-prefix. The set $\mathbb{V}_0$ obtained by the initial invocation of `Explore` contains only states with an empty set of matched observations, thereby satisfying invariant (IV1). Furthermore, invariant (IV2) is also satisfied because `Explore` explores all runs consistent with the empty set of observations and records the induced verdicts respectively, i.e., for each $\upsilon \in \mathcal{V}$:

$$\bigcup \{ [\![s^{\#}]\!] \mid \langle s^{\#}, \varsigma, \upsilon \rangle \in \mathbb{V}_0 \} = \bigcup \{ \mathsf{After}(\rho) \mid \rho \in [\![\mathfrak{T}]\!] \text{ s.t. } \mathsf{V}(\rho) = \upsilon \wedge \varrho(\rho) \rhd \epsilon \}$$

Therefore, initially both invariants are satisfied.

Maintaining the Invariants. Recall that the invariant (IV1) requires that the set of matched observations of any verdict state is a $\prec$-prefix of ω. To maintain this invariant, the exploration procedure must only ever be used to extend a verdict state with an observation such that $\varsigma \cup \{\theta\}$ is again a $\prec$-prefix of ω. To this end, we define the *frontier* $\mathsf{Frontier}_{\omega}(\varsigma) \subseteq \omega \setminus \varsigma$ of a $\prec$-prefix ς of ω to be the set of all observations $\theta \in \omega \setminus \varsigma$ such that $\varsigma \cup \{\theta\}$, coined the *$\theta$-extension* of ς, is also a $\prec$-prefix of ω. For each of the observations in the frontier of a given state's matched observations, we maintain a flag indicating whether we already extended the state with the respective observation. After making a new observation, we first remove those verdict states $\langle \langle l, \eta^{\#} \rangle, \varsigma, \upsilon \rangle \in \mathbb{V}$ from $\mathbb{V}$ that are no longer $\prec$-prefixes of the new ω. This restores the invariant (IV1), i.e., that for each $\langle s^{\#}, \varsigma, \upsilon \rangle \in \mathbb{V}$, ς is a $\prec$-prefix of ω.

Subsequently, to maintain both invariants, the exploration procedure is called iteratively to update the set $\mathbb{V}$. After making a new observation and while there is a verdict state $\mathsf{v} = \langle \langle l, \eta^{\#} \rangle, \varsigma, \upsilon \rangle \in \mathbb{V}$ where the frontier of ς contains observations with which v has not already been extended (determined using the flag), we proceed as follows: For each such observation $\theta \in \mathsf{Frontier}_{\omega}(\varsigma)$ in the frontier of ς, extend $\mathbb{V}$ with $\texttt{Explore}(\mathsf{v}, \theta)$ and flag θ for v indicating that v has already been extended with θ. By only ever extending verdict states with observations in the frontier, invariant (IV1) is clearly maintained by this approach. Invariant (IV2) is also maintained because EXPLORE is exhaustively used to explore all possible runs to obtain successor states and the verdicts they induce.

Example 5.7 Let us come back to the running example (Example 5.1). We assumed that the verdict domain is the simplified diagnosis domain (Definition 3.1.3) over a single fault class f_B indicating a faulty bearing. With a clock accuracy of ±3% and a minimal and maximal latency between 1 ms and 3 ms, respectively, as defined in the previous examples, the verdict offset Δ is:

$$\Delta = (1 + 0.03) \cdot (3\,\text{ms} - 1\,\text{ms}) = 2.06\,\text{ms} \tag{5.24}$$

For the timed automaton as introduced in Example 5.1, the initial verdict states are:

$$\underbrace{\langle\langle l_{N1}, \{0 \leq x \leq 200\}\rangle, \emptyset, \emptyset\rangle}_{\text{(a)}} \qquad \underbrace{\langle\langle l_{F1}, \{0 \leq x \leq 200\}\rangle, \emptyset, \{f_B\}\rangle}_{\text{(b)}} \tag{5.25}$$

Assuming that no observable events occurred yet, the system is either in location l_{N1} (a) or location l_{F1} (b). In both cases, the value of the clock x is somewhere in the interval $[0, 200]$ ms. To get to location l_{F1}, a bearing fault `fault_bearing` must have occurred which is tracked by the verdict state (b). Now, for the sake of the example, assume that an observation $\theta_0 = \langle 0, \langle 212\,\text{ms}, \texttt{trigger\_0}\rangle\rangle$ as defined in Example 5.3 is made. Applying Algorithm 2, the states (5.25) and the observation θ_0 are used to compute successor verdict states:

$$\begin{gathered}\underbrace{\langle\langle l_{N2}, \{0 \leq x \leq 6, \theta_0 = x\}\rangle, \{\theta_0\}, \emptyset\rangle}_{\text{(a')}} \\ \underbrace{\langle\langle l_{F2}, \{0 \leq x \leq 12, \theta_0 = x\}\rangle, \{\theta_0\}, \{f_B\}\rangle}_{\text{(b')}}\end{gathered} \tag{5.26}$$

The state (a′) is a successor of (a) and the state (b′) is a successor of (b) reached after the observable `trigger_0` action observed as θ_0, respectively. These states capture that, after the occurrence of `trigger_0`, the system is either in location l_{N2} (a′) or l_{F2} (b′). The `Reset` operation in line 10 of Algorithm 2 resets the clock, denoted by θ_0, which has been introduced for observation θ_0. At the same time, the `Reset` operation in line 14 resets the clock x as modeled by the timed automaton (cf. Example 5.1). As a result, both clocks start at zero when the `Future` operation is applied in line 14 and have the same value. The fact that both clocks continue to have the same value is captured by the constraint $0 \leq \theta_0 - x \leq 0$ which we abbreviate as $\theta_0 = x$ (highlighted in green in (5.26)). The clock for θ_0 tracks the time since the corresponding event occurred and this time coincides with the resetting of x.

Recall that all verdict states where ς is an active prefix of the set of observations made until time t need to be considered when computing the verdict (cf. (5.21)). At time $t = 212$ ms, when the observation θ_0 is made, the empty set $\emptyset$ is still an active prefix because θ_0 is not settled yet. Hence, after observing θ_0, the states in (5.25) and (5.26) have to be considered for computing the verdict as per (5.22). The resulting verdict is the empty set, no fault certainly occurred.

Clearly, the observation of just θ_0 alone is not indicative for a fault. To diagnose something, we need more observations. To this end, imagine that after θ_0 the observation $\theta_1' = \langle 1, \langle 224\,\text{ms}, \texttt{trigger\_1}\rangle\rangle$ as considered in Example 5.1 is made. Both observations together are indicative for a fault of the bearings because the large time difference between them can only be explained with increased friction. The

difference bounds between these observations are:

$$\mathrm{MaxD}(\theta_0, \theta_1') = 3\,\mathrm{ms} - 1\,\mathrm{ms} + 1.03^{-1} \cdot (212\,\mathrm{ms} - 224\,\mathrm{ms}) \approx -9.65\,\mathrm{ms} \tag{5.27}$$

$$\mathrm{MaxD}(\theta_1', \theta_0) = 3\,\mathrm{ms} - 1\,\mathrm{ms} + 1.03 \cdot (224\,\mathrm{ms} - 212\,\mathrm{ms}) = 14.36\,\mathrm{ms} \tag{5.28}$$

To account for the new observation θ_1', the diagnosis algorithm proceeds as follows:

- Starting with state (a′), the `trigger_1` transition from l_{N2} to l_{N1} must be taken. To this end, first, line 6 of Algorithm 2 takes into account the guard of the transition leading to the following zone:

  $$\{3 \leq x \leq 6, \theta_0 = x\}$$

  This zone is not empty (checked in line 7). Hence, the clock for the new observation is introduced and subsequently reset to zero (line 10) giving rise to the following zone:

  $$\{3 \leq x \leq 6, \theta_0 = x, \theta_1' = 0\}$$

  In the next step (line 11-12), the constraints based on the bounds (5.27) and (5.28) are added (highlighted in green):

  $$\{3 \leq x \leq 6, \theta_0 = x, \theta_1' = 0, 9.65 \leq \theta_0 - \theta_1' \leq 14.36\} \tag{5.29}$$

  The resulting zone, however, is empty because the existing constraints imply $3 \leq \theta_0 - \theta_1' \leq 6$ while the constraints based on the bounds require $9.65 \leq \theta_0 - \theta_1'$. The application of the `Reset` and `Future` operator in line 14 does not change the fact that the zone is empty, and, hence, the exploration procedure does not recurse (due to the check in line 15). Therefore, with the observation θ_1', the state (a′) has no successors.

- Now, for state (b′) the outcome is different. Analogously to the state (a′), the `trigger_1` transition from l_{F2} to l_{F1} must be taken which gives rise to the following zone:

  $$\{6 \leq x \leq 12, \theta_0 = x, \theta_1' = 0\}$$

  Again, the constraints for the bounds are added:

  $$\{6 \leq x \leq 12, \theta_0 = x, \theta_1' = 0, 9.65 \leq \theta_0 - \theta_1' \leq 14.36\}$$

  This time, the resulting zone is not empty. The application of the `Reset` and `Future` operator in line 14 subsequently yield the following non-empty zone:

  $$\{0 \leq x \leq 200, 206 \leq \theta_0 \leq 212, \theta_1' = x, 9.65 \leq \theta_0 - \theta_1' \leq 14.36\}$$

  So, the following successor state of (b′) is obtained:

  $$\left\langle l_{F1}, \left\{ \begin{array}{c} 0 \leq x \leq 200, \\ 206 \leq \theta_0 \leq 212, \\ \theta_1' = x, \\ 9.65 \leq \theta_0 - \theta_1' \leq 14.36 \end{array} \right\}, \{\theta_0, \theta_1'\}, \{f_B\} \right\rangle \tag{5.30}$$

After an additional delay of $\Delta = 2.06\,\text{ms}$, at time $t = 226.06\,\text{ms}$, the observation θ_1' becomes settled and $\{\theta_0\}$ stops being an active prefix. As a result, when the next observation arrives, the verdictor now only considers the state in (5.30) which contains the fault class f_B. Therefore, the fault `fault_bearing` is diagnosed after time $t = 226.06\,\text{ms}$, i.e., 2.06 ms after the observation θ_1'.

Robustness Theorem. Combining the considerations above, we obtain the following theorem for the continuous time verdictor algorithm:

Theorem 5.4.1 *The verdictor algorithm for the continuous-time case described above produces sound and Δ-complete verdicts for*

$$\Delta = (1 + \delta) \cdot (\max_{a \in \text{OAct}} l_{\max}(a) - \min_{a \in \text{OAct}} l_{\min}(a)) \tag{5.14}$$

and with respect to the TS semantics $[\![\mathfrak{T}]\!]$ of the timed automaton system model $\mathfrak{T}$, the continuous time observation model as per Definition 5.2.4, and the verdict oracle as per (5.1) based on the verdict annotations of the timed automaton $\mathfrak{T}$.

Proof Sketch. The proof rests on the already established theorems Theorem 5.3.3 and Corollary 5.3.1, and the invariants of the algorithm. As argued above, the algorithm establishes the invariants initially and then maintains them. Furthermore, the invariants together ensure that the verdict produced for an observation sequence is indeed the verdict as per the definition of the active prefix verdictor (5.21). □

As the continuous time observational model includes timing imperfections, Theorem 5.4.1 establishes that the verdictor algorithm is indeed robust as characterized in Section 3.4.

Optimizations. For exposition purposes, the presented algorithm keeps verdict states indefinitely even if they are no longer relevant. All diagnosis states $\langle s^\#, \varsigma, \upsilon \rangle$ whose frontier $\text{Frontier}_\omega(\varsigma)$ is non-empty and contains only settled observations can instead be discarded. Certainly, for those states, ς is not an active prefix and we have already obtained all possible successor states because new observations cannot be added to the frontier. Analogously, we can discard parts of ω. As a result of these optimizations, the size of the set $\mathbb{V}$ and ω is reduced while retaining soundness and Δ-completeness. Furthermore, instead of implementing EXPLORE recursively, any practical implementation should deploy either breadth- or depth-first search in an iterative fashion and use memoization, again reducing cost.

Sentinel Pruning. Note that the algorithm might produce the sentinel bottom verdict, as it tracks the verdicts of the taken edges by computing their meet. Analogously to the discrete-time case, the sentinel verdict is either not produced, or, if it is,

the states that do produce it can be dismissed as unrealistic. Changing the algorithm to discard verdict states with the sentinel verdict and stop the exploration of their successors is straightforward.

5.4.1 Bounded History Approximation

A major challenge remaining to be tackled is reducing the computational cost of the verdictor algorithm. While the presented verdictor algorithm processes observations incrementally, it introduces a fresh clock (with constraints according to Definition 5.3.1) for each new observation (line 10-12 in Algorithm 2). As a result, the memory consumption and time for inserting a new observation grows with each new observation. This renders the algorithm practically infeasible.

The crux of practical feasibility lies in adapting Algorithm 2 to consider the difference bounds only for a bounded history of observations so that the algorithm is still able to provide verdicts with an acceptable offset Δ while bounding its space and time requirements by the size of the system model.

As an extreme case, let us first look at a variant of Algorithm 2 where no difference bounds are considered at all. We obtain this variant by simply dropping lines 10-12 of Algorithm 2 but keeping everything else the same. The resulting algorithm only considers the order of events according to $\prec$ but not their precise timing. As a result, more verdict states are considered possible and we obtain we obtain over-approximate verdicts. This variant of the algorithm is still sound but not generally Δ-complete. So, while this reduces the computational costs, it also means that verdicts are becoming less specific as they do not take timing information into account.

Instead of considering no difference bounds at all, we now adapt the algorithm to take the difference bounds into account for a history of at most $B_H \in \mathbb{N}_0$ observations. In the following, we call B_H a *history bound*. We separate maintaining the $\prec$-prefixes of ω and a *history* $H \subseteq \omega$ considered for each verdict state when considering difference bounds such that $|H| \leq B_H$. Technically, we augment verdict states with H by turning them into quadruples $\langle s^\#, \varsigma, \upsilon, H\rangle$ and we modify the lines 10-12 of Algorithm 2 accordingly to manage the history H and to reuse clocks of discarded observations. That is, at most B_H additional clocks are introduced. Instead of adding a clock for every observation (line 10), a clock may be reused by discarding an observation from the history and using its clock again. Furthermore, the difference bounds are now only applied for the observations in the history.

Concerning the choice which observations to discard and which to keep in the history, there is a realm of possibilities whose investigation we leave for future work. For now, we treat H as a circular buffer such that the least recently inserted observation gets discarded when a new observation is added to the history once the bound B_H is reached. As a result, we obtain a family of algorithms (one for each history bound B_H) constituting sound but not generally Δ-complete verdictors. Combined

with the earlier discussed optimizations, the space and time requirements for inserting new observation is then bounded by the size of the model and independent of the number of observations—a property paramount for practical feasibility. Recall that the time complexity of the reachability problem for timed automata is exponential in the number of clocks [AD91]. With a history bound of B_H, the algorithm requires at most B_H additional clocks for keeping track when observable events occur. These additional clocks thus cause the time complexity to grow polynomially (with a polynomial rank of B_H) in the size of the original model.

Example 5.8 Let us revisit Example 5.7 with a history bound. Choosing $B_H = 0$ would have the effect that the difference bound constraints in (5.29) (in green) would be missing from the constructed zone. As a result, the zone would not be empty and (a′) would have a successor diagnosis state (without any faults) thereby preventing the diagnosis of the fault. So, with a history bound of $B_H = 0$, the fault is no longer diagnosable demonstrating the potentially incomplete nature of diagnosis with a history bound. However, with a history bound of $B_H = 2$ the fault is still diagnosed without any additional delay because with $B_H = 2$ the difference bound constraints in (5.29) (in green) will still be added by the modified algorithm thereby leading to an empty zone and subsequent diagnosis of the fault.

Correctness Theorem. We obtain the following correctness theorem for the continuous time verdictor algorithm with a bounded history:

Theorem 5.4.2 *The verdictor algorithm with a bounded history as described above produces sound verdicts with respect to the TS semantics $[\![\mathfrak{T}]\!]$ of the timed automaton system model $\mathfrak{T}$, the continuous time observation model as per Definition 5.2.4, and the verdict oracle as per (5.1) based on the verdict annotations of the timed automaton $\mathfrak{T}$.*

Proof Sketch. The original algorithm without a bounded history produces sound verdicts as per Theorem 5.4.1 by joining the verdicts associated with the verdict states as per (5.22). As argued above, by not considering certain difference bounds, a superset $\mathbb{V}' \supseteq \mathbb{V}$ of the verdict states compared to the original algorithm are considered for each observation sequence. Hence, the verdict produced by joining the verdicts associated with $\mathbb{V}'$ is at most as specific as the known-sound verdict produced by joining the verdicts associated with $\mathbb{V}$. Therefore, the verdict produced by joining the verdicts associated with $\mathbb{V}'$ is also sound. □

5.4.2 Non-Monotonic Verdicts

So far, we considered only timed automata with edge annotations, resulting in a monotonic verdict oracle. In general, with a non-monotonic verdict oracle, a verdictor

cannot be Δ-complete as the definition of Δ-completeness requires that the verdict is more specific for all future continuations after at most Δ time has passed, i.e., verdicts can only get more specific. We will now discuss how the algorithm can be adapted to the non-monotonic case. To this end, in addition to the edge annotations λ_E, we consider two functions $\kappa_L : I \to \upsilon$ and $\lambda_L : L \to \upsilon$ assigning verdicts to locations analogously to κ and λ for the discrete-time case (recall Definition 4.1.1). Using these, we then define a verdict oracle for the transition system semantics $[\![\mathfrak{T}]\!]$ of the timed automaton $\mathfrak{T}$ by instantiating Definition 4.1.1 with:

$$\kappa(\langle l, \eta\rangle) \coloneqq \kappa_L(l) \qquad \lambda(\langle l, \eta\rangle) \coloneqq \lambda_L(l) \qquad \gamma(\langle\langle l, \eta\rangle, a, \langle l', \eta'\rangle\rangle) \coloneqq \lambda_E(l, a, l') \tag{5.31}$$

We then adapt the algorithm as follows. Instead of initializing the initial verdict states with the top verdict $\top$ as per (5.23), we initialize them with the verdict assigned to the respective initial location as per κ_L:

$$\mathbb{V}_0 \coloneqq \bigcup \left\{ \texttt{Explore}\left(\left\langle\left\langle l, \eta^{\#}_{0,l}\right\rangle, \varnothing, \kappa_L(l)\right\rangle, \bot\right) \mid l \in I \right\}$$

Furthermore, we adapt the computation of verdicts from verdict states as per (5.22) by also taking the location of the abstract state into account:

$$\nu(\langle \omega, \mathbb{V}\rangle) \;\coloneqq\; \bigsqcup \{ \upsilon \sqcap \lambda_L(l) \mid \exists \langle\langle l, \eta^{\#}\rangle, \varsigma, \upsilon\rangle \in \mathbb{V} : \varsigma \in \mathrm{Pref}_A(\omega) \wedge [\![\langle l, \eta^{\#}\rangle]\!] \neq \varnothing \}$$

Due to the resulting possibly non-monotonic verdicts produced by the verdictor algorithm, it is, in general, no longer sound. Soundness requires the verdicts to be at most as specific as the verdict of the oracle, which allows lagging behind iff the oracle is monotonic. The just defined oracle is no longer monotonic and hence, the verdictor must not lag behind to be sound. However, as the algorithm does not speculate about possible future observations, it can lag behind and is thus no longer sound. For similar reasons, it is also not complete.

While the algorithm adapted for the non-monotonic case is generally neither sound nor complete for the general case, the verdicts it produces still bear a relation to the system model. If the verdict oracle is monotonic for a sufficiently long segment of at least Δ time, then the verdict oracle becomes sound after Δ. While it still potentially lags Δ behind, after Δ time has passed in a monotonic segment, the algorithm starts producing a sound verdict. An analogous argument for completeness can be made.

5.5 Discussion

In this chapter, we introduced a verdictor algorithm for the continuous-time setting. The algorithm effectively handles the superimposition of variable latencies and clock drift as well as inherent observability limitations with respect to the actions that can be observed. It produces verdicts that are robust with respect to timing imprecisions.

For the monotonic case, we indeed proved that the algorithm produces sound and Δ-complete verdicts with respect to a given system model. Moreover, we discussed the extension of the algorithm to the non-monotonic settings.

As the cost of a precise analysis grows exponentially with the number of observations, we parametrized the algorithm with a history bound. By considering only a bounded history, the space and time requirements become bounded by the size of the model and thereby independent of the number of observations—a property paramount for practical feasibility. This advantage, however, comes at the expense of producing over-approximate verdicts that are still sound but, in general, no longer Δ-complete. We will discuss the impacts of history bounds later in Section 7.3.

We will explore further concrete use cases beyond diagnosis in Chapter 7 and Chapter 8 and evaluate the algorithm empirically in Section 7.3 on a case study based on the industrial automation example.

Part III

From Theory to Practice

Chapter 6

Formal Modeling Toolbox Momba

The model-based methodology adopted by this thesis requires the creation of verdict-annotated formal models that accurately describe the possible behaviors of a system. Creating formal models for complex systems and *validating* that they indeed faithfully describe the system can be a challenge. Yet, it is required to apply the techniques presented in this thesis and to obtain provable guarantees. To ease the process of model creation, validation, and analysis, the author of this thesis developed *Momba*, a flexible Python framework for dealing with formal models. The techniques presented in this thesis have been implemented by the author within Momba around its extensible core and using its APIs. This implementation will be the basis for empirical evaluations in the subsequent chapters. Crucially, Momba goes beyond verdictors—it aims to be a general platform for model-based design and analysis workflows. In this chapter, we present Momba and discuss its architecture, use cases, the interfaces it provides, and empirically evaluate parts of its implementation.

Motivation and Requirements. Dealing with formal models encompasses a variety of tasks which can be challenging from time to time—especially for newcomers. Everything starts with the *construction* of a model or a family thereof. Often a textual or other, more formal, description of the scenario to be modeled is already existing. For instance, a rough sketch of the desired behavior or a circuit diagram. Then, after a formal model has finally been conceived, one has to *validate* that the model actually adequately models what should be modeled. In this regard models are just like any other human artifact, inadequate initially but getting better over time. Only after confidence in the model has been established, one is able to harvest the benefits of formal modeling by handing the model over to *analysis* tools, e.g., a model checker,

or leveraging the techniques developed in this thesis. Momba strives to deliver an integrated and intuitive experience to aid the process of model construction, validation, and analysis thereby making formal methods more accessible. Starting with this motivation, we elicit the following high-level requirements:

(HLR1) *Ease of Use.* Momba should be easily approachable by newcomers and experts alike. This includes a simple installation process and proper documentation of the APIs and other functionality.

(HLR2) *Model Construction.* Momba should enable the programmatic construction of formal models from preexisting scenario descriptions in a modular fashion while catching modeling errors early.

(HLR3) *Model Validation.* Momba should be able to serve as a platform for model validation by simulation. For instance, by enabling rapid prototyping of interactive model exploration and visualization tools which can be used to gain confidence into the accuracy of the model.

(HLR4) *Model Analysis.* Momba should provide unified interfaces for model analysis tools and make them readily available.

(HLR5) *Ecosystem Compatibility.* Momba should be compatible with the existing ecosystem surrounding formal models. In particular, it should be able to read existing models, interface with existing state-of-the-art tools, and produce models for these tools.

In addition to these original requirements, the usage of Momba as part of this thesis and in other work [Gro+22] also lead to the following requirement:

(HLR6) *Performance.* The performance of Momba's simulation and state space exploration engine should be state-of-the-art.

As we show in this chapter, Momba meets all these requirements.

Relevant Publications. Momba has first been presented at TACAS:

> [KKH21]: Maximilian A. Köhl, Michaela Klauck, and Holger Hermanns. "Momba: JANI Meets Python". In: *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2021.*

After its initial inception, Momba has been used and significantly extended for the work on robust diagnosis by the author of this thesis (cf. Chapter 5):

> [KH23]: Maximilian A. Köhl and Holger Hermanns. "Model-Based Diagnosis of Real-Time Systems: Robustness Against Varying Latency, Clock Drift, and Out-of-Order Observations". In: *ACM Transactions on Embedded Computing Systems, TECS 2023.*

Furthermore, Momba has also been used and extended for the following work on training and verifying decision-making agents based on formal models:

> [Gro+22]: Timo P. Gros, Holger Hermanns, Jörg Hoffmann, Michaela Klauck, Maximilian A. Köhl, and Verena Wolf. "MoGym: Using Formal Models for Training and Verifying Decision-making Agents". In: *Computer Aided Verification, CAV 2022*.

Besides these works, Momba's simulation and state space exploration engine has been empirically evaluated and compared to other state-of-the-art tools as part of the *Quantitative Verification Competition* (QComp) [And+24]. The author of this thesis coordinated[16] the respective category of QComp 2023.

To foster adoption within the community, the author of this thesis also held a tutorial on Momba as part of FM'21. The tutorial walks participants through Momba's model construction, simulation, and analysis capabilities using a platform game as an example. This tutorial is available online[17] and presents a further learning resource towards making Momba more accessible and satisfying (HLR1).

Availability Statement. The full source code and documentation of Momba is available as part of artifact (AT5). This artifact also includes the source code of the implementations developed for this thesis. This chapter is primarily based on the Momba TACAS paper [KKH21] whose respective artifact is (AT3).

Chapter Structure. Section 6.1 discusses the high-level architecture and design decisions behind Momba against the backdrop of the requirements presented above. Section 6.2 takes on the perspective of a user and showcases how Momba can be used for the construction, validation, and analysis of quantitative models. Section 6.3 empirically evaluates Momba's simulation and state space exploration engine and compares it to other state-of-the-art tools. Section 6.4 concludes this chapter by discussing and summarizing our findings and prospects for future work.

6.1 Architecture and Design

Momba is centered around the Python programming language and the JANI-model interchange format [Bud+17], which have been instrumental ingredients to meet the requirements (HLR1) and (HLR5), respectively. The performance critical parts of Momba have been written in Rust in order to meet (HLR6).

[16] This involved agreeing with the authors of the other tools on a methodology for the competition and then carrying out the competition in the way agreed upon by the authors of all participating tools.
[17] https://web.archive.org/web/20220124234200/https://fm21.momba.dev/

The idea to harvest a general purpose programming environment for formal modelling is not new at all. For instance, the SVL language combines the power of process algebraic modelling with the power of the bourne shell. As part of many CADP installations [Gar+13; GLM18], it is in daily use since its inception [Fer+96]. Some formal modeling tools also already provide Python bindings [Hen+22; Dur+16]. Momba strives to be not yet another incarnation of these ideas.

While the construction of formal models clearly is an integral part of Momba, Momba aims to be more than just a framework for constructing models with the help of Python. Most importantly, as stated above, it should also provide features to work with these models such as a simulator or interfaces to model checking tools. At the same time, it should also not just be a binding to an API developed for another language, say C++. Momba is *tool-agnostic* and aims to provide a pythonic interface for dealing with formal models while leveraging existing tools. Momba covers the whole process from model creation through validation to analysis.

Why Python? Python is a popular high-level programming language, preferred by many for its ease of use and vast ecosystem. Especially within the data science community, Python is the go-to language for data analysis and machine learning leveraging libraries such as TensorFlow [Aba+16] and scikit-learn [Ped+11]. Around these libraries, scientific general purpose tools such as Jupyter [Klu+16] have emerged. Jupyter provides a platform for documenting scientific experiments and their results in a reproducible way combining code, data, and documentation.

By basing our efforts on a popular language that is already appreciated by scientists and established in the scientific community, we hope to lower the entry barrier, especially for those outside the formal methods community. We believe that most individuals of Momba's target audience are already familiar with Python. In case someone is not, Python is also known for being easy to learn.

In addition to being already widely established and easily accessible, our vision further is to leverage Python's ecosystem and the excellent tools developed by the scientific community for dealing with formal models. Imagine, a Jupyter notebook documenting a model, including the code to construct it, with interactive visualizations of the model itself and various analysis results.

Why JANI? Traditionally, most analysis tools for formal models came with their own modeling languages and formats. The resulting fragmentation hindered interoperability between and comparability across different tools. JANI (cf. Section 2.2) [Bud+17] has been conceived with the vision to put an end to this fragmentation. It provides a solid, well-established, and powerful compositional modeling formalism for a variety of different kinds of systems involving concurrency, probabilistic and real-time aspects, as well as continuous dynamics.

A JANI model is composed of multiple automata interacting with each other through synchronization. In the context of JANI, an automaton is essentially a transition system enriched with quantitative capabilities (e.g., time and probabilities) and programming concepts (e.g., variables and conditionals).

Since its inception, JANI has been adopted by many quantitative model checkers [Hah+14; HH14; Hen+22] while for others translators have been developed [Hah+14; Hen+22]. This broad tool support enabled cross-tool comparability and fostered competitive evaluations among tools [Har+19; Hah+19; Bud+20]. Beyond the quantitative verification community, JANI has also been discovered by the planning community [Hof+20; Kla+20] for modeling planning problems. Given its adoption by the community, JANI is the natural foundation for a project like Momba. Momba supports all features of the JANI-model specification and some of its optional extensions. The vast tool support for JANI models enables Momba to build upon existing research and to outsource computation-intensive tasks via unified interfaces.

Why Rust? While performance was not one of Momba's original design goals, further work building upon Momba [i.e. KH23; Gro+22] made it a priority. In particular, the implementation of the continuous time verdictor algorithm (cf. Chapter 5) required a high-performance model exploration engine supporting timed automata. While Python is known for its ease of use and great scientific ecosystem, raw performance is not one of its strength. To achieve a performance competitive with other state-of-the-art tools, a different language became necessary.

Rust[18] is a relatively new language offering performance comparable to C and C++. In contrast to C and C++, it comes with powerful static analysis tools guaranteeing memory- and thread-safety [Jun+21]. In particular, this means that the Rust compiler is able to guarantee that a given program written in *Safe Rust*, does not exhibit any *undefined behavior* (UB). Undefined behavior, caused by bugs in a program, means that a program's behavior is unpredictable as per the language's specification, i.e., a program with UB can behave in any way including but not limited to crashing and producing subtlety wrong results. Most Rust programs are written entirely in Safe Rust. For some low-level algorithms and data structures, however, *Unsafe Rust* is required. In Unsafe Rust, the programmer is responsible for upholding certain invariants that the compiler cannot statically verify, like in C or C++.

By using Rust for the implementation of the simulation and state space exploration engine, Momba is able to achieve state-of-the-art performance. In addition, the results produced by Momba can be considered more trustworthy as Momba is mostly written in *Safe Rust*, thereby limiting the potential for UB to small, well-audited parts of Momba's implementation. An additional advantage of using Rust is that it is easy

[18] https://web.archive.org/web/20240725190358/https://www.rust-lang.org/

to develop modules for Python interoperability thanks to PyO3[19] and Maturin[20]. PyO3 is a Rust library providing types and interfaces to develop Python modules in Rust and Maturin is a tool to build Python packages from Rust code that uses PyO3. Together, they enable building Python packages in Rust that can be easily distributed and readily imported into Python projects.

Architecture. Figure 6.1 depicts the high-level architecture of Momba. Momba is split into three core Python modules providing (i) an object-oriented API to inspect, create, and modify JANI models, (ii) a state space exploration engine which can be used for simulation, and (iii) unified interfaces to external model analysis tools [KKH21]. As Momba is extensible, we also developed a Python module implementing the well-established OpenAI Gym API[21] for training and verifying decision-making agents (iv) [Gro+22]. All Python modules are inter-compatible. For instance, the state space exploration and analysis modules take objects of the modeling module as input. The implementations of the techniques for verdictors developed in this thesis are also part of the broader Momba ecosystem (v). For performance reasons, they have been implemented in Rust and directly interface with Momba's state space exploration engine (vi), where applicable.

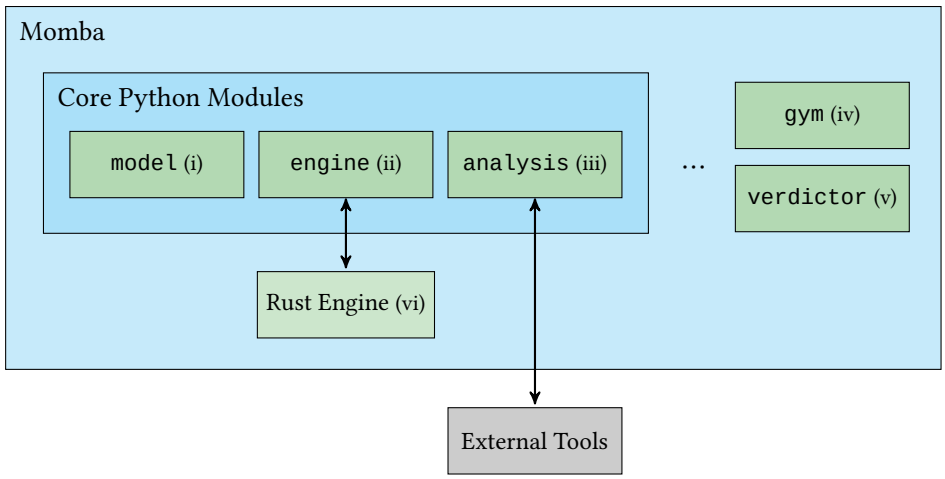


Figure 6.1: Momba's architecture comprises three core Python modules, a state space exploration engine written in Rust, and additional tools and modules implementing the techniques developed in this thesis and providing training and verification capabilities for decision-making agents.

[19] https://web.archive.org/web/20240714004505/https://pyo3.rs/v0.22.1/
[20] https://web.archive.org/web/20240523200025/https://www.maturin.rs/
[21] https://web.archive.org/web/20240622014439/https://gymnasium.farama.org/index.html

6.2 Momba: User Perspective

We now take on a user perspective and showcase the usage of the various features provided by Momba. Note that this thesis does not aim to be a comprehensive manual or documentation for Momba. Momba's documentation, including a user guide in the style of a tutorial and a comprehensive API reference, are available online[22] and as part of Momba's artifact (AT5). While not exhibited in detail here, the guide and the API reference are key for fulfilling the requirement (HLR1).

Installation. Momba can be installed via the Python Package Index[23] (PyPI), the standard way to distribute Python packages. To install Momba, including all modules, users can use the usual Python package management tools, e.g., `pip`. For instance, to install Momba with all of its optional components, run:

```
pip install "momba[all]"
```

We provide precompiled packages for Windows, Linux, and MacOS for the x86_64 and ARM64 architectures. Hence, Momba is straightforward to install on any operating system and architecture popular at the time of writing.

Example: Racetrack. In what follows, we demonstrate multiple facets of Momba using a variant of Racetrack, a well-known benchmark in autonomous AI decision making [BBS95; PZ14] which has also found its use in several model checking contexts [Gro+20b; Gro+20a; Bai+20; Gro+22]. In two of these works, the author of this thesis was also involved in [Bai+20; Gro+22]. Using the example of Racetrack, we go through the entire process from the programmatic construction of a family of models through their validation to their analysis. For each step, we highlight what Momba has to offer in terms of effectively supporting the process.

Originally Racetrack has been a pen-and-paper game [Gar73]. A *track* is a two-dimensional grid comprising *start*, *goal*, *wall*, and *blank* cells (see Figure 6.2, next page) [BBS95]. A car starts off with some initial velocity from a start cell, with the objective to reach a goal cell as fast as possible without crashing into a wall. The car is controlled by nine possible actions modifying the current velocity vector. Racetrack naturally lends itself as a benchmark for sequential decision making in risky scenarios, in particular, when extended with probabilistic noise. In a variety of such noisy forms, it has been adopted as a benchmark for *Markov Decision Process* (MDP) algorithms in the AI community [BBS95; BG03; MG05; Pin+13; PZ14].

For our demonstration, we consider multiple *variants* of Racetrack giving rise to a family of MDPs, which have already been studied [Bai+20] from a feature-oriented perspective [Chr+18]. For example, there are different tank options and fuel

[22] https://web.archive.org/web/20240519104251/https://momba.dev/
[23] https://web.archive.org/web/20240724073857/https://pypi.org/

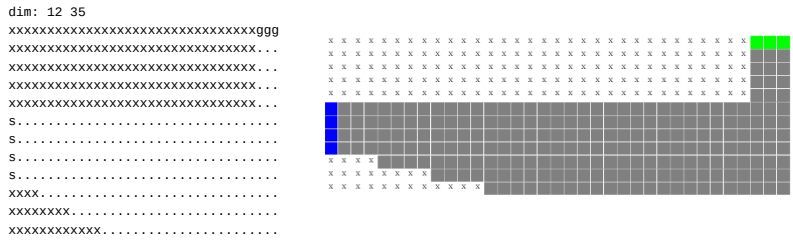


Figure 6.2: Textual representation (left) and picture of a track (right): Start cells in blue (s), goal cells in green (g), and wall cells marked with x.

is consumed according to various consumption models. In addition, there are different undergrounds inducing probabilistic noise modeling slippery road conditions. Clearly, this modeling scenario is beyond what is possible with mere model parametrization, especially so because we are interested in the car's performance on different tracks each inducing its own MDP [BBS95].

6.2.1 Scenario-Based Model Construction

Typically, formal models are not constructed out of thin air but based on some kind of scenario description existing upfront. Such descriptions usually comprise an operational characterization of the behavior to model together with additional and sometimes more formal information about the specific case. The Racetrack use case is no exemption: Here, a textual description of the behavior of the car is provided together with a specific track and a specification of the variant.

Naturally, Python can be used to nicely capture the formal parts of a scenario description in various data structures. Combined with a domain-specific parser for configuration files, scenario descriptions are interchangeable and easy to interface with the code for model construction. In our case, a textual representation of the track (cf. Figure 6.2) [BBS95] is provided and parsed together with additional parameters, like the size of the tank and the type of the underground, into a data structure tailored to that purpose. Figure 6.3 shows an excerpt of this data structure for the track. It consists of two classes, `Coordinate` and `Track`, capturing coordinates of cells and entire tracks, respectively. A track has a height and a width, and consists of sets of coordinates for the respective types of cells. For further details, we refer the interested reader to the `racetrack` Python package, which has been developed by the author of this thesis and is available online[24] and as part of Momba's artifact (AT5).

Now, how does Momba support the construction of models from such data structures? Momba provides an object-oriented modeling API together with convenience

[24] https://web.archive.org/web/20230926041226/https://pypi.org/project/racetrack/

```python
@dataclass(frozen=True, order=True)
class Coordinate:
    x: int
    y: int

@dataclass(frozen=True)
class Track:
    width: int
    height: int

    blank_cells: FrozenSet[Coordinate]
    blocked_cells: FrozenSet[Coordinate]
    start_cells: FrozenSet[Coordinate]
    goal_cells: FrozenSet[Coordinate]

    ...
```

Figure 6.3: Excerpt of the data structure for the track of a Racetrack model.

functions effectively turning Python into a syntax-aware macro language for the programmatic construction of models in a modular fashion. The provided APIs and convenience functions also enable the early catching of modeling errors. For instance, based on an instance `track` of the `Track` class, Momba can be used to declare JANI constants for the track's width and height:

```
ctx.global_scope.declare_constant("WIDTH", INT, value=track.width)
ctx.global_scope.declare_constant("HEIGHT", INT, value=track.height)
```

Here, `global_scope` is an object representing the global scope for declaring constants and variables within a JANI model represented by `ctx`. Every JANI model has a global scope as well as local scopes for each automaton, respectively. Variables are declared analogously to constants providing an initial value instead of a constant value. Figure 6.4 (next page) shows a variable declaration encoding the cells of a track, again given by a `Track` object, as a two-dimensional array in JANI. Each type of cell is represented by an integer between zero and three. By indexing the array with the coordinates of a cell, one obtains the type of the cell. Note that we cannot use a JANI constant here, as constants are not allowed to be arrays in JANI. Furthermore, we make the variable *transient*, which means that it only has a value when a transition is taken, i.e., the variable does not end up in the states of the resulting model, thereby reducing their size. We use the `ArrayValue` class provided by Momba to construct the two-dimensional array based on the `Track` object. Clearly, such constructions go beyond what is possible with mere model parametrization.

In the case of Racetrack, we also declare variables `car_x` and `car_y` for the x and y position of the car. To determine whether the car is outside the track, we then use the declared constants and variables. Momba provides a function `expr` for

```
value = ArrayValue(
    ArrayValue(
        ensure_expr(track.get_cell_type(Coordinate(x, y)).number)
        for x in range(track.width)
    )
    for y in range(track.height)
)
ctx.global_scope.declare_variable(
    "map",
    typ=array_of(array_of(types.INT.bound(0, 3))),
    is_transient=True,
    initial_value=value,
)
```

Figure 6.4: Variable declaration encoding a track as a two-dimensional array.

constructing JANI expressions. To construct a JANI expression indicating whether the car is outside of the track, we can use the following code:

```
expr("car_x >= WIDTH or car_x < 0 or car_y >= HEIGHT or car_y < 0")
```

While JANI does specify the structure of expressions in terms of abstract syntax trees, it does not specify a concrete syntax. To feel familiar to users of Python, Momba's `expr` function adopts a syntax that resembles Python expressions.

Expressions constructed with `expr` can then be used in a property, e.g., to indicate a crash when the car is outside of the track, or in guards of the edges of the automata to prevent the car from going outside of the track in the first place.

Syntax-Aware Macros. A distinguishing feature of Momba is that it allows the interpolation of expressions in a syntax-aware fashion. For our Racetrack use case, we want to be able to use different fuel consumption models. We capture them in terms of *macros* mapping JANI expressions to JANI expressions:

```
linear = lambda dx, dy: expr("abs($dx) + abs($dy)", dx=dx, dy=dy)
quadratic = lambda dx, dy: expr("$linear ** 2", linear=linear(dx, dy))
```

A macro is simply a Python function leveraging Momba's functionality for constructing JANI expressions. In this case, both macros take JANI expressions for the current velocity of the car in x and y dimension and return a JANI expression for the resulting fuel consumption, which is either *linear* or *quadratic* in the velocity.

In contrast to how macros work in languages like C, syntax-aware macros using Momba's `expr` function prevent surprises from mere text-based expansion. For instance, in case of the quadratic fuel model, naive text-based expansion or interpolation would lead the last summand to being squared instead of the whole sum. Using Momba's `expr` function prevents that as it understands the structure of the

individual expressions and combines them on the syntax tree level.

```
def construct_tank(ctx, tick_action, fuel_model):
    automaton = ctx.create_automaton(name="tank")
    initial = automaton.create_location(initial=True)

    consumption = fuel_model(expr("car_dx"), expr("car_dy"))
    fuel = expr(
        "min(TANK_SIZE, max(0, fuel - floor($consumption)))",
        consumption=consumption,
    )

    automaton.create_edge(
        source=initial,
        destinations=[
            create_destination(
                initial,
                assignments={"fuel": fuel},
            )
        ],
        action_pattern=tick_action,
        guard=expr(
            "fuel >= $consumption",
            consumption=consumption,
        ),
    )

    return automaton
```

Figure 6.5: Simplified version of the code for constructing the tank automaton.

Example: Tank Automaton. Figure 6.5 shows an example for constructing an entire automaton within a JANI model.[25] Again, `ctx` represents the model. The automaton constructed here models the tank of the car in the Racetrack use case. To this end, the function takes a tick action and a fuel model as input. Here, `tick_action` is an action object which is used elsewhere to synchronize with the tank automaton. Whenever a tick happens, fuel is consumed. The parameter `fuel_model` is one of the earlier defined macros for the different fuel consumption models. Being Python functions, macros can be easily passed around. To create the automaton, the method

[25] For a comprehensive documentation of all the capabilities of the modeling API, we refer the reader to Momba's documentation: https://momba.dev/reference/model/. This documentation is also included in (AT5) and generated directly from the Python source code.

`create_automaton` on `ctx` is used. Afterwards, an initial location for the automaton is created using `create_location`. Based on the provided fuel consumption model, an expression for the fuel consumption is constructed taking the current velocity of the car into account. The expression for the fuel consumption is then used to update the fuel stored in the `fuel` variable whenever the tick action occurs and there is still sufficient fuel left. This is achieved by constructing an assignment. To create this assignment, again Momba's `expr` function is used to construct an expression for the updated fuel level, that is, the current fuel minus the fuel consumption lower bounded to zero and upper bounded to the tank size. Note that the fuel consumption is also used as a guard for the constructed edge. As a result, the tank automaton may block transitions within the model in case the fuel is empty.

Model Construction API. As demonstrated by the examples, Momba provides an object-oriented API for the programmatic construction of models going well beyond what is possible with model parametrization. In particular, it enables the construction of models from preexisting scenario descriptions. Most of these functions provide all kinds of comforts, for instance, directly checking the types of the involved expressions. For example, adding an edge to an automaton whose guard is not a Boolean expression will result in an immediate error. Hence, Momba fulfills the requirement (HLR2). There is no conversion step necessary to turn the constructed model into JANI besides executing the Python code. Momba's internal model representation is based on the JANI specification and so is the provided API to construct models. Hence, every JANI model can be specified programmatically with Momba. This is a great benefit because JANI does not offer any features for building models programmatically while Momba offers the whole range of possibilities of Python.

For Racetrack, using syntax-aware macros and Momba's model construction API, we arrive at a Python script for generating a collection of JANI models from scenario descriptions comprising a track and specifying a variant. Again, the script is part of the `racetrack` Python package, which can also be found in artifact (AT5). Iterating over possible scenario descriptions, hundreds of JANI models can be generated fully automatically and then be analyzed.

Related Work. With Stormpy, the model checker Storm also provides Python bindings [Hen+22]. While Strom does support JANI models, the Python bindings do not provide an API to construct them. Furthermore, unlike Momba, Storm does not support timed automata. Support for timed automata is required to implement the continuous time verdictor algorithm presented in Chapter 5.

With the PRISM preprocessor[26], the popular PRISM model checker comes with a text-based preprocessor for the PRISM modeling language [KNP11]. Using Storm,

[26] https://web.archive.org/web/20240430102538/https://www.PRISMmodelchecker.org/PRISMpp/

PRISM models can be translated to JANI and vice versa. While supporting some programming constructs, the PRISM preprocessor is more limited than Python. For instance, it does not support reading and processing external files, e.g., as we have done with Racetrack track files. In contrast, with Momba, the full capabilities of Python can be used. Furthermore, Momba does not require users to learn a domain-specific preprocessing language and existing tools from the Python ecosystem (editor integrations with autocompletion, type checking, etc.) can be used. Another advantage of Momba over text-based preprocessing are Momba's syntax-aware interpolations that preserve the syntactical structure of expressions which may be lost by mere text-based substitutions (recall discussions in Section 6.2.1).

ProFeat [Chr+18] is another tool for generating PRISM models centered around family-based compositional modeling, while also enhancing the PRISM-inspired input language by loops, arrays, and macro-like functions. ProFeat is tailored towards modeling configurable systems. In comparison to Momba, similar considerations as for the PRISM preprocessor apply. Momba offers the full power of Python, with which most users are probably already familiar and which allows existing tooling to be reused. Still, for a family-based model, a more tailored approach as pursued by ProFeat can be more feasible. In particular, ProFeat allows the direct and explicit modeling of features and other aspects of configurable systems. In comparison, Momba is not tailored to this use case and requires a more manual and less explicit approach.

6.2.2 Validation by Simulation

Having our models ready, we have to somehow gain confidence that they actually model what we want them to, before handing them over to analysis tools or applying the techniques presented in this thesis. One way of gaining confidence into a model is by simulating its behavior and manually checking it for consistency with one's own understanding of what the model should do. Just like any kind of debugging, this can be a tedious and frustrating process, especially with text-based traces generated by some generic simulator. Momba instead comprises a built-in simulation engine, enabling rapid development of interactive visualizations.

Interactive Racetrack Game. In the case of Racetrack, we developed an interactive implementation of the game by visualizing the current state of the model and mapping user input to its transitions. This effectively allows a user to steer the car through a track thereby exploring a model's behavior, testing edge cases as in a racing game, and ultimately gaining confidence in the model. Figure 6.6 (next page) shows the interactive visualization. Here, the car is indicated by a yellow asterisk and the user can steer by entering acceleration values. Certainly, there is ample room for beautification of this simulator (see TraceVis [Gro+20a] for example) but for rapid model development and testing this is not needed. After playing around with the

interactive simulation for a while and testing various edge cases, we gained further confidence that the models we built are indeed adequate. The game is also available as part of the `racetrack` Python package for the reader to play around with.

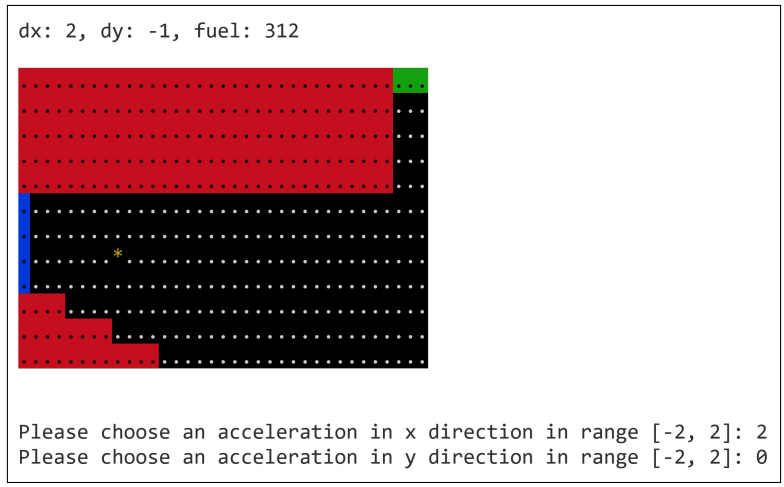


Figure 6.6: Interactive visualization using Momba's simulation engine.

Remark. Using the simulator essentially amounts to manual testing of the model. Unless testing is exhaustive, it cannot provide any strict guarantees as it may miss certain cases where the model does not behave as intended. Still, testing allows gathering evidence that the model faithfully represents the intended behaviors of the game. The simulation engine of Momba can also be used for more systematic and automated testing. Ultimately, these measures are meant to provide convincing evidence corroborating a model's adequacy.

State Space Exploration Engine. Momba's built-in state space exploration engine supports a variety of different JANI models, including timed models. It has been written from scratch in Rust with easy accessibility from Python in mind.

Figure 6.7 shows an example of using the state space exploration engine for the Racetrack model, which is a discrete-time model.[27] After creating an instance of the `Explorer` class, the initial states of the model can be queried. In this case, the model simply has a single initial state. The state object then exposes the global and local environments binding values to variables. In addition, it can be used to query the locations of the individual automata and the outgoing transitions of a state. In

[27] For a comprehensive documentation of all the capabilities of the state space exploration API, we refer the reader to Momba's documentation: https://momba.dev/reference/engine/. This documentation is also included in (AT5) and generated directly from the Python source code.

```
explorer = engine.Explorer.new_discrete_time(network)

(state,) = explorer.initial_states

print("x:", state.global_env["car_x"].as_int)
print("y:", state.global_env["car_y"].as_int)

state = random.choice(state.transitions).destinations.pick().state

print("x:", state.global_env["car_x"].as_int)
print("y:", state.global_env["car_y"].as_int)
```

Figure 6.7: A simple state space exploration example.

Figure 6.7, a successor state is chosen by selecting a transition uniformly at random and then picking one of its destinations according to the probability distribution of the transition. Recall that in the case of Racetrack the underlying model is an MDP, so we have nondeterminism and in addition each transition comes with a probability distribution over potentially multiple successor states. In principle, nondeterminism can be resolved by uniform random sampling or by querying an external oracle such as, in the case of our interactive visualization, the user, a testing framework, or even a neural network as done for DSMC [Gro+20b; Gro+22]. For each step, the engine provides all the necessary information about the current state and possible transitions, including actions, probability distributions, and, in the case of timed models, possible time delays. This information can then be extracted and used to display whatever is of interest for understanding and investigating the behavior of the model under scrutiny. We conclude that Momba's state space exploration engine can indeed serve as a platform for model validation by simulation. In particular, we have demonstrated how it can be used to prototype a tool for interactive model exploration and visualization. Hence, Momba fulfills requirement (HLR3).

6.2.3 Invoking Analysis Tools

We have seen how Momba can be used to construct models and explore their behavior by simulation. After having gained confidence through simulation that a model is adequate, we are now ready to harvest the benefits of formal modeling. To this end, we demonstrate how Momba's uniform APIs can be used to apply state-of-the-art analysis tools. Under the hood, this leverages the JANI-model interchange format. Momba provides the necessary functions to define properties and hand our models, with the respective properties attached to them, over to analysis tools.

For the Racetrack use case, imagine that we are interested in the maximal probability of reaching a goal cell with a non-empty tank from a given start cell: Using

Momba's `prop` function, we can express this property as follows:

```
goal_property = (
    prop("min({Pmax(F(map[car_y][car_x] == 3 and fuel > 0)) | initial})")
)
```

Note that we use the earlier declared variable `map` here to determine the type of cell the car is currently on. The number `3` indicates that the cell is a goal cell.[28] We can now pass this property together with the model to a model checker. For instance, to invoke `mcsta` of the Modest Toolset [HH14], we can use:

```
modest_checker = tools.modest.get_checker(accept_license=True)
results = checker.check(model, properties={"goal": goal_property})
print("Probability:", results["goal"])
```

After generating a model with the car starting from position $(0, 7)$ on the track depicted in Figure 6.2 and with sand as underground, `mcsta` calculates a probability of 87.5 % when invoked by Momba with the model. The model checker Storm can be invoked analogously. Furthermore, Momba provides an interface for cross-checking results by invoking multiple tools and comparing their results.

Momba also exposes the *deep statistical model checking* (DSMC) capabilities of the Modest Toolset [Gro+20b; Gro+22]. This allows the application of DSMC to neural networks as well as arbitrary Python functions [Gro+22].

Note that Momba automatically takes care of downloading the necessary tool or invoking it within Docker. For end users, this translates to a fully integrated experience, where they do not have to worry about downloading and installing the right tools—Momba simply does that for them. Thanks to the JANI-model interchange format, Momba connects well to the established tools of the ecosystem, thereby fulfilling requirement (HLR4) and requirement (HLR5).

6.3 Evaluation: State Space Exploration

We implemented the techniques presented in this thesis using explicit state techniques within Momba. More concretely, Momba's state space exploration engine is used to (partially) construct the state spaces of models towards an implementation of the techniques presented in the previous two chapters.

Beyond the techniques presented in this thesis, state space exploration engines form the foundation of numerous quantitative modeling tools, playing a pivotal role in their functionality. Explicit state model checkers, such as Storm with its `sparse` engine [Hen+22] and `mcsta`, the explicit state model checker of the Modest Toolset [HH14], rely on exploration engines to exhaustively construct the complete state space of a model before applying model checking algorithms. Additionally,

[28] JANI has no enums and we decided against using Python's enums here as they would require additional definitions complicating the presentation of the example.

statistical model checkers, such as `modes` [BHH12; Bud+18], leverage exploration engines to generate large amounts of traces for statistical analysis. Furthermore, the author of this thesis contributed to work using Momba's state space exploration engine for training and verifying decision-making agents [Gro+22].

In an effort to better understand the performance characteristics and features of the exploration engines utilized in different tools, we systematically compare and benchmark them. This evaluation has been carried out by the author of this thesis as part of QComp 2023 [And+24] in accordance with a methodology agreed upon by authors of all participating tools. For evaluation purposes, we consider the time and space needed for building an explicit representation of the complete state space of a model. Additionally, we compare the engines based on qualitative criteria such as the types of models they can handle and the interfaces they provide.

6.3.1 Tools and Engines

For the evaluation, we consider three tools and their respective engines: The Modest Toolset [HH14], Storm [Hen+22], and Momba [KKH21]. While Momba is a relatively new tool, both the Modest Toolset and Storm are well-established tools representing the state of the art when it comes to state space exploration of JANI models and quantitative model checking. Since all three tools support JANI, we will employ it as a foundation for comparing and contrasting their capabilities.

Besides the major differences discussed in the following, there are also some minor differences that do not directly map onto JANI model types and extensions, e.g., support for multi-dimensional arrays and complex specifications for initial states. For our comparison, we will leave those details aside and focus on major differences which concern supported JANI model types and extensions.

The Modest Toolset. The Modest Toolset [HH14] is a collection of tools designed to facilitate the modelling and analysis of a wide range of systems, encompassing hybrid, real-time, stochastic, and distributed systems. The tools of the Modest Toolset share a common state space exploration engine written in C#.

The engine of the Modest Toolset supports all types of models specified by JANI, including all JANI extensions.[29] In that regard, it stands out as the most versatile among those engines we consider here. For (probabilistic) timed automata, the engine supports digital clock semantics, explicit valuations, clock regions, as well as clock zones. In addition, it supports a symbolic treatment of continuous variables for hybrid models, further enhancing its capabilities.

[29] The `nondet-selection` JANI extension is only supported for simulation but not for exhaustive state space construction.

In contrast to both Storm and Momba, which both provide public interfaces to their engines, the Modest Toolset's engine is intended for internal use within the Modest Toolset only and does not provide a public interface. Nevertheless, it is noteworthy that the Modest Toolset supports the transpilation of models to Python code which can then be used to explore the state space of a model.

Storm. Storm [Hen+22] is a state-of-the-art model checker with a modular core putting an emphasis on time and memory efficiency. Written in C++, Storm's modular design enables the utilization of different model checking engines catering to the characteristics of different models. Notably, Storm excels in efficient symbolic model checking through its `dd` engine, leveraging (MT)BDDs [FMY97; Bry86].

For the QComp 2023 competition, Storm participated with its `sparse` and `dd-to-sparse` engines. While Storm's `sparse` engine, like the engines of the Modest Toolset and Momba, adopts a conventional explicit approach to construct the state space of a model, the `dd-to-sparse` engine is based on first constructing a symbolic BDD representation of the state space and subsequently translating this symbolic representation to a traditional explicit representation.

Storm supports all discrete and continuous-time model types specified by JANI, except timed and hybrid automata. The supported JANI extensions are `arrays`, `derived-operators`, `functions`, and `state-exit-rewards`.

Storm provides a C++ and a Python interface, as part of Stormpy, to its state space exploration engine. While the C++ API is fully featured, the Python API only supports the exploration of the entire state space of JANI models but not the simulation of individual traces. For PRISM models, however, there is no such limitation.

In contrast to the Modest Toolset and Momba, Storm offers support for arbitrary precision arithmetic. This feature enables precise calculations and analysis, particularly when dealing with models that require high precision.

Momba. While Momba itself supports all of JANI, Momba's state space exploration engine is more limited. The exploration engine supports all discrete-time model types and flavors of timed automata specified by JANI, except stochastic timed automata. The supported JANI extensions are `arrays`, `derived-operators`, `named-expressions`, and `trigonometric-functions`. In particular, the `functions` extension is not supported yet. For timed automata, it supports explicit valuations as well as clock zones. The Python API also provides functionality that goes beyond mere exploration, for instance, arbitrary JANI expressions can be evaluated in a given state. In addition to Momba's traditional state space exploration engine, Momba also participated in QComp 2023 with an alternative new engine supporting a parallelized exploration mode, harnessing the potential of multi-core systems. However, it is important to note that this alternative engine does not currently support timed automata and is not exposed via the Python API.

In addition to the aforementioned JANI extensions, Momba also implements support for an inofficial JANI extension for value passing via actions. This extension has been implemented as actions also serve the purpose of observables for the techniques developed in this thesis. By using this extension, values of variables and arbitrary expressions can be made observable via actions.

Qualitative Comparison: Summary. While the features offered by all three tools and engines overlap, neither is a subset of the other. Overall, the Modest Toolset's engine is most complete when it comes to JANI support in terms of model types and extensions. However, it lacks an accessible API. Storm on the other hand, does not support timed automata, while providing both a C++ and Python API. Momba's engine supports timed automata and provides a Rust and Python API, however, it lacks support for some of the model types and optional JANI extensions. In conclusion, the tools and engines are qualitatively incomparable and users must decide which tool or engine to choose based on their specific requirements.

6.3.2 Benchmark Setup and Results

Having presented and compared the tools and engines qualitatively, we now turn to quantitative benchmarks. In what follows, we describe the benchmarking methodology, including the selection of benchmark models, the metrics used for evaluation, and the experiment environment. Furthermore, we present and discuss the results obtained by running all tools on the selected benchmarks.

The tools, source code, benchmarks, and raw data required to reproduce the experiments is provided as part of the artifact (AT4).

Benchmarks and Methodology. For our evaluation, we utilize the quantitative verification benchmark set (QVBS) provided by QComp as the foundation for benchmarking the tools. To ensure a meaningful comparison, we focus exclusively on discrete-time models, as these are supported by all the participating tools. Out of the initial 229 benchmarks, 25 benchmarks resulted in timeouts after 30 minutes or were unsupported by all tools. Hence, the following analysis focuses on the remaining 204 benchmarks. For each benchmark, we measure the time required by each state space exploration engine to construct the entire state space of the respective model. Additionally, we track the number of states counted by the engines and assess the memory consumption associated with each state, whenever applicable. All benchmarks have been executed on Linux and a computer equipped with 128 GB RAM and a 16-core AMD EPYC-Milan Processor running at 3.4 GHz.

Benchmark Overview. Table 6.1 (next page) shows the number of benchmarks per tool and outcome. The table displays the number of benchmarks categorized into

four different outcomes: *solved*, *unsupported*, *timeout*, and *error*. *Momba (v2,`seq`)* is Momba's alternative engine ran in sequential mode and *Momba (v2,`par`)* is Momba's alternative engine ran in parallel mode, i.e., exploiting multiple CPU cores.

Engine	Solved	Unsupported	Timeout	Error
The Modest Toolset	194	9	1	0
Momba (v1)	159	45	0	0
Momba (v2,`seq`)	159	45	0	0
Momba (v2,`par`)	154	45	5	0
Storm (`dd-to-sparse`)	195	3	2	4
Storm (`sparse`)	202	0	2	0

Table 6.1: Number of benchmarks per outcome and tool.

The nine benchmarks not supported by the Modest Toolset's engine use a complex specification for the initial states. While the Modest Toolset does overall support the greatest number of JANI models, its support of complex specifications of initial states is limited. The 45 benchmarks not supported by Momba use the `functions` JANI extension and are a superset of the nine benchmarks not supported by the Modest Toolset. Like the Modest Toolset, Momba also lacks support for complex specifications of initial states. The three benchmarks not supported by Storm's `dd-to-sparse` engine use assignment indices[30] and for four benchmarks the same engine returned an error due to the BDD implementation running out of memory. The timeouts ocurred all for different benchmarks, respectively.

While the number of states reported by Storm and Momba is the same for all benchmarks and engines, the Modest Toolset sometimes reports less states which presumably is due to some state space reduction optimizations.

Running Time. Figure 6.8 depicts the running time for each benchmark in relation to the total number of states of the respective benchmark. Figure 6.9 (p. 178) presents the cumulative number of benchmarks solved within a certain time. For presentation purposes, we chose to cramp the running times at $0.1\,s$ and restrict the plots to benchmarks with more than 10^5 states. For smaller benchmarks, the differences in running times are practically insignificant. Additionally, Figure 6.9 is restricted to

[30] Normally, all assignments are executed simultaneously, i.e., the expressions of all assignments are evaluated before any changes to variables take effect. Assignment indices enable the specification of an order in which assignments are executed, thereby allowing the expressions of some assignments to see changes made by previous assignments within a transition between states.

benchmarks supported by all engines to prevent skewing of the plot.

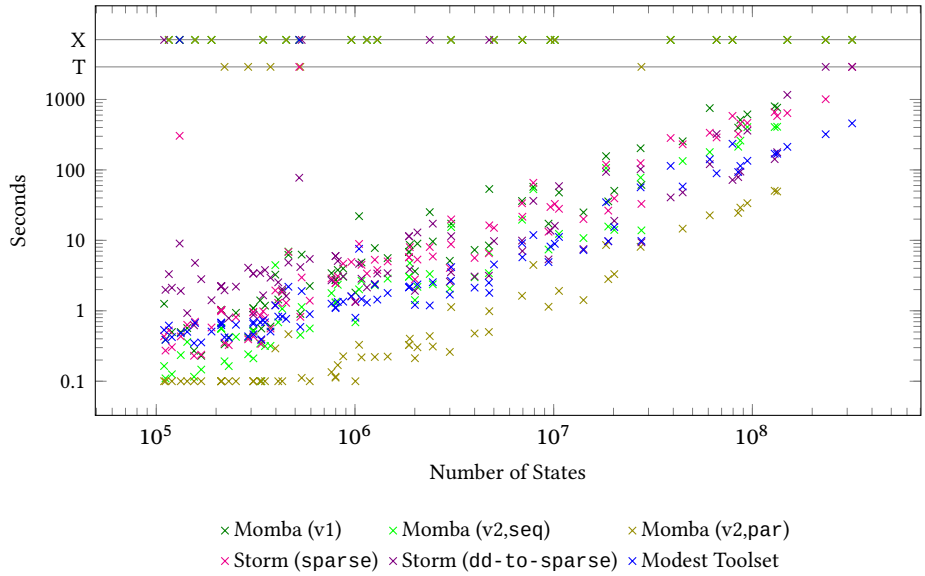


Figure 6.8: Running times in seconds (on the y-axis) in relation to the total number of states (on the x-axis). The marks at the top indicate timeouts (T), and unsupported benchmarks and benchmarks returning an error (X).

From these results it is evident, that the approach taken by the `dd-to-sparse` engine of Storm only pays off for larger models. And still, it is rarely faster than the conventional explicit engine of the Modest Toolset. In fact, among those engines exclusively using a single core, the Modest Toolset's engine is almost always the fastest although it has a larger startup overhead. This does not come as a surprise because the Modest Toolset's engine is based on compiling JANI models to C# bytecode which has to go through .NET's JIT,[31] enabling near-native performance but also coming with some initial overhead until the JIT is properly warmed up. Notably, Storm's `dd-to-sparse` engine uses multiple cores as the underlying BDD implementation, Sylvan [DP17], is parallelized. Momba's alternative parallel engine (v2,`par`) is the only other engine leveraging multiple cores. It is always faster (except for the five timeouts) for benchmarks of a significant size than any other engine. The average speedup when compared to its own sequential version is a factor of 9.1. In general, though, the running times of all engines are often quite similar.

Note that, as Storm and the Modest Toolset are primarily model checkers, they do a bit more work than Momba by creating a sparse matrix representation of the

[31] A *just-in-time compiler* (JIT) compiles bytecode to native machine code at runtime.

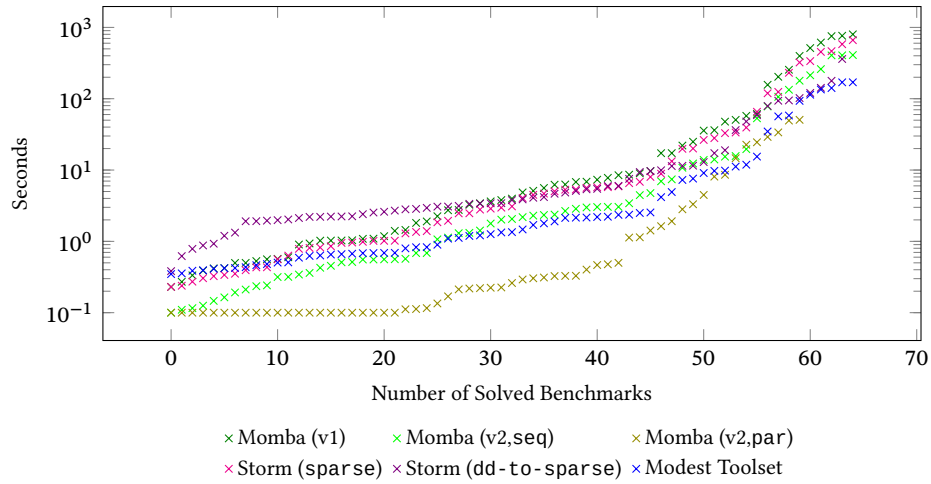


Figure 6.9: Running time in seconds (on the y-axis) in relation to the number of benchmarks solved in that time (on the x-axis).

transitions and computing atomic propositions. We expect the performance impact of this to be minor, however, we were unable to measure it directly.

Memory Consumption. Another interesting dimension when it comes to explicit state space construction is the required memory. State spaces can be quite huge and thus it is important to store them efficiently. For the traditional explicit state engines, the size of the state space is the product of the number of states and the size of each state. Figure 6.10 shows the size of the state spaces in relation to the number of states computed based on the number of states and the size of each state. Note that the sequential and parallel variant of Momba's alternative engine use the same representation. In contrast to the Modest Toolset, Storm's `sparse` engine and Momba's alternative engine use a more space efficient bit-packing representation of states thereby reducing the amount of required memory. Momba's original engine uses the worst representation and always requires at least 16 bytes per variable independent of the actual domain of the variable.

Conclusion. The presented results show that all engines are roughly comparable with respect to the time it takes to construct the entire state space of a model. In particular, the results show that both of Momba's engines can compete with the state of the art. Hence, Momba satisfies requirement (HLR6).

Storm's `dd-to-sparse` engine may only be advantageous for some large models while incurring a high overhead for small models. Among single-core engines, the engine of the Modest Toolset is almost always the fastest, especially for large models,

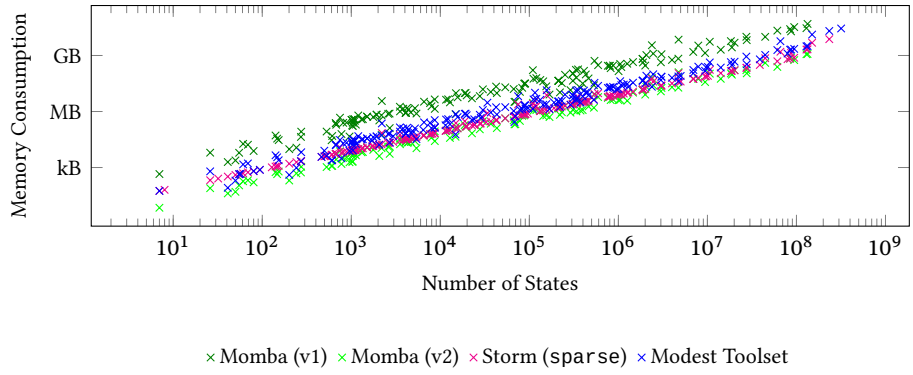


Figure 6.10: Memory consumption of the entire constructed state spaces in relation to the number of states of the respective models.

while being the most versatile at the same time. The alternative parallel engine of Momba demonstrates that parallel state space exploration is well worth it by outperforming all other engines for larger models. Momba's original engine usually requires more memory than the other engines. In contrast, Momba's new alternative engine requires the least amount of memory among all engines.

While being the fastest (almost always on a single core) and most versatile engine, the Modest Toolset's engine, unfortunately, does not provide a public API yet. Thus, if integration into another tool is a concern, Storm and, in particular, Momba have an advantage as they both provide a Python API in addition to APIs in C++ and Rust, respectively.

Limitations of the Evaluation. One of the original motivations of the competition was to also assess the engines with respect to their simulation capabilities for individual traces. However, the performance characteristics displayed here may not carry over to simulation of individual traces as there is a difference between always computing all successor states, as required for exhaustive exploration, and selectively computing only individual successor states which is, for instance, explicitly supported by Momba. Additionally, an exhaustive exploration requires maintaining a (hash) set of all visited states. To facilitate a fair comparison of simulation capabilities, a common simulation API should be developed.

6.4 Discussion

In this chapter, we discussed the architecture and functionality of Momba, a Python framework designed to enhance the accessibility and usability of formal modeling in

general, and the techniques contributed by this thesis in particular.

Momba was initially developed to simplify the creation, validation, and analysis of formal models—activities often perceived as daunting, particularly by those new to the field. By offering a pythonic and tool-agnostic interface, Momba has effectively lowered the barrier to entry. This makes the techniques developed in this thesis more approachable and widely applicable, which is pivotal for translating them into practical, real-world applications. Moreover, this accessibility extends to the powerful techniques developed by the quantitative verification community, which are made available through Momba via JANI and unified interfaces.

Based on the example of Racetrack, we demonstrated how Momba's APIs can be used to programmatically generate a family of models, to empirically validate those models by quickly prototyping an interactive visualization, and to analyze the resulting models for various properties. We furthermore discussed how Momba's model construction capabilities differentiate themselves from other approaches such as text-based preprocessing (recall Section 6.2.1).

Through an empirical evaluation and comparison with well-established state-of-the-art tools, we demonstrated that Momba's state space exploration engine offers competitive features and performance. Furthermore, due to its competitive performance, Momba's state space exploration engine has also been proven effective for training decision-making agents [Gro+22]. The performance of Momba's state space exploration engine will be critical in the following as it ensures that the empirical evaluations of the techniques presented in this thesis are representative and not compromised by inefficiencies in the underlying engine.

Chapter 7

Runtime Verification and Fault Diagnosis

In this chapter, we discuss how the generic framework we presented thus far can be instantiated towards concrete runtime verification and fault diagnosis techniques, enabling novel applications and broadening their area of use. Following our model-based methodology (cf. Chapter 1), the techniques we developed all require a system model and exploit the information regarding a system's behavior it contains.

Traditionally, runtime verification techniques do not take into account a model of the monitored system—they are purely based on a specification of a property. While this is an advantage, in case no accurate model of the system is available, taking a system model into account provides more information which, as we will discuss, can be used to make runtime verification techniques robust against observational imperfections or to produce predictions. Concretely, we discuss how *robust and predictive* automata-based runtime monitors can be obtained with the synthesis pipeline presented in Chapter 4 and how the same pipeline can be used for model-based CTL runtime monitoring. For diagnosis, we discuss how the presented techniques can be used to synthesize traditional model-based diagnosers and how they enable a more flexible diagnosis paradigm based on modal logic. Moreover, we evaluate the continuous time verdictor algorithm for diagnosis on a case study inspired by the industrial automation example presented in Chapter 5.

Chapter Structure. Section 7.1 instantiates the framework towards model-based runtime verification techniques with a particular focus on robust, predictive, and CTL runtime verification. Section 7.2 instantiates the framework towards fault diagnosis techniques enabling greater flexibility in how faults are modeled, how they are queried, and also with a particular focus on robust diagnosis. Section 7.3 presents a

case study around our industrial automation example on which we then evaluate the continuous time verdictor algorithm (recall Chapter 5).

7.1 Model-Based Runtime Verification

To make predictions about inevitable property violations, knowledge about possible future system behaviors is a prerequisite. Furthermore, knowing possible system behaviors is advantageous when accounting for observational imperfections, as it narrows down what could have happened within a system. The techniques developed in Part II of this thesis allow for *model-based runtime verification*, where an operational model of a system in terms of a transition system is assumed to be given and exploited to obtain predictions or account for observational imperfections.

The idea to use information about a system's possible behaviors for runtime verification is not new. Zhang, Leucker, and Dong [ZLD12] use finite *predictive words* to obtain possible continuations towards predictive verdicts regarding LTL properties. Such words can be obtained via static analysis of a monitored program. The work by Pinisetty et al. [Pin+17a] and Ferrando et al. [Fer+21] incorporates assumptions about a system in terms of properties the system fulfills. Cimatti, Tian, and Tonetta [CTT19] leverage *fair Kripke structures* [KPR98], a kind of transition system model, to incorporate assumptions about a system into LTL runtime monitoring. Their approach can deal with partial observability and produce predictions.

While these existing works are specific to runtime verification and truth verdicts, the algorithms contributed by this thesis are generic. We now discuss how they can be used for runtime verification. Due to the generality of the VTS framework, our algorithms are also suitable to be applied on future automata-based monitoring techniques, provided that they give rise to VTSs.

7.1.1 Robust and Predictive Runtime Verification

We have seen that traditional automata-based runtime monitors can be cast into VTSs (recall Section 3.5). To harvest the techniques presented in Chapter 4, we require a VTS that is tight with respect to a given system model. Runtime monitors constructed with traditional techniques are not tight, as they are only based on a property and do not take a system model into account. Using tightening (recall Definition 3.3.7), we can tighten a runtime monitor constructed with third-party techniques for a given system model. The resulting VTS is a runtime monitor that has been *specialized* for a given system model. Using this specialization, it becomes possible to apply the techniques for imperfect observations and predictions developed in Chapter 4 towards robust and predictive runtime monitors. In the following, we exemplify this approach for LTL runtime verification. Traditional LTL runtime verification techniques [BLS06b] assume that events within a system correspond to sets of atomic

propositions and are observed exactly once in the order they occurred. Further, they make no predictions about inevitable property satisfactions or violations based on a system's future behavior that is actually possible.

Example: Coffee Machine. Recall our earlier example of the coffee machine (see Example 2.1). For the model of the coffee machine (see Figure 2.1), assume that we want to monitor for the LTL property $\square\neg$`short_circuit`, i.e., the short circuit should never happen. Synthesizing a monitor for this property over the set

$$\mathsf{AP} = \{\texttt{short\_circuit}\}$$

produces the VTS depicted in Figure 7.1. While the monitor assumes that the atomic proposition `short_circuit` can actually be observed, faults can usually not be directly observed (recall Section 2.6). To account for partial observability of the system, we now aim to use the synthesis techniques developed in Chapter 4. To this end, we first need to tighten the monitor for the given system model.

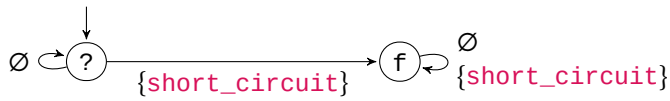


Figure 7.1: LTL runtime monitor for $\square\neg$`short_circuit`.

Recall that tightening requires a shared alphabet of observables between the system model and the VTS (cf. Definition 3.3.7). This can be achieved by just slightly modifying the system model and labeling all transitions which are relevant by sets of atomic propositions. In this case, `short_circuit` is the only atomic proposition, so we obtain the model depicted in Figure 7.2.

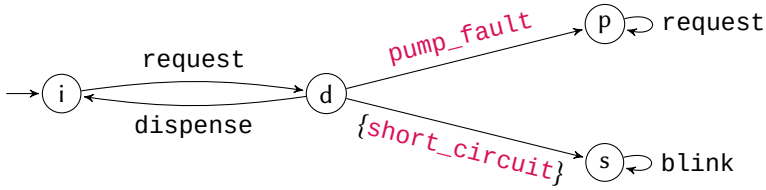


Figure 7.2: System model of the coffee machine adjusted for tightening the LTL runtime monitor for $\square\neg$`short_circuit`.

Tightening the monitor for this model gives us the VTS shown in Figure 7.3 (next page). This VTS is the synchronized product of the original monitor (see Figure 7.1) and the system model (see Figure 7.2). It tracks the state of the system and the state

of the monitor. It produces a verdict based on the monitor under the assumption that all actions of the system model can be observed. As it is tight with respect to the system model, we can now apply the algorithms for predictions and observational imperfections developed in Chapter 4.

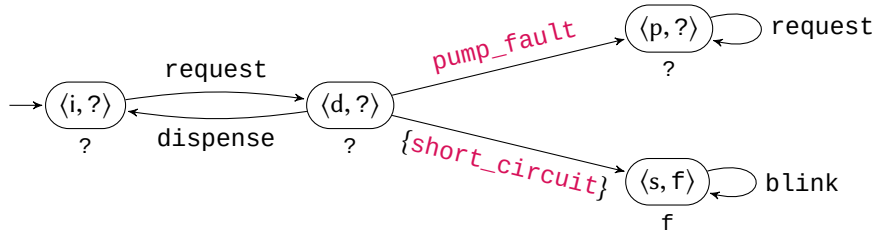


Figure 7.3: Tightening of the LTL runtime monitor for □¬short_circuit for the coffee machine system model as shown in Figure 7.2.

Assume that we want to adjust the VTS for the fact that we cannot directly observe faults, as discussed before. Applying observability projection (recall Section 4.3.1), we obtain the VTS shown in Figure 7.4. It observes the actions of the system and produces f iff short_circuit did occur, despite it being unobservable. That is, we obtained a runtime monitor for the coffee machine that monitors for □¬short_circuit without requiring short_circuit to be directly observable.

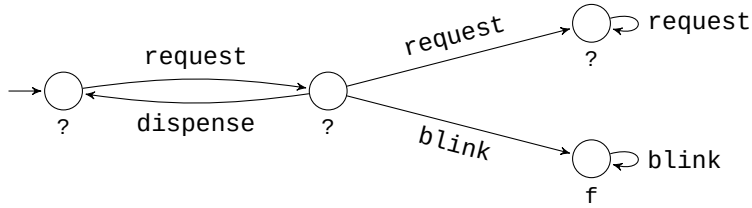


Figure 7.4: VTS for monitoring □¬short_circuit on the coffee machine without being able to observe short_circuit directly.

In this simple illustrative example, applying lookahead refinement (recall Section 4.2) before observability projection (i.e., to the VTS shown in Figure 7.3) would not actually change anything. This is because the system model does not allow the inevitability of the short circuit fault to be observed *before* it occurs, which would be required for predictions. In contrast, consider the alternative system model of the coffee machine shown in Figure 7.5. Here, a short circuit occurs as a result of a pipe bursting under pressure (burst) while the machine is dispensing a coffee. The resulting fluid leakage is then assumed to inevitably cause a short circuit.

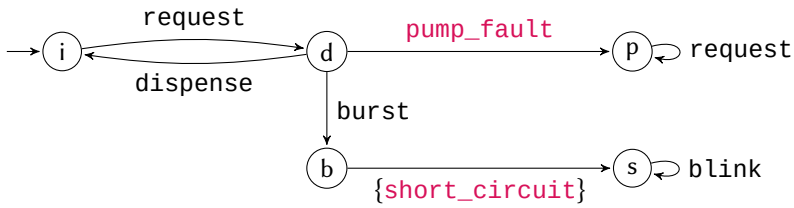


Figure 7.5: Alternative system model of the coffee machine where an inevitable short circuit can be observed before it occurred.

Tightening the monitor for this alternative model gives us the VTS shown in Figure 7.6. Again, the resulting VTS tracks the state of the monitor and the system, producing a verdict based on the monitor's state.

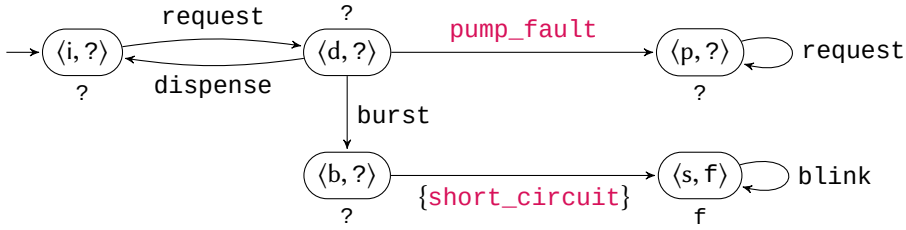


Figure 7.6: Tightening of the LTL runtime monitor for □¬short_circuit for the alternative coffee machine system model as shown in Figure 7.5.

Notably, now the verdict f is inevitable as soon as burst has been observed. Therefore, by applying lookahead refinement, we obtain the VTS shown in Figure 7.7. Here, the verdict f is propagated along monotonic states. As a result, it is produced right after observing burst, as the property will inevitably be violated by the system. Observability projection can then be used as before to account for the faults being unobservable, leading to a VTS for predictions under partial observability.

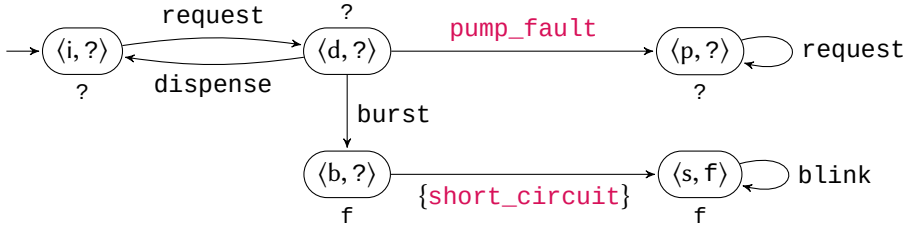


Figure 7.7: VTS obtained by lookahead refinement from the VTS shown in Figure 7.6 producing predictions regarding inevitable short circuits.

As we have seen, by tightening a VTS, in this case for LTL runtime monitoring, the generic techniques developed in Chapter 4 become applicable. They can then be used to make a monitor synthesized with existing techniques robust against observational imperfections, such as limited observability, but also against delays, losses, and out-of-order observations. Furthermore, by applying lookahead refinement, a monitor producing predictions can be obtained.

Note that predictions generated by lookahead refinement are based solely on the information and verdicts encoded in the input VTS. For instance, for the coffee machine the VTS shown in Figure 7.7 produces most specific predictions with respect to the system model (see Figure 7.5) and LTL runtime monitor (see Figure 7.1). It does, however, not produce most specific predictions with respect to the original LTL formula $\square\neg$`short_circuit`. According to the model, a short circuit becomes impossible after the pump broke.[32] Hence, a VTS producing most specific predictions for $\square\neg$`short_circuit` with respect to the system model would in fact produce `t` after the pump broke. The VTS we constructed with lookahead refinement does not do that as the information regarding infinite behavior is already lost when constructing the initial LTL monitor. The initial LTL monitor (see Figure 7.1) has no way of detecting an infinite run without a short circuit and therefore also lacks a state with the `t` verdict. So, while lookahead refinement does produce most specific predictions with respect to its input VTS and is fully generic, techniques specially targeted at LTL [ZLD12; Pin+17a; Fer+21; CTT19] may allow for more specific predictions, as they can take the infinite nature of LTL into account. Nevertheless, runtime monitors taking the infinite nature of systems and properties into account can be constructed with the techniques developed in this thesis. As an instantiation of this idea, we next look at CTL runtime verification.

7.1.2 CTL Runtime Verification

In the literature, runtime verification of LTL has been extensively studied [BLS06b; BLS11; FFM12]. While there is a host of other runtime verification techniques, most of them, including stream-based approaches [e.g. DAn+05; Con+18; Bau+20; GS18], have in common that they rely on a linear notion of time. This is rooted in the fact that, at runtime, a single linear execution is observed. Notable exceptions to this commonality are approaches for monitoring hyperproperties [Fin+19] and past-time CTL [Aud+22]. In case of hyperproperties, the monitor has access to multiple linear executions recorded in the past in addition to the ongoing execution. In case of past-time CTL, the monitor actually uses a tree of events which is based on a distributed or concurrent execution of a system where events cannot be ordered

[32] This is indeed the case for the model of the coffee machine (see Figure 2.1 and Figure 7.5) presented for illustration purposes. Of course, in practice, this is not a realistic assumption.

linearly. Monitoring for branching-time logics has been studied by Francalanza, Aceto, and Ingólfsdóttir for *Hennessy-Milner Logic* with recursion (μHML), a reformulation of the modal μ-calculus [FAI15]. They introduce *MHML* (monitorable HML) as the maximally monitorable subset of μHML under the assumption that only a single execution is given. Perhaps unsurprisingly, this also shows that full branching-time logics are not monitorable when given only a single execution as this would require a computation tree, which cannot be directly observed at runtime. This insight also applies to CTL, which can express non-monitorable properties.

In the following, we will investigate monitoring under the assumption that observations of a single execution and *in addition* a system model is given. We then show how a runtime monitor for a given CTL formula can be constructed with the algorithms presented in Chapter 4. The resulting monitor produces verdicts based on the computation trees implied by the system model. In contrast to traditional runtime verification constructions, the resulting monitors are specific to the given system models and do not rely on observing atomic propositions. As an instance of the VTS Synthesis Problem, we solve the CTL Monitoring Problem:

CTL Monitoring Problem. Given a CTL formula Ψ and a finite model $\mathfrak{S}$ of a system where states are labeled with sets of atomic propositions, synthesize a monitor for Ψ that indicates whether the formula is satisfied or violated based on imperfect observations of a single execution.

The basis for solving the CTL Monitoring Problem will be a VTS constructed with annotation tracking. Recall that the semantics of CTL assigns a set of states $[\![\Psi]\!]_{\text{CTL}}$ to each formula Ψ (Section 2.3.2). Using the CTL semantics, we define the following verdict annotation over the three-valued verdict domain $\mathbb{B}_3$ (recall Figure 3.1b):

$$\kappa(s) \coloneqq ? \qquad \lambda(s) \coloneqq \begin{cases} \texttt{true} & \text{if } s \in [\![\Psi]\!]_{\text{CTL}} \\ \texttt{false} & \text{otherwise} \end{cases} \qquad \gamma(t) \coloneqq ?$$

Now, taking this verdict annotation as a basis, a sound and complete VTS for the verdict oracle as per Definition 4.1.2 and a given observation model produces correct and most specific verdicts regarding the satisfaction or violation of the formula Ψ in the states the system may be in. That is, such a VTS constitutes a CTL runtime monitor and a solution to the CTL Monitoring Problem. Given this verdict annotation, we construct such a VTS with the synthesis pipeline presented in Chapter 4, optionally taking observational imperfections into account and transforming the VTS for predictions. This approach also generalizes to any logic and specification mechanism for which state satisfaction sets can be computed. In particular, it generalizes to arbitrary μ-calculus properties and therefore also to LTL properties.

Note that the verdict oracle for these annotations never assigns the sentinel verdict # to any run. The underlying reason for this is that we label all transitions

with the top element of the verdict domain, ? in this case.

Example 7.1 Take the CTL property $\Psi = \mathsf{EF}\,\mathsf{i}$ from Example 2.4 for the coffee machine as an example. This property is only satisfied in states where there exists a path back to the idle state in which the coffee machine accepts requests and may dispense coffee. If it is not satisfied, then the machine got stuck and will not dispense coffee anymore. A sound and complete VTS for the verdict oracle as per Definition 4.1.2 and the CTL verdict annotation allows detecting violations of the property.

Remark. Runtime verification techniques are often regarded as lightweight alternatives to design time verification where an entire system is verified [LS09]. For such use cases, the presented approach for CTL runtime monitoring is clearly infeasible, as it requires the computation of state satisfaction sets—which amounts to design time verification. The intended use cases are cases where it is known that a property may be satisfied or violated at runtime and one wants to detect *when* that is the case. For instance, we may know that due to physical failures some conditions could be violated and we may want to respond to such cases appropriately.

7.2 Fault Diagnosis

In the discrete-time setting, the contributions of this thesis enable (a) the synthesis of traditional diagnosers, (b) the synthesis of diagnosers for transient faults, (c) the synthesis of diagnosers robust to certain observation imperfections, (d) the synthesis of predictive diagnosers, and (e) the synthesis of diagnosers capable of answering powerful modal logic queries. We now discuss those concrete contributions.

7.2.1 Traditional Diagnosers

Throughout Chapter 4, we have already seen that the modular synthesis building blocks can be combined to synthesize a VTS equivalent to the traditional construction by Sampath et al. [Sam+95] for the coffee machine example. To this end, a TS with fault actions, as described in Section 3.5.1, is first annotated with verdicts from $\langle \wp(\mathcal{F}), \supseteq \rangle$ (e.g., Figure 4.2). Transitions with fault actions f are annotated with singleton sets $\{f\}$ of the respective fault class $f \ni \mathsf{f}$. All other transitions (and states) are annotated with the empty set $\varnothing$ (recall Figure 4.2 for an example). Thus, for each run ρ, the verdict oracle as per Definition 4.1.2 gives us a set of fault classes, corresponding to the faults that occurred on ρ since $\sqcap = \cup$. Using this annotated TS, we obtain a VTS equivalent to a traditional diagnoser by annotation tracking, followed by possibility lifting, observability projection onto OAct, determinization, and then minimization. Here, possibility lifting changes the verdict domain from $\langle \wp(\mathcal{F}), \supseteq \rangle$ to the usual diagnosis domain $\langle \wp(\wp(\mathcal{F})), \subseteq \rangle$. The correctness of the construction follows by combining the theorems of the individual algorithms, as discussed.

Transient Faults. As discussed in Section 4.1.1, fault annotations are more flexible than the fault actions and classes used by the traditional techniques. In particular, such annotations also allow the consideration of *transient faults*, i.e., faults that may later be repaired or go away on their own. We already saw this in Chapter 4. Modeling transient faults is achieved by labeling states with non-empty sets of faults. This has the desired effect because only the verdict from the last state goes into the verdict returned by the oracle (cf. Definition 4.1.2). This demonstrates the flexibility and expressive power of verdict-annotated models.

Predictive and Robust Diagnosis. Traditional diagnosis does not allow predictions. By incorporating lookahead refinement before possibility lifting, in the algorithm described above, we obtain *predictive diagnosers*. A predictive diagnoser indicates inevitable faults no later than the traditional techniques. Furthermore, we can obtain *robust diagnosers* taking into account any combination of limited observability, delays, losses, and out-of-order observations.

7.2.2 Modal Logic Fault Queries

Traditional model-based diagnosis assumes that each transition corresponds to at most one fault class (cf. Section 2.6): Each transition with a fault action has exactly one fault class associated to it. All other transitions have no fault class associated to them. Moving further beyond the traditional construction, we can also annotate transitions and states with Boolean expressions over some set Faults of *basic fault events* (independent of the actions). For instance, $e_1 \vee e_2$ or $e_1 \wedge \neg e_2$ with $e_1, e_2 \in \mathsf{Faults}$. Recall that the usual semantics of Boolean expressions induce a lattice $\langle \wp(\wp(\mathsf{Faults})), \subseteq \rangle$ where each expression ϕ corresponds to a set $[\![\phi]\!]_{\mathbb{B}} \subseteq \wp(\mathsf{Faults})$ of its satisfying assignments. Naturally, a Boolean expression ϕ_1 is more specific than another ϕ_2 iff ϕ_1 implies ϕ_2, which is the case iff all assignments satisfying ϕ_1 also satisfy ϕ_2, i.e., iff $[\![\phi_1]\!]_{\mathbb{B}} \subseteq [\![\phi_2]\!]_{\mathbb{B}}$. Hence, the lattice $\langle \wp(\wp(\mathsf{Faults})), \subseteq \rangle$ is a verdict domain.

By annotating the transitions of a TS with Boolean expressions indicating whether or not they are enabled in the presence of certain combinations of basic fault events and annotating the states of the TS with Boolean expressions indicating whether they can be entered when certain faults are present, we obtain an annotated TS over the verdict domain $\langle \wp(\wp(\mathsf{Faults})), \subseteq \rangle$. Notably, fault trees (cf. Section 2.7) may serve as a basis for such annotations as they are commonly used to model how top-level faults are caused by lower-level faults, and have a natural interpretation as Boolean expressions over a set of basic fault events [RS15]. By applying the instance of the VTS synthesis pipeline with possibility lifting as described above to such a TS, we obtain a diagnoser over the verdict domain $\langle \wp(\wp(\wp(\mathsf{Faults}))), \subseteq \rangle$. Here, each verdict can be interpreted as a set of sets of possible worlds in terms of modal logic [Che80]. Given a verdict $v \in \wp(\wp(\wp(\mathsf{Faults})))$ and a modal logic

formula $\Psi \in \mathrm{MO}[\mathrm{Faults}]$ (cf. Section 2.3.3), we can check whether v satisfies Ψ, i.e., whether $v \subseteq [\![\Psi]\!]_{\mathrm{MO}}$. For instance, it is necessary that fault e_1 or fault e_2 occurred iff $v \subseteq [\![\mathsf{N}(e_1 \vee e_2)]\!]_{\mathrm{MO}}$ and it is possible that fault e_1 occurred and fault e_2 did not occur iff $v \subseteq [\![\mathsf{P}(e_1 \vee \neg e_2)]\!]_{\mathrm{MO}}$. Hence, the diagnoser constructed as described above goes well beyond what is traditionally possible and allows answering powerful modal logic queries. Towards practical implementations, symbolic representations such as BDDs [Bry86] are favorable to succinctly represent sets of faults.

Note that the underlying fault model is different from traditional diagnosis. While traditionally, it is assumed that faults occur when transitions are taken, we now assume that faults may occur at any time and that we can only take certain transitions if they did occur.

Example 7.2 Figure 7.8 shows a system model for the coffee machine (recall Example 2.1) with fault annotations as described above obtained from the fault tree presented as part of Example 2.9. Here, the state i can only be entered when the pump's inlet is not clogged (¬ci), when the pump's shaft is not broken (¬bs), when there are no exposed wires (¬ew), and when there is fluid leakage (¬fl).

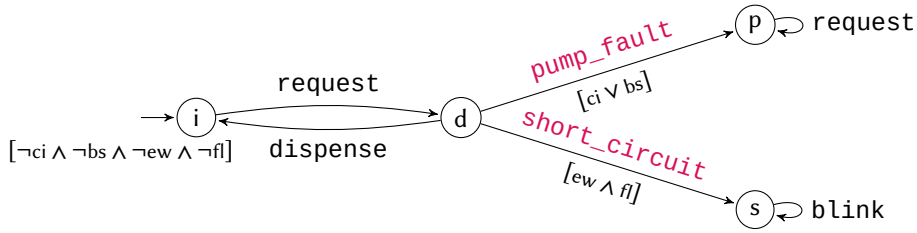


Figure 7.8: System model annotated with Boolean expressions over basic fault events according to the fault tree shown in Figure 2.10.

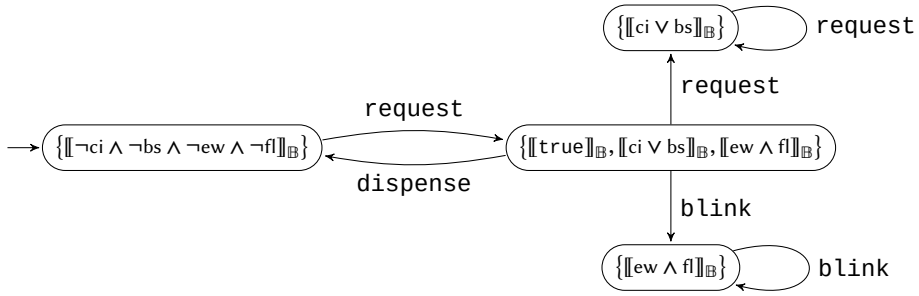


Figure 7.9: Diagnoser synthesized from the model depicted in Figure 7.8.

Applying the synthesis pipeline to this system model, we obtain the VTS shown in Figure 7.9. In the left state, no fault is possible, i.e., we have:

$$\mathsf{N}\,(\neg \mathsf{ci} \wedge \neg \mathsf{bs} \wedge \neg \mathsf{ew} \wedge \neg \mathsf{fl})$$

Note that this is the case due to the state annotation of the initial state of the model. As discussed above, without this annotation, every fault would always be possible. When observing a request, now all faults are possible but no fault necessarily occurred. For the state where we traditionally diagnosed a faulty pump, we now have

$$\mathsf{N}\,(\mathsf{ci} \vee \mathsf{bs})$$

indicating that either the inlet got clogged or the shaft broke. Each of the faults is also individually possible but not necessary. Analogously, for the state where we traditionally diagnosed a short circuit, we now have

$$\mathsf{N}\,(\mathsf{ew} \wedge \mathsf{fl})$$

indicating that wires are exposed and there has been fluid leakage. Here, both faults are also individually necessary.

7.3 Case Study: Robust Real-Time Diagnosis

A completely novel application enabled by the contributions of this thesis is diagnosis of continuous-time systems where observations are subject to timing imprecisions as discussed in Section 5.1. An extension of the traditional model-based diagnosis problem to the real-time setting has been pioneered by Tripakis [Tri02] and later been picked up by Bouyer et al. [BCD05]. However, both works require strict assumptions on the order of events and assume that timing can be assumed with absolute precision (up to discretization). While timing imprecisions are unavoidable for embedded systems consisting of distributed components, they have, to our best knowledge, received no attention in the model-based diagnosis literature so far.

We now study the scalability of the verdictor algorithm presented in Chapter 5 for the diagnosis of faults in real-time systems. Furthermore, we investigate the impact of various timing imprecisions on the quality of the produced verdicts. To this end, we use a case study based on the industrial automation example.

The source code, Momba models, and data used for the experiments presented here is available as part of artifact (AT2).

A Family of System Models. For the case study, we return to our running example (cf. Figure 5.1) of the sorting system. Instead of the illustrative timed automaton shown in Figure 5.2, we devise a family of system models with a configurable conveyor

belt length. These models also explicitly model a discrete position of a single item. We assume that the conveyor moves with a velocity varying between 18 cm s^{-1} and 20 cm s^{-1} which introduces uncertainty with regard to the discrete position of the item at any point in time. As a result, the state space to be considered for diagnosis grows with the length of the belt enabling us to study the impact of its size by varying the length of the belt.[33] The grippers are not modelled explicitly. Whenever an item reaches the end of the conveyor, a new item is assumed to appear at the other end. The two sensors are placed at the center of the conveyor and are 10 cm apart. Both sensors operate at a fixed sampling rate of 10 Hz. Once an item enters the respective field of view, an observable `trigger_`i event is generated where $i \in \{0, 1\}$ is the number of the sensor. Unless explicitly stated otherwise, we assume that these events are observed with a latency between 1 ms and 3 ms. With regard to clock drift we assume $\delta = 0.01$. Hence, the verdict offset as per (5.14) is:

$$\Delta = (1 + 0.01) \cdot (3\,\text{ms} - 1\,\text{ms}) = 2.02\,\text{ms}$$

At any time, a ball bearing fault (`fault_bearing`) may happen which suddenly causes the conveyor to slow down to a varying speed between 13 cm s^{-1} and 16 cm s^{-1}. In addition to the bearing fault, each sensor can fail and begin to trigger spuriously, i.e., despite no item entering its field of view.

Experiment Setup. As part of Momba, we developed an implementation of the continuous time verdictor algorithm presented in Chapter 5 instantiated for diagnosis. The implementation takes as input a specification of the timing imprecisions involved, of the faults and observables, and of the system as a timed automata network in the JANI-model format [Bud+17]. The models for the experiments have been constructed directly with Momba using the approach showcased in Section 6.2.1. While running, the partial order $\prec$ on the observations is kept as a transitivity reduced *Directed Acyclic Graph* (DAG) in which observations are inserted upon arrival. Abstract states are represented using *Difference Bound Matrices* (DBMs) [Dil89; Lar+97]. For representing abstract system states and exploring the zone graph, the state space exploration engine of Momba is used.

The implementation supports generating observations by random simulation of the model together with fault injection capabilities according to a given time-to-fault distribution or after a specific amount of observations have been generated. These observations (or observations made from a real system) are then fed into the verdictor algorithm instantiated for diagnosis. We implemented the optimizations and the over

[33] In practice, a model more akin to Figure 5.2 would be preferable as it leads to a much smaller state space to be considered for diagnosis. However, for the purpose of evaluation, we actually want a model where the amount of diagnosis states to be considered is configurable.

approximation described in Chapter 5 with a configurable history bound. In addition, we use a *Depth-First Search* (DFS) optimized variant of Algorithm 2.

All experiments have been conducted within a Linux VM on an Intel Xeon processor at 2.2 GHz. Each diagnosis job has been allocated a single core and 4 GB of RAM. The implementation itself has not been parallelized, i.e., the reported running times are single core times.

Research Questions. In this setting, we aim to address the following research questions based on the industrial automation case study:

(RDRQ1) How does the approach with a bounded history scale in terms of the model size? (Section 7.3.1)

(RDRQ2) How does the history bound impact the running time of the algorithm? (Section 7.3.2)

(RDRQ3) How does the history bound impact diagnosability? (Section 7.3.2)

(RDRQ4) How do latency and jitter guarantees impact the running time of the algorithm? (Section 7.3.3)

(RDRQ5) How do latency and jitter guarantees impact diagnosability? (Section 7.3.3)

7.3.1 Scalability of the Verdictor Algorithm

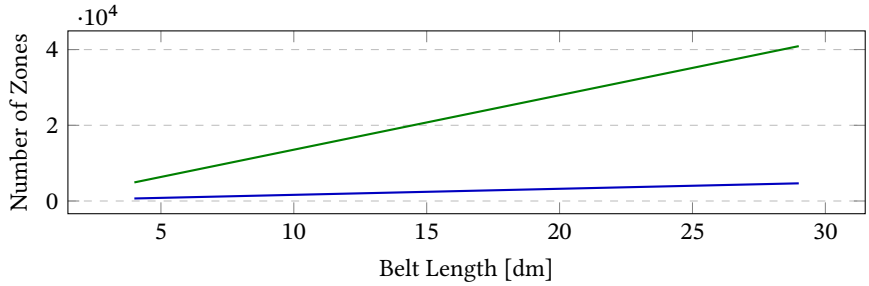


Figure 7.10: Size of the zone graph with bearing and sporadic sensor faults (——) and with bearing but without sporadic sensor faults (——) as a function of the conveyor belt length in decimeter (dm).

Figure 7.10 shows the number of zones in the zone graphs of our models as a function of the length of the conveyor belt with and without sporadic sensor faults enabled for the two sensors. As depicted, the size of the zone graphs of the models scales

linearly with the length of the conveyor belt independent on whether there may be sporadic sensor faults. As expected, models with sporadic sensor faults enabled have significantly more zones, since sensors may trigger spuriously at any time.

(RDRQ1): *How does the approach with a bounded history scale in terms of the model size?* To assess the scalability of the approach, we randomly simulated 1 000 runs[34] of 120 s simulation time for each configuration injecting a ball bearing fault after an average of 200 s, determined according to an exponential distribution. As a result, roughly 45 % of all runs contain a bearing fault, resulting in a roughly equal distribution of runs with and without faults. Of course, the fault rate of the ball bearings will be much lower in practice. For each run, we then randomly generated a set of observations by applying a varying latency between 1 ms and 3 ms and a clock drift sampled uniformly at random. We then fed those observations according to their observation time to the verdictor algorithm for diagnosis and recorded the wall-clock time required to process each observation.

Note that within the 120 s simulation time, more observations will be made on the shorter belts than on the longer ones as an item needs less time to run through and thus more items are considered. Within 120 s eight items can be expected to run by the sensors on the largest belt (2.9 m with a speed of approximately 20 cm s^{-1}) yielding 16 observations. For the shortest belt (4 dm), however, 60 items can be expected to run by the sensor yielding 120 observations. For comparison, we consider the average time in seconds of wall-clock time required to process a single observation.

Figure 7.11 shows the results of this experiment as a function of conveyor belt length with a history bound of $B_H = 2$ which is, as we will see in Section 7.3.2, a good choice for the given model family. As expected, the time to process a single observation grows non-linearly (according to the theory the growth should be polynomial, see Section 5.4.1) in the size of the original model which grows linearly with the belt length (cf. Figure 7.10). Without the runs on which faults occur (solid lines), the time required to process a single observation increases. This effect is explained as follows: Whenever a fault occurs and has been diagnosed, the verdictor algorithm can ignore those parts of the state space where the fault has not occurred, reducing the number of verdict states and thus the running time of processing observations made after the fault has been diagnosed.

Answering (RDRQ1), the empirical results with a fixed history bound reconfirm our theoretical findings. The running time of the approach scales polynomially with the size of the model, while the time required to process a single observation is not affected by the number of observations. In absolute terms, the running times for the model family without sporadic sensor faults could be acceptable for real-time applications. To keep up, the algorithm has one second to process an observation for

[34] See Figure 7.12 for the variance of the observation processing times in a similar setting. Simulating 1000 runs provided a good tradeoff between noise and performance of conducting the experiments.

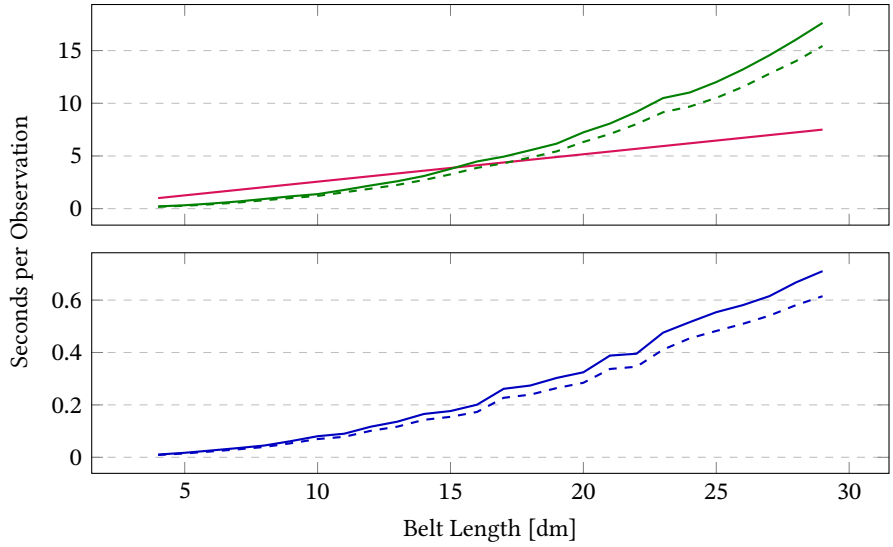


Figure 7.11: Average time in seconds of wall-clock time required to process a single observation with bearing and sporadic sensor faults (——, top) and with bearing but without sporadic sensor faults (——, bottom) as a function of the conveyor belt length in decimeter. The dashed lines are with and the solid lines are without runs containing faults. For the case with sporadic sensor faults (top), the red line (——) indicates the time available to process a single observation to keep up with the rate of observations.

the shortest belt and 7.5 s for the longest belt. For the case without sporadic sensor faults, the time required to process a single observation is below the respective thresholds for all conveyor belt lengths we consider here. In contrast, the model with sporadic sensor faults has a much larger state space (cf. Figure 7.10). For this model, at around a belt length of 15 dm, the algorithm is no longer able to keep up with the rate of observations (red line in Figure 7.11). Indeed, multiple seconds per observation is likely too much for any practical application. It should be noted, however, that the case study with sporadic sensor faults has been designed to be particularly challenging for the algorithm as the complete state space has to be considered at each point in time due to the unreliable nature of the sensors.

7.3.2 Impact of the History Bound

For the previous experiment, we chose a history bound of $B_H = 2$. We now investigate the impact of this choice both, on the running time of the algorithm as well as on the quality of the produced verdicts.

(RDRQ2): *How does the history bound impact the running time of the algorithm?* We have already established that by considering only a bounded history (a) the space and time requirements of the algorithm become bounded by the size of the system model, and (b) this bound is polynomial (with the polynomial rank being the history bound) in the original size of the model. To further study these effects, we consider a model with a conveyor belt length of 5 dm without sporadic sensor faults. As before, we randomly generated 1 000 runs and corresponding sets of observations by simulation and fed those observations to the verdictor algorithm.

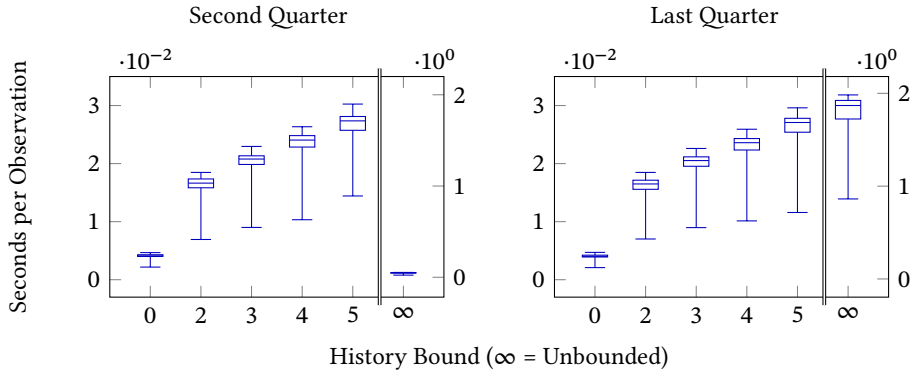


Figure 7.12: Time in seconds of wall-clock time required to process a single observation for the second (left) and last (right) quarter of observations with different history bounds and without any history bound (indicated by ∞). The scale on the y-axis with and without history bound differs by two orders of magnitude. In the case of no history bound, the median is 0.05 s for the second quarter (left) and 1.87 s for the last quarter (right). With a history bound of $B = 5$, the median is 0.027 s for both the second quarter (left) and the last quarter (right) showing the huge impact of the history bound on the running times.

Figure 7.12 shows how much time it takes to process a single observation for the second (left) and last (right) quarter of observations with different history bounds $B_H \in \{0, 2, 3, 4, 5\}$ and without any history bound (indicated by ∞). Considering the second and last quarter allows us to see differences based on the amount of observations already processed. As one can see, there is no significant difference between the second and last quarter when the history is bounded because the space and time requirements do not grow with the number of observations in this case. However, as expected, in the unbounded case, there is a significant increase between the second (left) and last (right) quarter of observations because ever more clocks and constraints are introduced. This effect can also be seen when comparing the number of diagnosis states required and is explained by the introduction of ever more clocks causing the running time to grow exponentially.

Note that the scale on the y-axis with and without history bound differs by two orders of magnitude. It is clearly evident that a history bound significantly reduces the cost. The huge jump from 0.05 s for the second (left) to 1.87 s for the last (right) quarter without any history bound is explained by the exponential nature of the underlying reachability problem—the cost grows exponentially with the number of clocks, i.e., the number of observations.

The boxplots have been chosen to also display the variance of observation processing times. The boxes represent 50% of all values with the median being marked within the box. The whiskers represent the extrem values.

(RDRQ3): *How does the history bound impact diagnosability?* As we have seen, a lower history bound allows processing much more observations per second. However, this comes at the expense of imprecision of the analysis. We now discuss a number of high-level insights about those imprecisions gained from individual experiments with our models.

For our example, it turns out that a ball bearing fault becomes *undiagnosable* with a history bound of $B_H = 0$, i.e., it is impossible to diagnose this fault on runs where it occurred. Intuitively, a bearing fault surfaces only in the timing difference of consecutive `trigger_i` events. However, with a history bound of 0, the precise timing of observations is ignored and only the possible orderings of their corresponding events can serve as basis for diagnosis. This makes diagnosis impossible for $B_H = 0$ and this insight generalizes to other models and history bounds. In general, if the timing between any two observations is deemed decisive and there may be at most N observations between them, then a history bound of $N + 2$ is enough. Indeed, assuming that the sensors are not triggering spuriously, i.e., there are no other observations between two consecutive trigger observations, a bearing fault is always diagnosable with $B_H = 2$. This is the reason why we have chosen $B_H = 2$ for our scalability study.

In contrast, sporadic sensor faults always surface (with the given timing imprecisions) in the order and actions of observations, if they surface at all. As the sensors trigger only when an item enters their field of view, as long as there is no fault, it is, for instance, impossible that the second sensor triggers twice without the first sensor triggering in-between. Hence, this fault mode can *oftentimes* be diagnosed even with a history bound of $B_H = 0$. Interestingly, sporadic sensor faults are not always diagnosable even with an unbounded history. This is because, although highly unlikely in practice, a faulty sensor's spurious triggering may, by pure chance, coincide with the time of an item entering its field of view. Note that this does not contradict Δ-completeness of the algorithm because Δ-completeness is about diagnosing a fault with a worst-case delay of Δ if it surfaces in the observations. The latter is not the case for such pathologic runs. The observable behavior of a faulty sensor that notoriously triggers whenever a part enters its field of view is undistinguishable

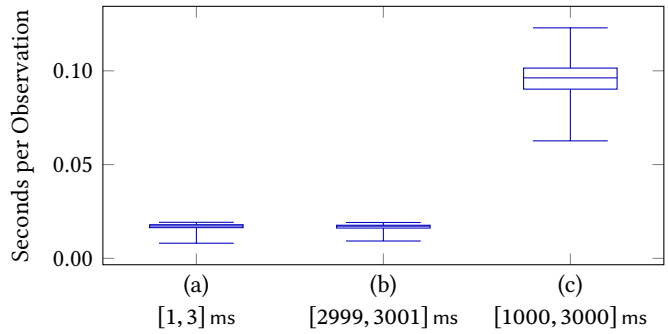


Figure 7.13: Time in seconds of wall-clock time required to process a single observation with a latency varying between (a) 1 ms and 3 ms, (b) 2999 ms and 3001 ms, and (c) 1000 ms and 3000 ms.

from the observable behavior of a correctly functioning sensor. Hence, any diagnosis technique based on passive observation cannot detect such faults.

With a history bound of $B_H = 2$ it is possible to diagnose sporadic faults earlier. Consider a case where the first sensor is faulty and produces a sporadic `trigger_0` event right after a `trigger_1` event. Now, if we do not take the timing into account, we have to wait until the next item enters the field of view of the first sensor leading to yet another `trigger_0` event thereby enabling a diagnosis with a history bound of 0. However, with a history bound of 2, we will be able to diagnose the fault right when we observe the first `trigger_0` event as the corresponding observation is made too early in relation to the observation of the `trigger_1` event.

These considerations exemplify the over-approximative nature of the history bound. Considering only a bounded history may (a) render certain faults undiagnosable and (b) prolong the time until a fault is diagnosed. This tradeoff needs further investigation in future work. In practice, it is of course desirable, to always be able to diagnose faults after a bounded amount of time. Ideally, with an analysis of the model and timing imprecisions, one should be able to determine whether a fault can always be diagnosed *in time* when using a specific history bound. We consider this question of in-time diagnosability with a history bound an important focus for future work.

7.3.3 Impact of Latency and Jitter

So far our focus has been on a realistic varying latency between 1 ms and 3 ms. We now study the effect of different latency guarantees for our example.

(RDRQ4): *How do latency guarantees impact the running time of the algorithm?* To empirically study the effect of different latency guarantees, we generated 1 000 sets of observations as described in Section 7.3.2 for each of three different latency

configurations. Figure 7.13 shows how much time it takes to process a single observation for the different latency configurations (a) with a latency between 1 ms and 3 ms, (b) with a latency between 2999 ms and 3001 ms, and (c) with a latency between 1000 ms and 3000 ms. These experiments demonstrate that high latency (Figure 7.13, b) per se is no problem and mostly pertains the verdict offset Δ, i.e., the time it takes until an observation becomes settled and thus has to be taken into account by the verdictor algorithm. High jitter, i.e., variation in latency, however, has the effect that the verdictor algorithm needs to consider many different orderings of observations (recall Section 5.2.3) thereby blowing up its running time. This becomes evident when comparing (a) and (b) with (c) in Figure 7.13. Here, (c) has a very high variation of 2000 ms in contrast to (a) and (b) where the variation is only 2 ms.

(RDRQ5): *How do latency and jitter guarantees impact diagnosability?* If the shared network is underdimensioned[35] and thus induces a highly varying latency between 1000 s and 3000 ms (Figure 7.13, c), faults of the ball bearings become undiagnosable because the timing difference between two consecutive `trigger_i` events is blanketed by the timing imprecisions. In contrast, a mere high latency between 2999 ms and 3001 ms (Figure 7.13, b) still allows diagnosing such faults. Future work should tackle how to (i) verify that a fault is diagnosable given some timing imprecisions and (ii) obtain upper bounds for the allowed latency variation.

7.4 Discussion

In this chapter, we explored various concrete applications instantiating the generic verdictor algorithms developed earlier. We discussed how the contributions advance the state of the art in runtime verification and fault diagnosis, focusing on robustness and predictive capabilities enabled by the techniques developed in this thesis.

In addition to the related work discussed in Section 7.1, robustness and predictions have also been studied from a stream-based runtime verification perspective. Leucker et al. utilize abstraction techniques to deal with partial information [Leu+19]. They integrated and evaluated their techniques within the stream-based runtime verification framework TeSSLa [Leu+18]. Kallwies, Leucker, and Sánchez utilize symbolic techniques to deal with uncertainties and assumptions about a system [KLS22]. This approach has been evaluated on Lola using exhaust emission monitoring data and a Lola specification originally developed by the author of this thesis [KHB18]. Again, we view our contributions as complementary to these works. They provide robustness and predictions based on a formal system model.

[35] For instance, in the case of CAN, insufficient bandwidth may lead to congestion resulting in a high (variation in) latency for low-priority messages which may be non-critical for the functioning of the system but highly indicative of faults.

Through a case study centered around the industrial automation example, we demonstrated that the verdictor algorithm for the continuous-time case scales effectively and we investigated the impact of timing imprecisions and a history bound on the verdicts it produces. Our experiments demonstrated that for our case study, an appropriate history bound can be chosen such that faults remain diagnosable. The experiments also revealed an interesting tradeoff between the delay with which a fault is detected and the history bound used, the latter translating directly into the information the diagnoser needs to keep track of and the time required to account for new observations. A particular focus of future work should be on the investigation of in-time diagnosability with a bounded history. It should investigate how to determine *best* and *worst* case delays for diagnosing a given fault in the presence of a bounded history while also considering more advanced and effective strategies of dropping observations, instead of just cutting the history (recall Section 5.4.1). Combining such an analysis with the presented techniques helps to ensure that faults are always diagnosed timely. To this end, the theory and algorithm put forward in this thesis unlock a rigorous treatment and understanding of the tradeoffs, by establishing the needed formal grounds. Use cases beyond diagnosis, as they are enabled by the generic treatment in this thesis, may also be a particular focus of future work.

The applications presented here leverage the developed techniques for answering operational questions regarding the satisfaction and violation of properties (Q1) as well as the presence of faults (Q2). The remaining questions regarding possible systems configurations (Q3), will be addressed next.

Chapter 8

Variability-Aware Monitoring

This chapter showcases what the contributions of this thesis enable beyond the existing application areas of runtime verification and fault diagnosis. With *variability-aware monitoring* we introduce an entirely novel application domain.

Most modern systems are highly configurable based on customer needs or through their inherent adaptivity to the environments in which they operate. For instance, cars may come with a diverse set of driver assistance or infotainment systems, depending on what the customer payed for, or robots may adapt their behaviors depending on whether they operate in a machine-only or human-machine co-adaptive setting. Within software systems we also encounter configurability, e.g., in *software product lines* that can be configured through *features* as incremental or optional functionalities [Zav00; Ape+13]. Often the configuration spaces are exponentially large in the number of configuration options or features, which renders the development and analysis of such systems particularly challenging.

The manifold possibilities to configure such systems also raise further challenges in the runtime monitoring setting that have been barely addressed in the existing literature. Tackling this gap, we identify two challenges towards variability-aware monitoring and show how they can be addressed by exploiting and expanding upon the techniques developed in the preceding chapters.

The Configurable Monitors Challenge. As modern systems are highly configurable, runtime monitors may need to adapt to them. For instance, depending on a system's configuration different properties may be of interest or must be monitored based on different data. To accommodate this variability in a system being monitored, monitors must be configurable themselves. In the context of configurable systems, naive solutions typically suffer from an exponential blowup of configurations in the number of features. So, the first challenge is to develop techniques for the effective specification, synthesis, and implementation of *configurable monitors* that do not

incur an exponential blowup in the number of features. In particular, considering separate monitors for each configuration individually will typically not be feasible. Notably, configurable monitors have applications beyond configurable systems and can also be used in cases where the properties of interest may depend on other external factors, e.g., different use cases of a monitor.

The Configuration Monitoring Challenge. While a system's configuration may be known at time of deployment, this can no longer be assumed at runtime where configurations often are not readily exposed [AFW18]. For example, configurations of legacy or physical components might be unknown to the running system—imagine a factory worker who physically configures a machine (cf. Chapter 1). Furthermore, configurations may be disguised to enhance security and privacy [CCM08], or they may change after deployment, rendering them unknown at runtime. Yet, knowing a system's configuration at runtime is often beneficial, e.g., to check whether a system is correctly configured for the next step of a production process or to configure external monitoring and diagnostics equipment [Kim+10], like a configurable monitor. Also to detect configuration vulnerabilities [RS02], to determine information leakages [PSJ18], or to reason about possible configuration-based attacks that compromise system security, information about the system configuration is very valuable. Therefore, the second challenge we address is determining an a-priori unknown configuration of a system at runtime solely by observing its behavior.

Overview. To address the first challenge, we introduce a featured variant of verdict transition systems which can serve as a the basis for configurable monitors and their synthesis. We show how such featured VTSs can be synthesized from properties specified in a featured variant of linear temporal logic (LTL) [Cla+13]. Furthermore, we introduce a configurable variant of the stream-based specification language Lola [DAn+05] together with a family-based analysis for well-formedness and efficient monitorability that exploits commonalities across specification variants.

To address the second challenge, we instantiate the generic VTS synthesis pipeline developed in Chapter 4 for the synthesis of *configuration monitors* from FTS system models. The resulting monitors produce most specific configuration verdicts, i.e., sets of configurations the system may be in according to the observations it generated (cf. Section 3.1). We validate this approach through an empirical evaluation on established configurable systems community benchmarks. This evaluation will also demonstrate the effectiveness of the VTS synthesis pipeline.

Relevant Publications. The two challenges surrounding variability-aware runtime monitoring and the techniques for configurable monitors, which we present in the following, have first been introduced in:

> [DK22]: Clemens Dubslaff and Maximilian A. Köhl. "Configurable-by-Construction Runtime Monitoring". In: *Leveraging Applications of Formal Methods, Verification and Validation, ISoLA 2022.*

Configuration monitoring has been introduced in:

> [KDH24]: Maximilian A. Köhl, Clemens Dubslaff, and Holger Hermanns. "Configuration Monitor Synthesis". In: *Automated Technology for Verification and Analysis, ATVA 2024.*

As a running example, we will use runtime monitoring of real driving emissions, which has first been introduced in:

> [KHB18]: Maximilian A. Köhl, Holger Hermanns, and Sebastian Biewer. "Efficient Monitoring of Real Driving Emissions". In: *Runtime Verification, RV 2018.*

With involvement of the author of this thesis, the work on monitoring real driving emissions has been expanded in subsequent work [Her+18; Bie+21; Bie+23].

Exceeding these published works, the featured variant of VTSs is an unpublished contribution enabled by the theoretical framework presented in Chapter 3. Featured VTSs generalize configurable LTL_3 monitors as they have been introduced in the original work on configurable monitors [DK22].

Chapter Structure. Section 8.1 introduces monitoring of real driving emissions as our running example. Section 8.2 approaches the challenge of configurable monitors and their specification from the automata-based perspective, introducing a featured variant of VTSs and a synthesis approach from featured LTL properties. Section 8.3 presents a configurable variant of the stream-based specification language Lola together with family-based analysis techniques for well-formedness and efficient monitorability. Section 8.4 tackles the problem of determining a system's configuration solely by observing its behavior. To this end, we harvest the generic synthesis techniques developed earlier in Chapter 4.

8.1 Example: Real Driving Emissions

In response to the massive revelation of fraudulent behavior programmed inside diesel cars across Europe in 2015, the European Union has defined a procedure to test for *Real Driving Emissions* (RDE) [Eur16; Eur17]. This procedure has been gradually put into force since September 2017. An RDE test is meant to evaluate the emissions of a vehicle under realistic conditions.[36] To this end, the RDE regulation comes with

[36] Similar test procedures exist for characteristics of electric vehicles, for instance, those issued by the United States Environmental Protection Agency [Uni].

an informal but relatively precise specification that spells out in how far a real trip, i.e., a trajectory driven with a car on public roads, constitutes a valid RDE test, or not. The regulation then stipulates emission limits for such valid RDE tests. The author of this thesis developed a formalization of the RDE test procedure in the stream-based specification language Lola [KHB18]. As part of this work, a low-cost variant of the RDE test procedure was developed, which can be conducted without expensive test equipment but solely with on-board sensors.

In subsequent work, the low-cost variant of the RDE test procedure has been used to gather evidence that the exhaust emission cleaning system of an Audi A7 indeed behaves differently under certain conditions that do not occur during traditional exhaust emission tests [Her+18]. Furthermore, the original Lola specification has been translated to RTLola, an extension of Lola for real-time systems [Fay+19; Bau+20], for usage in an Android application enabling laypersons to conduct exhaust emission tests [Bie+21; Bie+23].

In the following, we use the low-cost variant of the RDE test procedure as an example for variability-aware runtime monitoring. To this end, we focus on certain parts of the Lola specification of the RDE test procedure, which have been streamlined for presentation purposes as part of the work on configurable-by-construction runtime monitoring [DK22]. For the full details regarding RDE testing with Lola and RTLola, we refer to the original papers [KHB18; Her+18; Bie+21; Bie+23].

Low-Cost RDE Testing. At the core of low-cost RDE testing is the idea to use the on-board sensors of a car to compute exhaust emissions and other values relevant for RDE testing. The types of on-board sensors as well as their values can be queried via the standardized *On-Board Diagnostic* (OBD) interface. By law, any modern car is required to be equipped with an OBD interface [Eur98]. In contrast, for actual RDE tests, an expensive *Portable Emissions Measurement System* (PEMS) is required and must be attached to the exhaust pipe of the car while driving a test.

As different cars come with different on-board sensors, the concrete instantiation of their OBD interface differs. Hence, a runtime monitor for RDE testing must be configurable with respect to the sensors of each particular car. For example, to compute the amount of emitted pollutants, such as nitrous oxide (NOx), the *exhaust mass flow* (EMF), i.e., the mass of exhaust emitted per time unit, and the relative concentration in parts per million (ppm) of the pollutant in the exhaust gas must be known. While many modern diesel cars come equipped with sensors providing the relative concentration of nitrous oxide, the EMF is rarely provided directly via OBD due to the car not having an EMF sensor. Fortunately, the EMF can be computed based on various other values such as the *mass air flow* (MAF) in combination with the fuel rate (FR) or the *fuel-air equivalence* (FAE) ratio. So, depending on the on-board sensors of the concrete car in question and the values it exposes via OBD, the Lola specification for the low-cost RDE test must be adapted to that car following its

configuration. This is a prime practical example of the need for configurability in runtime monitoring.

Low-Cost RDE Variability Modeling. Using feature diagrams (recall Section 2.4), we can model the different configurations for a low-cost RDE test of the NOx emissions of a car. To conduct such a test, the car needs to be equipped with an NOx sensor and there must be a way to obtain the EMF. The EMF can be obtained via an EMF sensor (EMFs) or via computation (EMFc) based on other values. To compute the EMF, an MAF sensor and an FR or FAE sensor are required. Hence, the feature diagram depicted in Figure 8.1 describes all the valid sensor combinations that allow us to conduct a low-cost RDE test of the NOx emissions of a car.

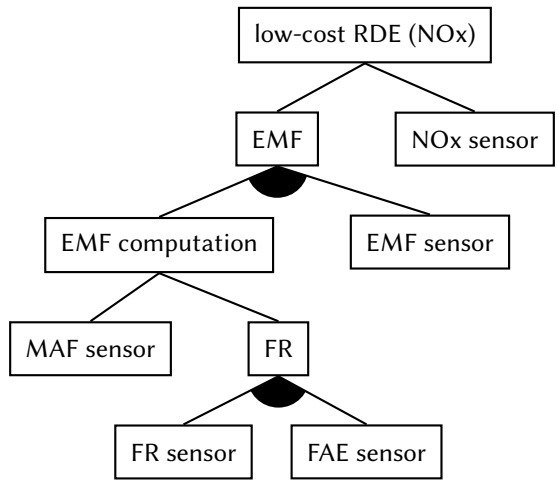


Figure 8.1: Feature diagram describing the different sensor configurations that allow us to conduct a low-cost RDE test of the NOx emissions.

8.2 Configurable LTL$_3$ Monitoring

Tackling the configurable monitors challenge, we now introduce a featured variant of VTSs. Similar to featured transition systems (recall Section 2.4), featured VTSs can avoid an exponential blowup by succinctly representing monitor families.[37] After introducing featured VTSs, we present a product construction which can be leveraged for the effective compositional synthesis of featured VTSs. As an explicit instance, we develop a synthesis technique for featured LTL specifications. To this end, we

[37] More generally, they represent verdictor families but our focus here is on monitoring.

utilize featured VTSs as a target representation and employ the product construction to combine monitors for individual featured LTL formulas that have been derived with LTL_3 runtime monitoring techniques [BLS06b].

8.2.1 Featured VTSs

We introduce a featured variant of VTSs combining the orthogonal ideas of FTSs and VTSs. The resulting *featured verdict transition systems* (FVTSs) represent families of verdict transition systems. Analogously to how TSs are obtained from FTSs, VTSs are obtained from FVTSs by instantiating them for a given configuration.

Definition 8.2.1 *Given a set of observables* Obs *and a set of features* F*, a* featured verdict transition system *(FVTS)* $\mathcal{F}$ *is a tuple*

$$\langle Q, J, \mathrm{Obs}, \rightarrow, \mathcal{V}, \sqsubseteq, \nu, F, \mathrm{Conf}, \mathrm{g}, \iota \rangle$$

where
- $\langle Q, J, \mathrm{Obs}, \rightarrow \rangle$ *is a TS,*
- $\langle \mathcal{V}, \sqsubseteq, \nu \rangle$ *is a verdict extension (recall Section 3.2), and*
- $\langle F, \mathrm{Conf}, \mathrm{g}, \iota \rangle$ *is a feature extension (recall Section 2.4).*

We define the projection of an FVTS $\mathcal{F}$ onto a configuration c analogously to the projection for FTSs (recall Definition 2.4.2):

Definition 8.2.2 *For a given FVTS*

$$\mathcal{F} = \langle Q, J, \mathrm{Obs}, \rightarrow, \mathcal{V}, \sqsubseteq, \nu, F, \mathrm{Conf}, \mathrm{g}, \iota \rangle$$

and valid configuration $c \in \mathrm{Conf}$*, the* c-projection *of* $\mathcal{F}$*, denoted by* $\mathcal{F}{\downarrow}c$ *is a VTS*

$$\langle Q, J{\downarrow}c, \mathrm{Obs}, \rightarrow{\downarrow}c, \mathcal{V}, \sqsubseteq, \nu \rangle$$

where $J{\downarrow}c := \{ s \in J \mid c \in [\![\iota(s)]\!]_{\mathbb{B}} \}$ *and* $\rightarrow{\downarrow}c := \{ \mathrm{t} \in \rightarrow \mid c \in [\![\mathrm{g}(\mathrm{t})]\!]_{\mathbb{B}} \}$.

So, for each configuration $c \in \mathrm{Conf}$ of an FVTS $\mathcal{F}$, we obtain a VTS $\mathcal{F}{\downarrow}c$. We also refer to the projection $\mathcal{F}{\downarrow}c$ as the *instantiation* of $\mathcal{F}$ for c.

An FVTS can succinctly represent a family of verdictors exploiting structural sharing between the different configurations. As such, they are a foundational building block for addressing the configurable monitors challenge: By exploiting structural sharing, they can avoid the combinatorial blowup that would otherwise result from considering each configuration as a VTS individually. When defined over a truth domain for monitoring, e.g., the three valued truth domain $\mathbb{B}_3$, an FVTS represents a configurable monitor that can be instantiated for its configurations.

Composition. For the synthesis of FVTSs from featured specifications, we will exploit the synchronized product of multiple FVTSs, corresponding to a concurrent lockstep execution of those FVTSs. The result is an FVTS over the product verdict domain of the verdict domains of the individual FVTSs. We define the composition for a pair $\langle \mathcal{F}_1, \mathcal{F}_2 \rangle$ of FVTSs as follows:

Definition 8.2.3 *Given a pair $\langle \mathcal{F}_1, \mathcal{F}_2 \rangle$ of FVTSs*

$$\mathcal{F}_i = \langle Q_i, J_i, \mathsf{Obs}, \rightarrow_i, \mathcal{V}_i, \sqsubseteq_i, \nu_i, F, \mathsf{Conf}, g_i, \iota_i \rangle$$

with the same feature domain F, valid configurations Conf*, and observables* Obs*, we define the composition of $\mathcal{F}_1$ and $\mathcal{F}_2$, denoted by $\mathcal{F}_1 \parallel \mathcal{F}_2$, as*

$$\mathcal{F}_1 \parallel \mathcal{F}_2 = \langle Q_1 \times Q_2, J_1 \times J_2, \mathsf{Obs}, \rightarrow, \mathcal{V}_1 \times \mathcal{V}_2, \sqsubseteq_1 \times \sqsubseteq_2, \nu, F, \mathsf{Conf}, g, \iota \rangle$$

such that $\rightarrow$ is the smallest relation satisfying

$$\frac{\langle q_1, \mathsf{o}, q_1' \rangle \in \rightarrow_1 \qquad \langle q_2, \mathsf{o}, q_2' \rangle \in \rightarrow_2 \qquad \mathsf{o} \in \mathsf{Obs}}{\langle \langle q_1, q_2 \rangle, \mathsf{o}, \langle q_1', q_2' \rangle \rangle \in \rightarrow}$$

and with ν, g, and ι being defined as follows

$$\begin{aligned} \nu(\langle q_1, q_2 \rangle) &:= \langle \nu_1(q_1), \nu_2(q_2) \rangle \\ g(\langle \langle q_1, q_2 \rangle, \mathsf{o}, \langle q_1', q_2' \rangle \rangle) &:= \chi\big(g_1(\langle q_1, \mathsf{o}, q_1' \rangle) \wedge g_2(\langle q_2, \mathsf{o}, q_2' \rangle)\big) \\ \iota(\langle q_1, q_2 \rangle) &:= \chi\big(\iota_1(q_1) \wedge \iota_2(q_2)\big) \end{aligned}$$

where $\chi(\phi)$ denotes a canonical characteristic representation of the Boolean expression ϕ, e.g., in disjunctive normal form.

Remark. A similar product construction can be defined for VTSs corresponding to their concurrent lockstep execution and producing verdicts of the respective product verdict domain.

8.2.2 Featured LTL$_3$ Monitoring

To specify variability-aware properties, e.g., for FTSs, *featured linear temporal logic* (FLTL) has been introduced by Classen et al. [Cla+13] as a featured extension of LTL. Formulas in FLTL over a set of features F and of atomic propositions AP are of the form $[\phi]\ \varphi$ where $\phi \in \mathbb{B}[F]$ is a Boolean expression over F, referred to as *guard*, and φ is an LTL formula over AP. For a non-empty set Φ of FLTL formulas and a configuration $c \subseteq F$, the *c-projection* of Φ is the conjunction of all the formulas of Φ whose guard is satisfied by c. Formally, we define the c-projection of Φ, as follows:

$$\Phi\!\downarrow_c \;:=\; \bigwedge \{\, \varphi \mid [\phi]\ \varphi \in \Phi \text{ s.t. } c \in [\![\phi]\!]_{\mathbb{B}} \,\} \tag{8.1}$$

We aim to synthesize a configurable monitor from a set of FLTL formulas such that this monitor can be instantiated to monitor for the formula corresponding to each configuration, respectively. Formally, we aim to solve the following problem:

Featured LTL_3 Monitor Synthesis Problem. Given a non-empty set Φ of FLTL formulas, synthesize an FVTS $\mathcal{F}$ such that $\mathcal{F}\downarrow c$ is an LTL_3 monitor for $\Phi\downarrow c$ for each configuration $c \subseteq F$, respectively.

Example 8.1 Let us specify FLTL properties for the RDE example (recall Section 8.1). In case the EMF sensor is present (EMFs), we want the EMF to be measured directly instead of computing it (*comp-EMF*) from other values such as the MAF and fuel sensors. This requirement can be expressed by an FLTL formula:

$$\psi_0 \quad = \quad [\mathsf{EMFs}]\,\Box\neg \mathit{comp\text{-}EMF}$$

Another example would be the property that if the EMF computation feature (EMFc) is enabled and no EMF sensor is present, then after the MAF sensor has been read (*read-MAF-sensor*), in the next step the EMF must be computed. This is expressed by the following FLTL formula:

$$\psi_1 \quad = \quad [\mathsf{EMFc} \wedge \neg\mathsf{EMFs}]\,\Box(\mathit{read\text{-}MAF\text{-}sensor} \rightarrow \bigcirc \mathit{comp\text{-}EMF})$$

Given the set $\Phi = \{\psi_0, \psi_1\}$, we aim to construct a matching configurable LTL_3 monitor.

Synthesizing Featured LTL_3 Monitors. To synthesize a featured LTL_3 monitor for a non-empty set Φ of FLTL formulas, we first construct an LTL_3 monitor $\mathcal{M}_{\mathrm{LTL}}^{\varphi_i}$ for each FLTL formula $[\phi_i]\,\varphi_i \in \Phi$ with $i \in \{1 \,..\, n\}$ and $n = |\Phi|$. To this end, we utilize the techniques developed by Bauer, Leucker, and Schallhart for LTL_3 monitoring [BLS06b]. We then transform each of these monitors $\mathcal{M}_{\mathrm{LTL}}^{\varphi_i}$ into an FVTS $\mathcal{F}_{\varphi_i}$ with two initial states, one for when ϕ_i is satisfied and one for when it is not. Finally, we compose all the individual FVTSs using Definition 8.2.3 and then transform the verdicts of the resulting FVTS to be the conjunction over the individual verdicts.

Definition 8.2.4 *For each $i \in \{1 \,..\, n\}$, let $\mathcal{M}_{LTL}^{\varphi_i} = \langle Q_i, \{q_i\}, \wp(\mathsf{AP}), \rightarrow_i, \mathbb{B}_3, \sqsubseteq, \nu_i\rangle$ be the VTS obtained from the LTL_3 monitor for $[\phi_i]\,\varphi_i \in \Phi$. Further, for each of these VTSs, let $\mathcal{F}_{\varphi_i}$ be an FVTS such that*

$$\mathcal{F}_{\varphi_i} = \langle Q_i \cup \{t\}, \{q_i, t\}, \wp(\mathsf{AP}), \rightarrow_i, \mathbb{B}_3, \sqsubseteq, \nu_i', F, \mathsf{Conf}, g_i, \iota_i\rangle$$

with $\iota_i(q_i) := \phi_i$, $\iota_i(t) := \neg\phi_i$, $g_i'(\mathsf{t}) = \mathit{true}$ for $\mathsf{t} \in \rightarrow_i$, $\nu_i'(q) = \nu_i(q)$ for $q \in Q_i$, and $\nu_i'(t) = t$. We obtain an FVTS $\mathcal{F}_\Phi^\times = \mathcal{F}_{\varphi_1} \parallel \cdots \parallel \mathcal{F}_{\varphi_n}$ by composition.

The FVTS $\mathcal{F}_\Phi^\times$ produces verdicts of the product domain, i.e., it retains the verdicts of the individual formulas. To get an FVTS $\mathcal{F}_\Phi$ for the conjunction, we must thus redefine the verdict function ν of $\mathcal{F}_\Phi^\times$ by combining the individual verdicts by conjunction. For two formulas, we may redefine ν as follows:

$$\nu'(\langle q_1, q_2\rangle) \coloneqq \begin{cases} \mathsf{t} & \text{iff } \nu(\langle q_1, q_2\rangle) = \langle \mathsf{t}, \mathsf{t}\rangle \\ \mathsf{f} & \text{iff } \nu(\langle q_1, q_2\rangle) = \langle \mathsf{f}, \cdot\rangle \text{ or } \nu(\langle q_1, q_2\rangle) = \langle \cdot, \mathsf{f}\rangle \\ ? & \text{otherwise} \end{cases}$$

That is, for $|\Phi| = 2$, the FVTS $\mathcal{F}_\Phi$ is obtained by replacing the verdict function ν of $\mathcal{F}_\Phi^\times$ with ν' as just defined. This definition naturally extends to multiple formulas, leading to a featured LTL$_3$ monitor $\mathcal{F}_\Phi$ for Φ.

Theorem 8.2.1 *Given a non-empty set Φ of FLTL formulas, the FVTS $\mathcal{F}_\Phi$ is a solution to the featured LTL$_3$ monitor synthesis problem, i.e., for each configuration $c \subseteq F$, we have that $\mathcal{F}_\Phi{\restriction}_c$ is an LTL$_3$ monitor for $\Phi{\restriction}_c$.*

Proof Sketch. Proven by induction on the number of FLTL formulas [cf. DK22]. □

The resulting featured LTL$_3$ monitors are instances of configurable monitors. They can be instantiated for a given configuration and they can avoid an exponential blowup by structural sharing. Therefore, we have addressed the configurable monitor challenge for featured LTL successively.

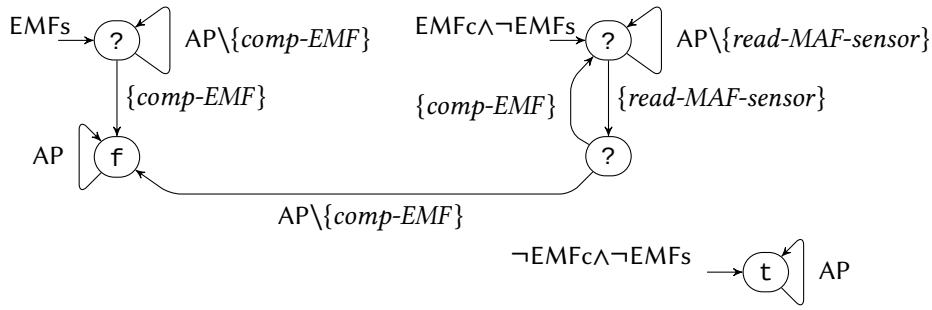


Figure 8.2: Featured LTL$_3$ monitor for the set of formulas defined in Example 8.1. For presentation purposes, the observables have been simplified to AP instead of $\wp$(AP), i.e., it is assumed that individual atomic propositions are observed.

Example 8.2 Figure 8.2 shows the featured LTL$_3$ monitor constructed for the set Φ of FLTL formulas as defined in Example 8.1 for the RDE use case. It has been constructed by composition from the FVTSs shown in Figure 8.3. According to the

RDE feature model (cf. Figure 8.1), the EMF computation feature (EMFc) or the EMF sensor feature (EMFs) is required. If the EMF sensor feature is enabled, then the EMF should not be computed according to ψ_0. If the EMF sensor feature is not enabled and the EMF computation feature is, then the EMF should be computed after reading the MAF sensor according to ψ_1. Depending on the configuration, the combined monitor starts in one of three states. In case neither the EMF computation nor the EMF sensor feature is enabled, none of the properties apply and the monitor thus yields `t`. If the EMF sensor feature is enabled, then the monitor produces `?` until it observes the computation of the EMF at which point it produces `f`. If the EMF sensor feature is not enabled but the EMF computation feature is, then it waits for the reading of the MAF sensor and requires the computation of the EMF in the next step. If this computation does not happen, it produces `f`. Otherwise, it produces `?`.

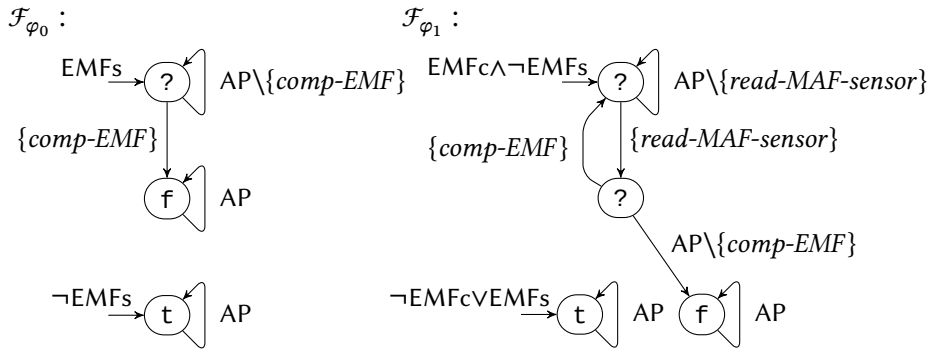


Figure 8.3: FVTSs constructed from the LTL_3 monitors for the FLTL formulas as defined in Example 8.1. As for Figure 8.2, the observables have been simplified.

Concise Featured LTL_3 Monitors. The feasibility of the approach can be further enhanced towards more concise featured LTL_3 monitors. The presented definitions consider the entire set of configurations instead of just the valid configurations according to some feature model. Constraining the construction to valid configurations can reduce the size of the resulting monitors. To this end, initial states with invalid configurations must be disregarded together with any fragments that may become unreachable as a result. Furthermore, results from bisimulation minimization for FTSs [BBV15] can be exploited towards minimal featured LTL_3 monitors. To this end, bisimulation minimization must be performed with respect to state classes preserving verdicts, analogously to the minimization of VTSs (cf. Section 3.2.2).

8.3 Configurable Monitoring with Lola

With featured VTSs and featured LTL_3 monitors, we addressed the configurable monitors challenge with an automaton-based approach. Addressing the challenge also from the stream-based perspective, we now present a configurable variant of the stream-based specification language Lola [DAn+05] (see Section 2.5.2). We refer to this variant as *configurable Lola*. Towards configurable Lola, we first introduce a notion of composability for sets of Lola specifications. After introducing configurable Lola, we then present a family-based analysis for checking whether all valid configurations of a configurable Lola specification are well-formed and efficiently monitorable. Recall that well-formedness is required for the Lola monitoring algorithm to be applicable and that efficient monitorability does guarantee that the memory requirements for monitoring do not grow without bounds (see Section 2.5).

Example 8.3 Let us return to the example of RDE monitoring. In the original works on RDE monitoring, multiple Lola specification fragments are pieced together in an ad-hoc text-based fashion to account for the different sensors a car may provide [KHB18; Bie+21; Bie+23]. Figure 8.4 (next page) shows excerpts of the different fragments. We see two fragments for obtaining the EMF (`emf`) and the fuel rate (`fuel_rate`), respectively. Consider the case where a car has an FAE and a MAF sensor. In this case, the fragments FRc1 and EMFc must be combined to compute the fuel rate based on the FAE and MAF sensor, and the EMF based on the computed fuel rate and the MAF sensor. Instead of piecing those fragments together in a mere text-based fashion, we will introduce a more systematic approach also amenable to analyses.

Composition of Specifications. We say that two Lola specifications are *composable* iff their dependent stream variables do not overlap, i.e., iff there exists no stream variable for which both specifications define an expression. For instance, in case of the fragments shown in Figure 8.4, EMFs and EMFc both define an expression for the variable `emf`, i.e., they are not composable. In contrast, the fragments EMFc and FRc1 are composable, as their dependent stream variables do not overlap. This definition of composability is naturally lifted to families of Lola specifications:

Definition 8.3.1 *A family* $\{L_i\}_{i=1}^n$ *of Lola specifications, all over the same set* $\mathbb{S}$ *of typed stream variables, is* composable *iff their dependent stream variables (recall Section 2.5.2) are disjoint. Formally, that is* $\text{Dom}(L_i) \cap \text{Dom}(L_k) = \emptyset$ *for all* $1 \leq i < k \leq n$. *Their* composition *is given by* $\bigcup_{i=1}^n L_i$.

Notably, $\bigcup_{i=1}^n L_i$ is a Lola specification, as the union of a family of partial functions with non-overlapping domains is a partial function (cf. Section 2.1). As discussed, the restriction to composable specifications ensures that the domains are non-overlapping.

EMF sensor fragment (EMFs):

```
input emf_sensor: Float64; // g/s
output emf = emf_sensor;
```

EMF computation fragment (EMFc):

```
input maf_sensor: Float64; // g/s
input fuel_rate: Float64; // g/s
output emf = maf_sensor + fuel_rate;
```

FR sensor fragment (FRs):

```
input fuel_rate_sensor: Float64; // g/s
output fuel_rate = fuel_rate_sensor;
```

FR computation fragment (FRc1):

```
input maf_sensor: Float64; // g/s
input fae_sensor: Float64; // ratio
output fuel_rate = maf_sensor / (14.5 * fae_sensor)
```

Figure 8.4: Lola specification fragments for the different variants of computing the EMF and fuel rate for low-cost RDE testing [KHB18; Bie+21; Bie+23].

Thereby, the composition also does not have any ambiguity with respect to the stream expressions assigned to dependent stream variables. We call a family of Lola specifications *incompatible* iff they share dependent stream variables. Incompatible Lola specifications cannot be composed.

Configurable Lola Specifications. Leveraging this notion of composability, we now define *configurable Lola specifications* as follows:

Definition 8.3.2 *A* configurable Lola specification *is a family* $\Xi = \{\mathsf{L}_i\}_{i=1}^{n}$ *of Lola specifications over the same set* $\mathbb{S}$ *of typed stream variables. We call the individual specifications* L_i features *of* Ξ*. A* configuration C *of* Ξ *is a subset of* Ξ*. A* valid configuration C *of* Ξ *is a configuration that is composable.*

Configurable Lola specifications make variability a first-class concept, enabling the specification of configurable monitors. Each valid configuration corresponds to a unique Lola specification as per Definition 8.3.1. In the following, we make use of this fact and simply treat valid configurations as if they are Lola specifications. In particular, we apply the concepts of well-formedness and efficient monitorability (see Section 2.5.2) directly to them. Notably, all incompatible pairs of features can

also be efficiently enumerated in $\mathcal{O}(|\Xi|^2 \cdot |\mathbb{S}|)$ time, by going through all pairs and checking whether they have overlapping dependent variables.

Given a configurable Lola specification, we aim to check whether all its valid configurations are (a) well-formed and (b) efficiently-monitorable.

Configurable Lola Analysis Problem. Given a configurable Lola specification Ξ, determine wether all valid configurations of Ξ are (a) well-formed and (b) efficiently-monitorable.

A naive solution to the Configurable Lola Analysis Problem would simply enumerate all configurations, check whether they are valid, and then apply the normal algorithms for checking well-formedness and efficient-monitorability. In practice, this approach quickly gets infeasible as the number of configurations grows exponentially in the number of features of the specification. Therefore, a more efficient family-based analysis is needed that can avoid the combinatorial blowup by exploiting commonalities between (valid) configurations. Such *all-in-one* family-based analyses are commonplace in feature-oriented system design [Thü+14].

Example 8.4 The Lola specification fragments shown in Figure 8.4 together form a configurable Lola speciation $\Xi = \{\mathsf{EMFs}, \mathsf{EMFc}, \mathsf{FRs}, \mathsf{FRc1}\}$. In this case, there are 16 configurations, nine of which are valid. The valid configurations are:

$$\{\} \quad \{\mathsf{EMFs}\} \quad \{\mathsf{EMFc}\} \quad \{\mathsf{FRs}\} \quad \{\mathsf{FRc1}\}$$
$$\{\mathsf{EMFs}, \mathsf{FRs}\} \quad \{\mathsf{EMFs}, \mathsf{FRc1}\} \quad \{\mathsf{EMFc}, \mathsf{FRs}\} \quad \{\mathsf{EMFc}, \mathsf{FRc1}\}$$

By checking each valid configuration individually, we can see that they are indeed all well-formed and efficiently monitorable.

8.3.1 Family-Based Specification Analysis

For checking well-formedness and efficient-monitorability for all valid configurations without an exponential blowup, we present a family-based analysis that exploits commonalities between configurations. Instead of using a dependency graph for each configuration in isolation, we construct a *family dependency graph*:

Definition 8.3.3 *Let $\Xi = \{\mathsf{L}_1, \ldots, \mathsf{L}_m\}$ be a configurable Lola specification with m features over the stream variables $\mathbb{S}$. The* family dependency graph *for Ξ is a directed, weighted, and feature-labeled multi-graph $G = \langle \mathbb{S}, E \rangle$ where E is the set of* edges. *An* edge *is a quadruple $\langle s_x s_y, z, i \rangle$ where $s_x, s_y \in \mathbb{S}$, $z \in \mathbb{Z}$, and $1 \leq i \leq m$. The set E of edges contains an edge $\langle s_x s_y, z, i \rangle$ if and only if $s_x \in \mathsf{Dom}(\mathsf{L}_i)$ and the expression $\mathsf{L}_i(s_x)$ contains an expression $s_y[z, c]$ for some constant c.*

According to Definition 8.3.3, the existence of an edge $\langle s_x, s_y, z, i \rangle$ in E records the fact that the stream for s_x depends on the stream for s_y with an offset of z when activating the feature L_i. The family dependency graph is a superimposition of the dependency graphs of each individual feature with additional edge labels for the features they belong to. Hence, the following criterion is easily established:

Lemma 8.3.1 *If the family dependency graph of a configurable Lola specification does not contain a zero-weight cycle, then every valid configuration is well-formed.*

Proof Sketch. Clearly, the dependency graph of every valid configuration is a subgraph of the family dependency graph. Therefore, if the family dependency graph does not contain a zero-weight cycle then the dependency graphs of any individual valid configuration cannot contain such a cycle either. □

Figure 8.5 shows the family dependency graph (without the dashed lines) for the configurable Lola specification based on the fragments in Figure 8.4. It does not contain any zero-weight cycles, in fact, it does not contain any cycles at all. Hence, all valid configurations for the configurable Lola specification are well-formed.

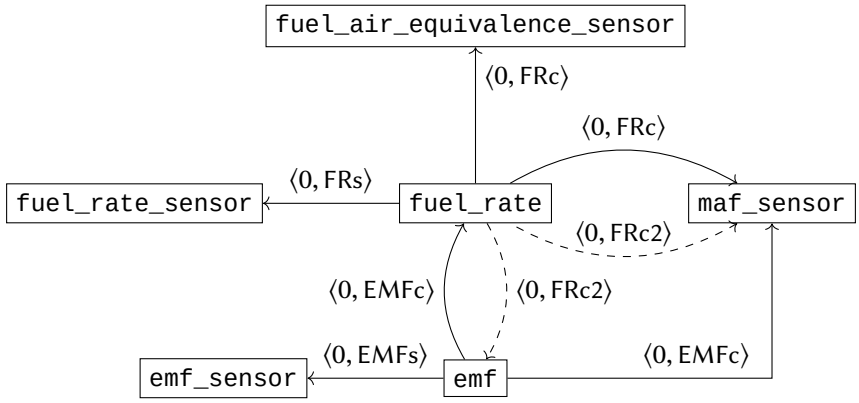


Figure 8.5: Family dependency graph for the configurable Lola specification based on Figure 8.4. The arrows are to be read as "depends on" with the given offset and feature. The dashed lines are introduced by the FRc2 feature (see Figure 8.6).

Lemma 8.3.1 gives us a sufficient criterion for well-formedness of every valid configuration. This criterion can be checked with the same algorithms and techniques as checking well-formedness of an individual specification. In contrast to the naive approach, which would consider each configuration individually, these algorithms can now exploit commonalities between the dependency graphs of the different configurations thereby mitigating the exponential blowup due to the often exponential

number of configurations. While the criterion is sufficient for well-formedness, it is more strict than required. We now relax the criterion towards a necessary and sufficient criterion for well-formedness.

Family-Based Well-Formedness. Intuitively, only those zero-weight cycles pose a problem that can actually arise from a *valid* configuration, i.e., a set of pairwise composable features. By adding this additional condition for cycles, we obtain the following theorem for family-based well-formedness:

Theorem 8.3.1 *All valid configurations of a configurable Lola specification Ξ are well-formed iff each zero-weight cycle in the family dependency graph of Ξ contains at least two edges labeled with features which are not composable.*

Proof Sketch. The dependency graph of an individual valid configuration can be obtained from the family dependency graph by removing all edges corresponding to features which are not enabled. Now, if a zero-weight cycle in the family dependency graph contains at least two edges labeled with features which are not composable, this cycle will not be included in a dependency graph of any of the valid configurations as a valid configuration can only contain features which are pairwise composable. If this is the case for all zero-weight cycles in the family dependency graph, then none of these cycles will be included in the dependency graph of any of the valid configurations. As a result, all valid configurations are well-formed. Conversely, if a zero-weight cycle exists which does not contain two edges that are labeled with non-composable features, then the respective features on this cycle can be composed to form a valid configuration which is not well-formed. □

FR computation fragment (FRc2)

```
input maf_sensor: Float64; // g/s
input emf: Float64; // g/s
output fuel_rate = emf - maf_sensor;
```

Figure 8.6: Additional feature for computing the fuel rate from the EMF and MAF.

Example 8.5 As an example, consider the feature FRc2 as shown in Figure 8.6 to be added to the configurable Lola specification based on Figure 8.4. This introduces additional edges in the dependency graph (dashed edges in Figure 8.5). With this feature, the family dependency graph now contains one elementary zero-weight cycle between `fuel_rate` and `emf`. Indeed, the valid configuration with both the FRc2 and EMFc feature is not well-formed. Enabling both features would mean that the `fuel_rate` should be computed based on the `emf` but at the same time the `emf`

should be computed based on the `fuel_rate`. This cycle means that the monitor for this configuration is not well-defined. Note that FRc2 and EMFc are composable because they contain no overlapping definitions. With the family-based analysis this can be detected solely relying on the family dependency graph and without considering all $2^5 = 32$ configurations one-by-one.

Complexity. As the zero-weight cycle problem, the problem of finding a zero-weight cycle not containing two edges that are labeled with incompatible features is also NP-complete. It is more general than the zero-weight cycle problem because it contains an additional condition on cycles: Instances of the traditional zero-weight cycle problem can be encoded by having a specific feature for each edge. Nevertheless, the problem still lies in NP because it is easy to verify in polynomial time that a cycle is zero-weight and does not contain two edges that are labeled with non-composable features. Hence, the complexity class of the problem remains unchanged.

Notably, the family dependency graph is typically larger than the graphs for the individual configurations. To reduce its size, one can collapse all edges $\langle s_x, s_y, z, i\rangle$ between the same vertices s_x and s_y that have the same weight z into a single edge $\langle s_x, s_y, z, I\rangle$ where I is the set of all feature labels found on any of these edges. This can drastically reduce the number of edges and thereby the number of potential zero-weight cycles to be considered by the analysis.

Efficient Monitorability. Besides well-formedness, efficient monitorability is a property that is of particular interest for Lola specifications. Efficiently monitorable specifications are guaranteed to be monitorable with a bounded amount of memory independent of the length of the involved streams. A Lola specification is efficiently monitorable if and only if its dependency graph does not have positive cycles [DAn+05]. Our family-based analysis and Theorem 8.3.1 is easily extended to the question whether all configurations are efficiently monitorable:

Theorem 8.3.2 *All valid configurations of a configurable Lola specification Ξ are efficiently monitorable iff every positive-weight cycle in its feature dependency graph contains at least two edges labeled with features which are not composable.*

Proof Sketch. The reasoning for Theorem 8.3.2 is analogous to Theorem 8.3.1. □

Practical Impact. Taken together Theorem 8.3.1 and Theorem 8.3.2 provide an effective solution for the Configurable Lola Analysis Problem. Checking the family dependency graph instead of the dependency graph of each valid configuration in isolation can mitigate the exponential blowup in the number of features. It allows making sure that all valid configurations indeed give rise to a well-formed and efficiently monitorable specification. This, in turn, means that a monitor can be synthesized

and that its memory consumption will be bounded. Based on found zero-weight cycles, valid configurations that would not lead to a well-formed specification can be identified ahead-of-time, providing a static guarantee for configurations at runtime. If the valid configurations are further restricted, e.g., by a feature diagram, then zero-weight (or positive-weight) cycles not corresponding to a valid configuration according to the additional restrictions can be ignored. Thereby, the presented analyses extend to cases where not all valid configurations according to Definition 8.3.2 may be relevant in practice but only a subset of them.

We also like to mention that Lola specifications can be parametrized and that parametrization can be used for configurable monitors. However, while emulating features as we considered them with parameters and ite expressions (see Section 2.5.2) is possible to some extent, the traditional well-formedness analysis will not understand that the different cases of ite expressions are mutually exclusive. Thus, it would essentially correspond to a coarse-grained analysis according to Lemma 8.3.1. Instead, the analysis we propose here is more fine-grained. In addition, when emulating features using parameters, the independent variables of all features would be merged with no explicit distinction about which actually have to be provided and which are merely an encoding artifact. Thus, our contributions complement parameters offering a more fine-grained analysis and explicit treatment of features and independent variables. Together, parameters and features as we considered them make Lola a perfect fit for runtime verification of configurable systems.

8.4 Configuration Monitoring

Having addressed the configurable monitors challenge from an automaton- and stream-based perspective, it remains to tackle the configuration monitoring challenge. To this end, we instantiate the generic synthesis pipeline developed in Chapter 4 for configuration monitor synthesis. A configuration monitor determines an a-priori unknown static configuration of a running system.

8.4.1 Configuration Monitor Synthesis

Recall that configurable systems are commonly modeled as featured transition systems (FTSs) (see Section 2.4). Given an FTS $\mathfrak{F} = \langle \mathcal{S}, I, \mathrm{Act}, \twoheadrightarrow, F, \mathrm{Conf}, g, \iota \rangle$ modeling a system, we aim to synthesize a configuration monitor that takes observations generated by the system and that produces configuration verdicts indicating which configurations the system may have (cf. Definition 3.1.4).

Leveraging the techniques developed in Chapter 4, configuration monitors are straightforward to synthesize given an FTS system model. To this end, we first need to define a verdict annotation as per Definition 4.1.1. Recall that the configuration verdict domain is $\langle \wp(\mathrm{Conf}) \setminus \{\varnothing\}, \subseteq \rangle$ (see Definition 3.1.4), where Conf is the set of valid

system configurations. Each configuration verdict $v \in \wp(\text{Conf})\backslash\{\varnothing\}$ corresponds to a set of valid configurations the system may have. For purpose of configuration monitor synthesis, we define the following verdict annotation over this verdict domain on the underlying transition system $\langle S, I, \text{Act}, \twoheadrightarrow\rangle$ of the FTS $\mathfrak{F}$:

$$\iota(s) := [\![\iota(s)]\!]_{\mathbb{B}} \cap \text{Conf} \qquad \lambda(s) := \text{Conf} \qquad \gamma(t) := [\![g(t)]\!]_{\mathbb{B}} \cap \text{Conf} \quad (8.2)$$

The resulting verdict oracle as per Definition 4.1.2 essentially combines the feature guards of all taken transitions by conjunction. As a result, a VTS that is sound and complete with respect to this verdict oracle and a given observation model produces most specific configuration verdicts indicating which configurations the system may have. Thus, a sound and complete VTS with respect to this verdict oracle is a configuration monitor. It can be synthesized with the synthesis pipeline by applying annotation tracking followed by sentinel pruning (recall Section 4.1.2).

Recall that in this case the sentinel verdict indicates that a certain run does not actually exist in any valid configuration (see Section 4.1.2). The empty set is not a part of the verdict domain, so if the conjunction of the guards is empty, the sentinel verdict will be returned. If we can assume that the system can only be configured towards valid configurations, which is the usual assumption, we can prune those states from a VTS that have the sentinel verdict #. Otherwise, we may as well add the empty set to the verdict domain to detect observations that do not conform to any of the configurations of the FTS.

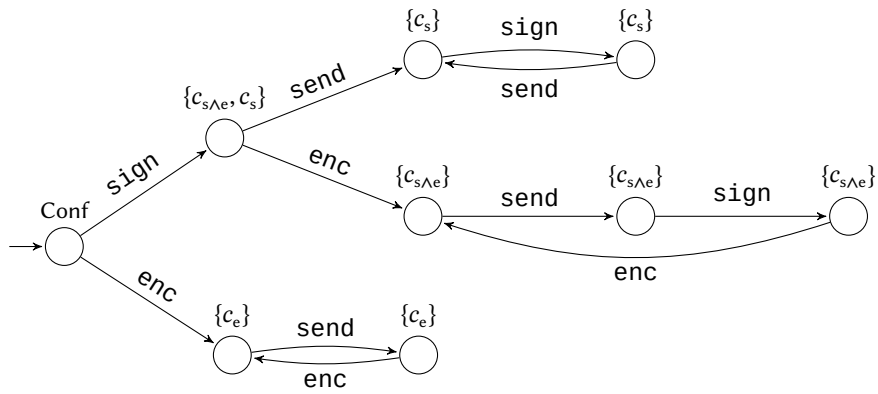


Figure 8.7: VTS constructed from the FTS model of the email system (see Figure 2.5) by annotation tracking with sentinel pruning.

Example 8.6 Figure 8.7 depicts the VTS constructed by annotation tracking with sentinel pruning (see Section 4.1.2) from the email system model depicted in Figure 2.5. Notably, due to sentinel pruning, the VTS does not accept traces that do not

correspond to any valid configuration. For instance, while possible in the underlying transition system, the trace

$$\texttt{sign} \diamond \texttt{send} \diamond \texttt{enc}$$

does not actually correspond to any valid configuration. If `sign` is directly followed by `send`, then the email system has been configured with the sign feature but not with the encryption feature, otherwise, the email would have to be encrypted before it is sent. Hence, such a sequence cannot be followed by `enc`. The VTS correctly recognizes this and produces the verdict $\{c_s\}$, when fed with the trace $\texttt{sign} \diamond \texttt{send}$, indicating that only the sign feature is enabled. It also correctly recognizes the other configurations as soon as possible, i.e., the verdicts are indeed factually correct and most specific. For instance, when fed the trace $\texttt{sign} \diamond \texttt{enc}$, it will indicate that both, the signing and the encryption feature must be enabled.

To check whether a system has been configured correctly, the verdict function of a configuration monitor can be redefined after synthesis to produce truth verdicts. Given a configuration monitor $\langle Q, J, \text{Obs}, \rightarrowtriangle, \wp(\text{Conf}), \subseteq, \nu \rangle$ over some set Conf of valid configurations and a non-empty set $C \subseteq \text{Conf}$ of *correct configurations*, we define the following alternative verdict function $\nu' : Q \to \mathbb{B}_3$:

$$\nu'(q) := \begin{cases} \texttt{t} & \text{if } \nu(q) \subseteq C \\ \texttt{f} & \text{if } \nu(q) \cap C = \varnothing \\ \texttt{?} & \text{otherwise} \end{cases}$$

Substituting this function into the original configuration monitor, we obtain a VTS $\langle Q, J, \text{Obs}, \rightarrowtriangle, \mathbb{B}_3, \sqsubseteq, \nu' \rangle$. This VTS produces `t` if the configuration of the system is certainly correct, i.e., all possible configurations are correct. In case all possible configurations are not correct, it produces `f`. Otherwise, it produces `?`, indicating that there is not sufficient information to conclude whether the configuration of the system is correct. With this transformation, configuration monitors can be used to answer operational questions regarding the correct configuration of a system, e.g., whether manufacturing or medical equipment has been correctly configured to safely proceed to the next step (recall (Q3) from Chapter 1).

Online Reconfiguration. Analogously to fault diagnosis (recall discussion in Section 4.1.1 and Section 7.2), extending configuration monitors to settings with online reconfiguration is straightforward by annotating states of the system model with sets of configurations a system may have in each state, respectively. That is, instead of annotating states with the set of all valid configurations Conf as per (8.2), the annotation is restricted to those configurations a system may have when being in a given state. The synthesis pipeline can then used as is.

Furthermore, without sentinel pruning, one would also get the sentinel verdict # in cases where the system has been reconfigured at runtime leading to observations that do not belong to any single valid configuration. So, configuration monitors can also be used to detect online reconfigurations.

8.4.2 Evaluation on Community Benchmarks

To demonstrate the efficacy of the developed synthesis techniques, we consider configuration monitors synthesized from established FTS benchmarks of the configurable systems community: Svm and Minepump [Cla10], and Aerouc5, Cpterminal, and Claroline [Dev17]. Table 8.1 (next page) shows an overview of these benchmarks and their important characteristics.

	Svm	Minepump	Aerouc5	Cpterminal	Claroline
|Conf|	24	32	256	4 774	820 193 280
|Act|	12	23	11	15	106
FTS	9/13	25/41	25/46	11/17	106/11 236
monitor	87/120	560/992	94/178	102/161	5 431 296/575 717 376
minimized	87/120	496/928	56/156	93/152	65 536/6 946 816
relaxed	17/26	103/337	4/4	11/26	65 536/1 515 520

Table 8.1: For each model, the rows show (1) the number of valid configurations, (2) the number of actions, (3) the size of the FTS (states/transitions), (4) the size of the monitor constructed from the FTS, and the size of the monitor after (5) language-preserving and (6) language-relaxing minimization.

As part of Momba, we developed an implementation of the synthesis pipeline instantiated for configuration monitors where we use BDDs [Bry86] to succinctly represent and operate on sets of configurations. The implementation uses the well-established CUDD library [Som15].[38] The implementation first applies annotation tracking with sentinel pruning to the FTSs (as discussed above), followed by observability projection, determinization, and minimization. This leads to deterministic, minimal, sound, and complete configuration monitors.

The following experiments have been conducted on a 16 core *AMD Ryzen 9 5950X* CPU with 128 GiB of RAM running Ubuntu 22.04. The source code, models, and data used for the experiments presented here is available as part of artifact (AT1). In our

[38] OxiDD [Hus+24] has not yet been released at the time of development. In future versions, we plan to switch from CUDD to OxiDD, towards a state-of-the-art symbolic integration.

evaluation, we aim to answer the following research questions concerning our novel contribution of configuration monitors:

(CMRQ1) How do monitor sizes scale with the number of configurations?

(CMRQ2) What are the potential space savings of minimization?

(CMRQ3) How does partial observability impact the specificity of verdicts?

(CMRQ1): *How do monitor sizes scale with the number of configurations?* Except for Claroline, the size of the FTSs and the number of configurations is comparably small. Note that a configuration monitor may need to distinguish all possible configurations, potentially leading to an exponential blowup. Table 8.1 shows the size of the configuration monitors prior to minimization (4) and after minimization (5,6). We observe across all benchmarks that configuration monitors can be significantly smaller than the number of configurations would suggest. Claroline shows the greatest divergence with roughly $8 \cdot 10^8$ configurations while the monitor has about $5 \cdot 10^6$ states. A similar but not as extreme observation can be made about Aerouc5 and Cpterminal. Monitor sizes also influence the construction timings. The Claroline monitor synthesis took around seven minutes, while for all other benchmarks the synthesis (including determinization and minimization) took only a few milliseconds. Reachability analysis on Claroline was already shown to be challenging [Bee+19; Bee+22]). While known to be challenging, even for Claroline, our techniques allow for fast and effective configuration monitor synthesis.

(CMRQ2): *What are the potential space savings of minimization?* Table 8.1 shows the size of the monitors after normal (5) and language-relaxing (6) minimization, respectively (see Section 4.4). For the latter, we also removed self loops, i.e., the monitor stays in its state if a non-enabled action is observed. In particular, language-relaxing minimization reduces the number of states significantly, leading to very small monitors. For Aerouc5, Cpterminal, and Claroline, we discover that the number of states is even further reduced. So, we conclude that minimization can indeed significantly reduce VTS sizes. Noteworthily, the number of states provides an upper bound on the number of configurations which can be distinguished by observation. In the extreme case, Claroline, this number is four orders of magnitude lower than the number of configurations. Thus, most configurations are indistinguishable by an observer, even under full-observability. So, as a byproduct, our work on configuration monitoring has revealed an explanation for successes reported in family-based analysis [Cla+13]. Family-based analyses are effective precisely because most configurations share behavior. The same property is also what enables the effective synthesis of configuration monitors.

	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = \lvert\text{Act}\rvert$
Svm	26% (0%)	61% (23%)	79% (26%)	**83%** (33%)	83%
Minepump	26% (0%)	45% (0%)	60% (0%)	71% (0%)	**79%**
Aerouc5	25% (0%)	**44%** (0%)	44% (0%)	44% (0%)	44%
Cpterminal	24% (0%)	37% (0%)	**40%** (0%)	40% (8%)	40%

Table 8.2: Maximal (minimal) expected percentage of ruled-out configurations after 1 000 steps over all combinations of k observable actions.

(CMRQ3): *How does partial observability impact the specificity of verdicts?* To answer this question, we employ the following methodology: We construct monitors where only a limited number of k actions of Act are considered to be observable. For this, we range over all subsets of Act with k elements, for $1 \leq k \leq 4$ and $k = |\text{Act}|$, and employ Monte Carlo simulation to compute the expected percentage of ruled-out configurations after 1 000 steps. To this end, $160 \cdot 10^3$ runs, each of length 1 000, are simulated through the system models and observations are fed to the synthesized monitor.[39] For each of these runs, we uniformly sample a configuration and choose actions uniformly at random. This gives us a set of expected percentages for each k of which we report the maximum and minimum in Table 8.2. In total, for Table 8.2, we synthesized 14 200 different monitors and conducted approximately $2\,272 \cdot 10^6$ simulation runs. Exploiting parallelization, this took 2.5 hours on our benchmark machine. Note that this approach based on Monte Carlo simulation is unsuitable for Claroline as the huge number of valid configurations would require many more runs to obtain statistically significant results. Hence, we omit it here.

Looking at the results in Table 8.2, we see that for Aerouc5 and Cpterminal, on average only around 42% of configurations can be ruled out after 1 000 steps (which is sufficient for the monitor to converge on a verdict). In contrast, for Svm and Minepump, on average 81% of configurations can be ruled out. This fits our earlier observation that the monitors for the latter benchmarks have a higher number of states compared to the number of valid configurations than Aerouc5, Cpterminal, and Claroline. Thus, they can distinguish more configurations.

The results also show that the precise set of actions is crucial, as otherwise the specificity of the verdicts may not improve at all. For instance, for Aerouc5, with $k = 2$ observable actions, we already can obtain optimal verdicts, while even with $k = 4$, there are sets of observable actions where we cannot discern any configurations at all. For Svm $k = 4$ and for Cpterminal $k = 3$ actions can be sufficient. For Minepump, no k-combination for $k \leq 4$ is sufficient. Note that the number of possible

[39] These numbers provide a good tradeoff between noise and experiment running time.

subsets of Act is $2^{|\text{Act}|}$, hence, we did not investigate anything beyond $k = 4$, except $k = |\text{Act}|$, which represents full observability. For our benchmarks, we conclude that is often suffices to observe just a small set of decisive actions.

8.5 Discussion

In this chapter, we tackled two significant challenges that arise for runtime monitoring in the context of configurable systems.

Towards addressing the configurable monitors challenge, we extended the theory around VTSs developed in this thesis with features. The resulting notion of FVTSs provides a solid foundation for configurable monitors. We demonstrated the practical utility of this approach by showing how to synthesize FVTSs from sets of FLTL formulas building upon existing LTL monitoring techniques [BLS06b]. Furthermore, we introduced a configurable variant of the stream-based specification language Lola, together with family-based algorithms for effectively determining well-formedness and efficient-monitorability for all possible valid configurations. We presented a use case for configurable Lola centered around exhaust emission monitoring. Our family-based analysis presents a substantial improvement over existing work where multiple parts of a specification are pieced together in an ad-hoc text-based fashion [Bie+21; Bie+23], without any guarantees on the well-formedness and efficient-monitorability of the resulting composite specifications.

As a system's configuration may be unknown at runtime and not readily exposed, we further presented configuration monitoring. Tackling the configuration monitoring challenge, we showed how the techniques developed in Chapter 4 can be leveraged for configuration monitor synthesis from FTSs. Conceptually, synthesizing configuration monitors has then been straightforward due to the flexibility of the generic verdictor algorithms we developed earlier in this thesis. Within the broader aim of this thesis, we have established that configuration monitors can be used to answer operational questions regarding the correctness of a system's configuration (recall (Q3) from Chapter 1). We further validated this synthesis approach on FTSs benchmarks from the configurable systems community. Our results show that configuration monitors can be effectively synthesized, demonstrating the feasibility of the verdictor synthesis pipeline. Furthermore, the results provide a new explanation for successes seen in family-based system analysis.

Other Runtime Verification Techniques. We have introduced featured LTL_3 runtime verification as well as configurable Lola. For presentational simplicity, we considered the original variant of Lola [DAn+05]. Our extension towards configurable Lola is orthogonal to the extensions introduced by Lola 2.0 [Fay+16] and RTLola [Fay+19; Bau+20]. It appears straightforward to extend our family-based

analysis approach to these extensions. Besides LTL_3 runtime verification and Lola, the huge body of research on runtime verification has developed many more highly valuable techniques (see Chapter 1 for some examples). With regard to configurable monitors, it remains an open challenge to explore how those can be made variability-aware and harvested for configurable monitoring.

Continuous Time Configuration Monitoring. While we discussed configuration monitoring in the discrete-time setting, an extension to the continuous-time setting is straightforward. To this end, a system model in terms of a featured timed automaton [Cor+12] may be given. Analogously to the verdict annotation derived from an FTS system model for configuration monitoring, a featured timed automaton can be used to derive verdict annotations as required for the continuous time verdictor algorithm developed in Chapter 5. This algorithm can than be used for configuration monitoring in the continuous time setting with timing imprecisions. The ease of this extension really highlights the power of the generic approach presented in this thesis, demonstrating its adaptability across different settings.

Self-Adaptive Monitoring. With configurable monitors and configuration monitors, we have developed solid foundations for further advances in variability-aware monitoring. In particular, by combining configurable monitors with configuration monitors, the techniques contributed by this thesis enable a powerful *self-adaptive monitoring* paradigm where a configurable monitor is configured based on the configuration determined by a configuration monitor. Related to this idea is the integration and combination of verdictors for different domains. For instance, observations indicative of a fault may vary with the configuration of a system and thus require the combination of a configuration monitor with a fault diagnoser. We believe that exploring such combinations in future work will enable many more applications beyond what seems obvious now.

Chapter 9

Conclusion and Outlook

Maintaining a consistently safe and functional state amid complex operational demands, potential technical failures, and human errors is paramount for safety- and mission-critical systems. This is particularly evident in domains like aviation, manufacturing, and healthcare, where any failures can directly impact human lives and lead to significant financial losses. For instance, accurate system diagnostics in aviation can be life-saving, while in manufacturing, ensuring correct configurations is crucial to avoid costly downtimes and ensure worker safety. Addressing these challenges requires not only rigorous design-time strategies but also robust techniques that provide timely and accurate information about a system's operational state at runtime. These techniques must be capable of answering critical operational questions, such as whether a system is operating within its safe limits (Q1), whether any vital components have failed (Q2), or whether the system is correctly configured (Q3).

In practice, obtaining accurate information about a system's operational state is complicated by observational imperfections, such as limited observability, delays, losses, and out-of-order observations. These imperfections introduce uncertainties that can obscure the true state of a system, hampering its accurate assessment. Yet, accurate information is often a prerequisite for effective interventions and safeguarding, as it ensures that a system can be reliably steered back to a safe and functional state when anomalies or failures occur. Conversely, inaccurate information can lead to misguided interventions, potentially exacerbating the situation, causing system failures, or leading to other issues which put the system at risk.

Against this backdrop, the central challenge addressed in this thesis is the development of techniques to obtain *provably accurate* information about a system's operational state. Following a model-based methodology, the aim has been to develop techniques that not only produce provably accurate information with respect to a given formal model of a system while considering potential observational imperfections, but also are sufficiently generic to enable novel applications.

Addressing this challenge, this thesis presents a range of contributions, from theoretical foundations to generic algorithms and concrete applications. We demonstrated that the developed techniques are broadly applicable across diverse domains, including runtime verification, fault diagnosis, and configurable systems. A particularly novel area opened up by the contributions of this thesis is variability-aware monitoring, tackling significant challenges when it comes to monitoring configurable systems. The practical applicability of the developed techniques is further enhanced by ensuring that they can be robust against a variety of observational imperfections typically unavoidable in practice.

Retrospective. As a starting point and rigorous foundation for the contributions of this thesis, we have presented a theoretical framework in Chapter 3. This framework comprises formalizations of key concepts underpinning this thesis, including verdict domains, verdict transition systems, observation models, and verdict oracles. The theoretical framework has culminated in a precise formal characterization for what it means to produce accurate information about a system's operational state. Within this framework, such information takes the form of verdicts produced based on observations by a verdict transition system modeling a verdictor.

To produce provably accurate verdicts, a verdictor must be proven to be sound and complete with respect to a given system model, observation model, and verdict oracle. Such a verdictor produces verdicts that are as specific as possible without sacrificing correctness. Besides playing a pivotal role for the generic verdictor algorithms presented in Chapter 4 and Chapter 5, we have showcased and discussed how the theoretical framework can serve as a unifying common ground for previously independent research strands on runtime verification and fault diagnosis. While their commonalities had been discussed before [Hav+10], we are not aware of any work explicitly providing a unifying formal foundation as we have done.

Facilitated by the theoretical framework, we have then developed several generic algorithms for verdictors in Chapter 4 and Chapter 5. These algorithms allow the synthesis and implementation of provably accurate verdictors across a variety of different applications in both discrete- and continuous-time settings.

For the discrete-time setting, we have presented a modular verdictor synthesis pipeline that follows an annotative approach resting on verdict-annotated system models. Through versatile transformations, the pipeline can be instantiated to synthesize verdictors that produce most specific predictions and that account for limited observability, unbounded and bounded losses, unbounded and bounded delays, as well as out-of-order observations. Since we established general theorems regarding these transformations, they can also be applied to verdictors synthesized with third-party techniques, as exemplified in Section 7.1.1. With language-relaxing minimization we furthermore have presented an algorithm for obtaining even smaller verdictors, at the cost of tolerating additional observation sequences.

For the continuous-time setting, we have presented a verdictor algorithm capable of processing observations with explicit timing information that are subject to timing imprecisions. As for the discrete-time setting, this algorithm is based on verdict annotations, allowing for a broad range of applications. A verdictor based on this algorithm produces accurate verdicts that are robust against varying latencies, varying clock drift, and unknown clock offsets, while also being capable of handling limited observability. While the algorithm's running time grows exponentially in the number of observations, we also introduced an over-approximative variant that mitigates this issue at the cost of producing potentially less specific verdicts. This over-approximative variant bounds the considered history and thereby achieves space and time complexity bounded by the size of the model.

All constructions, transformations, and algorithms have been proven correct with respect to the theoretical framework. By choosing appropriate verdict annotations, we have demonstrated that they can indeed be used across diverse domains, including runtime verification, fault diagnosis, and configurable systems.

As the contributions of this thesis are centered around formal models, we have developed Momba to make formal modeling more accessible. In Chapter 6, we have discussed Momba from a user perspective and have demonstrated its model construction, simulation, and analysis capabilities. Additionally, Momba has also served as the core for the implementation and evaluation of the techniques presented in this thesis. We evaluated Momba's performance by comparing it to other state-of-the-art tools. The results show that Momba can compete with state-of-the-art tools and even outperform them with its alternative engine on multi-core systems. So, with Momba, we contributed a tool that does implement the algorithms developed in this thesis and is not only easy to use and straightforward to install but is also state-of-the-art when it comes to the performance of its engine.

We have explored concrete applications of the developed techniques for runtime verification and fault diagnosis in Chapter 7. We have demonstrated how these techniques can be used to obtain robust and predictive runtime monitors and fault diagnosers, generalizing and complementing existing constructions. We have instantiated the techniques for full CTL runtime verification, and have introduced a new diagnosis approach based on Boolean fault annotations and modal logic. Furthermore, we have empirically validated the effectiveness of the verdictor algorithm for the continuations-time setting with a case study on diagnosis in a real-time industrial automation scenario. While not a comprehensive study on multiple real-world applications, our results are encouraging and show that the proposed over-approximative algorithm scales well while still producing sufficiently specific verdicts. The application for diagnosis of real-time systems relaxes the strict assumptions about the observation of event timings inherent to previous works [Tri02; BCD05]. The applications considered in Chapter 7 demonstrate that the techniques developed in this thesis can be used to answer pressing operational questions regarding the satisfaction

or violation of properties (Q1) and the presence of faults (Q2).

With variability-aware monitoring, we have introduced an entirely novel application area in Chapter 8, addressing key challenges for monitoring of configurable systems. Here, configurability of the system and its monitor are a concern and must be matched. We have presented a featured variant of verdict transition systems and leveraged it for the compositional synthesis of configurable monitors for sets of featured LTL formulas. With configurable Lola, we have also introduced an approach for configurable stream-based monitoring. To this end, we have developed a family-based analysis to determine well-formedness and efficient monitorability for all configurations of a specification while avoiding an exponential blowup that would occur with a naive consideration of each configuration individually. Finally, we have introduced configuration monitors, which determine an a-priori unknown configuration of the system, and have instantiated the techniques developed in Chapter 4 to synthesize them. We have evaluated the techniques for the discrete-time setting on configuration monitors. In addition to demonstrating practical effectiveness, our results also provide an explanation for the successes seen in family-based model analysis. Together these contributions enable adaptive monitoring, where a monitor is adapted to an a-priori unknown configuration of a system based on the verdicts produced by a configuration monitor. With configuration monitoring, the techniques developed in this thesis can be used to anwser pressing operational questions regarding possible system configurations and whether a system is configured correctly (Q3).

Conclusion. This thesis presents a host of cross-cutting contributions across runtime verification, fault diagnosis, and configurable systems. It advances the state of the art on a foundational level through the development of a theoretical framework (Contribution FT). Building upon this framework, it introduces concrete algorithms (Contribution TT) that have been proven correct and demonstrated to be practically applicable across various applications and domains (Contribution TP). Additionally, by providing state-of-the-art tooling with Momba (Contribution FP), this work also ensures that these innovations are accessible and usable.

Through the comprehensive exploration and development of various techniques, this thesis has made significant strides to address the central challenge of obtaining provably accurate information about a system's operational state. By combining theoretical rigor with practical applicability, the work presented here not only provides robust solutions to the challenges posed by observational imperfections but also introduces versatile tools that extend the reach of runtime verification and fault diagnosis, while enabling variability-aware monitoring as a completely novel application. The contributions made represent a meaningful advancement in the capability to monitor, diagnose, and ensure the safety of complex systems in dynamic and uncertain environments. This thesis has therefore laid a strong foundation for future research and applications in these critical areas.

Prospects. While we have discussed concrete future work as part of the individual chapters, we would like to emphasize here the broader potential of our contributions. First and foremost, the foundational and algorithmic building blocks we have developed can be combined freely towards novel applications. Although, we have showcased this potential on runtime verification, fault diagnosis, and variability-aware monitoring, we believe that there is ample room for further exploration and innovation. We envision such work to be complemented by comprehensive empirical evaluations on multiple real-world case studies. With Momba and the tool support it provides, we have already layed the foundations for such studies.

We view our work as a foundational starting point, and we anticipate that future research can build upon this foundation to develop further versatile and powerful verdictor algorithms, e.g., to account for other observational imperfections such as noise or spurious observations which we did not consider here.

An especially promising direction for future research is the exploration of verdictors in the context of explainability. Verdicts generated by the techniques developed in this thesis have the potential to serve as explanations for system behavior. For instance, identifying the occurrence of a fault or the presence of a certain feature can explain why a system behaves in a particular way when considered in conjunction with the system's model. Furthermore, accurate verdicts bear a causal relation to observations in line with established counterfactual theories of causation and explanation [Lew73; HP01]. For instance, if a fault had not occurred, then some observations would not have been made. Hence, having made said observations, one must conclude that the fault indeed occurred. In other words, the observations are explained by a verdict and in combination with the system's model, the observation model, and the verdict oracle. Causal explanations based on formal models and for configurable systems have recently also found some attention in recent literature [Bai+21; Dub+24]. Future research could further explore how verdicts can be systematically leveraged as explanations, enhancing the interpretability and transparency of system behavior for various stakeholders [Köh+19].

Appendix

Appendix A

Detailed Proofs

A.1 Modular Discrete-Time Verdictor Synthesis

A.1.1 Proof of Theorem 4.1.1

We restate Theorem 4.1.1 from the body of this thesis (p. 93).

Theorem 4.1.1 *The VTS $\mathfrak{V}^{\#}_{\mathfrak{S},\kappa,\lambda,\gamma}$ is sound and complete with respect to the system model $\mathfrak{S}$, the observation model Ω_{Trace}, and the verdict oracle $\mathsf{V}_{\kappa,\lambda,\gamma}$.*

For notational clarity, we simplify the notation as follows:

$$\Omega \coloneqq \Omega_{\mathsf{Trace}} \qquad \mathfrak{V} \coloneqq \mathfrak{V}^{\#}_{\mathfrak{S},\kappa,\lambda,\gamma} \qquad \mathsf{V} \coloneqq \mathsf{V}_{\kappa,\lambda,\gamma}$$

Proof. As per Lemma 3.4.1, $\mathfrak{V} = \langle \mathcal{Q}, J, \mathsf{Act}, \rightarrowtriangle, \mathcal{V}, \sqsubseteq, \nu \rangle$ is sound and complete with respect to $\mathfrak{S} = \langle \mathcal{S}, I, \mathsf{Act}, \twoheadrightarrow \rangle$, Ω, and V, iff it produces the most specific verdict $\mathsf{V}(\omega)$ as per (3.7) for every observation sequence $\omega \in \mathcal{L}\restriction_{\Omega}(\mathfrak{S})$ in the observable language of $\mathfrak{S}$. As per Definition 3.2.2, the verdict $\nu(\omega)$ produced by $\mathfrak{V}$ for a given observation sequence ω is the join of the verdicts of the states reached after ω.

Thus, we obtain the following proof goal:

$$\forall \omega \in \mathcal{L}\restriction_{\Omega}(\mathfrak{S}) : \quad \underbrace{\bigsqcup \{ \nu(q) \mid q \in \mathsf{After}_{\mathfrak{V}}(\omega) \}}_{\nu(\omega) \text{ as per Definition 3.2.2}} = \underbrace{\bigsqcup \{ \mathsf{V}(\rho) \mid \rho \in \mathsf{Runs}(\omega) \}}_{\mathsf{V}(\omega) \text{ as per (3.7)}} \tag{G1}$$

Here, $\mathsf{Runs}(\omega)$ is the set of runs of the system as per (3.4) that may generate ω. As Ω is the trace observation model (see Definition 3.3.3) we have:

$$\mathsf{Runs}(\omega) = \{ \rho \in \mathsf{Runs}(\mathfrak{S}) \mid \mathsf{Trace}(\rho) = \omega \} \tag{P1}$$

To prove (G1), let us distinguish the cases $\omega = \epsilon$ and $\omega \neq \epsilon$.

Case $\omega = \epsilon$. If ω is empty, then the following transformations apply:

$$\begin{aligned}
&\overbrace{\bigsqcup \{ \nu(q) \mid q \in \mathsf{After}_{\mathfrak{B}}(\epsilon) \}}^{\nu(\epsilon) \text{ as per Definition 3.2.2}} \\
&\quad \text{per (2.2)} \\
&= \bigsqcup \{ \nu(q) \mid q \in J \} \\
&\quad \text{per Definition 4.1.3} \\
&= \bigsqcup \{ \kappa(s) \sqcap \lambda(s) \mid s \in I \} \\
&\quad \text{per Definition 4.1.2} \\
&= \underbrace{\bigsqcup \{ \mathsf{V}(\rho) \mid \rho \in \mathsf{Runs}(\epsilon) \}}_{\mathsf{V}(\epsilon) \text{ as per (3.7)}}
\end{aligned}$$

This concludes the proof for the case $\omega = \epsilon$.

Case $\omega \neq \epsilon$. For the case $\omega \neq \epsilon$, we show (G1) by structural induction on ω. To this end, we first need to strengthen (G1) as follows:

$$= \overbrace{\mathsf{After}_{\mathfrak{B}}(\omega)}^{X(\omega)} \quad \underbrace{\left\{ \left\langle s'_n, \kappa(s_1) \sqcap \bigsqcap \{ \gamma(\mathrm{t}) \mid \langle \cdot, \mathrm{t} \rangle \in \rho \} \right\rangle \mid \rho = \left(\langle s_i, \alpha_i, s'_i \rangle \right)_{i=1}^{n} \in \mathsf{Runs}(\omega) \right\}}_{Y(\omega)} \tag{G2}$$

In the following, we refer to the constituents of (G2) by $X(\omega)$ and $Y(\omega)$ as indicated above. Now, if (G2), then the following transformations apply:

$$\begin{aligned}
&\overbrace{\bigsqcup \{ \nu(q) \mid q \in \mathsf{After}_{\mathfrak{B}}(\omega) \}}^{\nu(\omega) \text{ as per Definition 3.2.2}} \\
&\quad \text{per (G2) and Definition 4.1.3} \\
&= \bigsqcup \left\{ \kappa(s_1) \sqcap \left(\bigsqcap \{ \gamma(\mathrm{t}) \mid \langle \cdot, \mathrm{t} \rangle \in \rho \} \right) \sqcap \lambda(s'_n) \;\middle|\; \rho = \left(\langle s_i, \alpha_i, s'_i \rangle \right)_{i=1}^{n} \in \mathsf{Runs}(\omega) \right\} \\
&\quad \text{per Definition 4.1.2} \\
&= \underbrace{\bigsqcup \{ \mathsf{V}(\rho) \mid \rho \in \mathsf{Runs}(\epsilon) \}}_{\mathsf{V}(\epsilon) \text{ as per (3.7)}}
\end{aligned}$$

Thus, (G2) strengthens (G1) and it remains to show (G2). We proceed by structural induction. The base case is $\omega = a$ for some $a \in \mathrm{Act}$ as $\omega \neq \epsilon$.

Base Case $\omega = a$. The following transformations apply:

$$\overbrace{\mathsf{After}_{\mathfrak{B}}(a)}^{X(a)}$$

$$
\begin{aligned}
&\text{per (2.2) and (2.1)}\\
&= \{\langle s', v'\rangle \in \mathcal{Q} \mid \exists \langle\langle s, v\rangle, a, \langle s', v'\rangle\rangle \in \rightarrow, \langle s, v\rangle \in J\}\\
&\text{per Definition 4.1.3}\\
&= \{\langle s', v'\rangle \in \mathcal{Q} \mid \exists \langle s, a, s'\rangle \in \twoheadrightarrow, s \in I, v' = \kappa(s) \sqcap \gamma(\langle s, a, s'\rangle)\}\\
&\text{per (P1)}\\
&= \underbrace{\left\{\left\langle s', \kappa(s) \sqcap \bigsqcap \{\gamma(t) \mid \langle \cdot, t\rangle \in \rho\}\right\rangle \;\middle|\; \rho = \langle s, a, s'\rangle \in \mathsf{Runs}(a)\right\}}_{Y(a)}
\end{aligned}
$$

This concludes the base case of the induction.

Induction Step $\omega' = \omega \diamond a$. The following transformations apply:

$$
\begin{aligned}
&\overbrace{\mathsf{After}_{\mathfrak{B}}(\omega \diamond a)}^{X(\omega \diamond a)}\\
&\text{per (2.2) and (2.1)}\\
&= \{\langle s', v'\rangle \in \mathcal{Q} \mid \exists \langle\langle s, v\rangle, a, \langle s', v'\rangle\rangle \in \rightarrow, \langle s, v\rangle \in \mathsf{After}_{\mathfrak{B}}(\omega)\}\\
&\text{per the induction hypothesis}\\
&= \{\langle s', v'\rangle \in \mathcal{Q} \mid \exists \langle\langle s, v\rangle, a, \langle s', v'\rangle\rangle \in \rightarrow, \langle s, v\rangle \in Y(\omega)\}\\
&\text{per Definition 4.1.3}\\
&= \{\langle s', v \sqcap \gamma(\langle s, a, s'\rangle)\rangle \in \mathcal{Q} \mid \exists \langle s, a, s'\rangle \in \twoheadrightarrow, \langle s, v\rangle \in Y(\omega)\}\\
&\text{per (P1) and definition of } Y\\
&= \underbrace{\left\{\left\langle s'_n, \kappa(s_1) \sqcap \bigsqcap \{\gamma(t) \mid \langle \cdot, t\rangle \in \rho\}\right\rangle \;\middle|\; \rho = \left(\langle s_i, \alpha_i, s'_i\rangle\right)_{i=1}^{n} \in \mathsf{Runs}(\omega \diamond a)\right\}}_{Y(\omega \diamond a)}
\end{aligned}
$$

This concludes the structural induction and with it the proof of Theorem 4.1.1. □

A.2 Robust Continuous Time Verdictor Algorithm

A.2.1 Proof of Lemma 5.2.1

We restate Lemma 5.2.1 from the body of this thesis (p. 132).

Lemma 5.2.1 *For all observable actions $a \in \mathsf{OAct}$, observation times ${}^{\downarrow}t \in \mathbb{Q}$, occurrence times ${}^{\uparrow}t \in \mathbb{R}$, indices $j, i \in \mathbb{N}$, and time-remapping functions r as per Definition 5.1.1, the following condition holds:*

$$
{}^{\uparrow}t \in {}^{\uparrow}T_r(\langle j, \langle {}^{\downarrow}t, a\rangle\rangle) \iff {}^{\downarrow}t \in {}^{\downarrow}T_r(\langle i, \langle {}^{\uparrow}t, a\rangle\rangle) \tag{5.6}
$$

Proof. Lemma 5.2.1 is established by the following transformations:

$$
{}^{\uparrow}t \in {}^{\uparrow}T_r(\langle j, \langle {}^{\downarrow}t, a\rangle\rangle)
$$

$$
\begin{aligned}
&\text{per (5.5)}\\
\iff\ & r^{-1}\big({}^{\downarrow}t\big) - \mathsf{l}_{\max}(a) \le {}^{\uparrow}t \wedge {}^{\uparrow}t \le r^{-1}\big({}^{\downarrow}t\big) - \mathsf{l}_{\min}(a)\\
&\text{per elementary arithmetic}\\
\iff\ & r^{-1}\big({}^{\downarrow}t\big) \le {}^{\uparrow}t + \mathsf{l}_{\max}(a) \wedge {}^{\uparrow}t + \mathsf{l}_{\min}(a) \le r^{-1}\big({}^{\downarrow}t\big)\\
&\text{per Definition 5.1.1, } r \text{ is a bijection}\\
\iff\ & {}^{\downarrow}t \le r\big({}^{\uparrow}t + \mathsf{l}_{\max}(a)\big) \wedge r\big({}^{\uparrow}t + \mathsf{l}_{\min}(a)\big) \le {}^{\downarrow}t\\
&\text{per (5.4)}\\
\iff\ & {}^{\downarrow}t \in {}^{\downarrow}T_r\big(\langle i, \langle {}^{\uparrow}t, a\rangle\rangle\big)
\end{aligned}
$$

This concludes the proof of Lemma 5.2.1. □

A.2.2 Proof of Lemma 5.2.2

We restate Lemma 5.2.2 from the body of this thesis (p. 133).

Lemma 5.2.2 *Given the drift parameter δ (recall Definition 5.1.1), the following equation holds for any two timed observations $\langle j, \langle {}^{\downarrow}t, a\rangle\rangle$ and $\langle j', \langle {}^{\downarrow}t', a'\rangle\rangle$:*

$$
\begin{aligned}
&\mathsf{MaxD}\Big(\langle j, \langle {}^{\downarrow}t, a\rangle\rangle, \langle j', \langle {}^{\downarrow}t', a'\rangle\rangle\Big)\\
=\ & \mathsf{l}_{\max}(a') - \mathsf{l}_{\min}(a) + \begin{cases} (1+\delta)({}^{\downarrow}t - {}^{\downarrow}t') & \text{if } {}^{\downarrow}t \ge {}^{\downarrow}t'\\ (1+\delta)^{-1}({}^{\downarrow}t - {}^{\downarrow}t') & \text{otherwise}\end{cases}
\end{aligned}
$$

Proof. Let $\theta = \langle j, \langle {}^{\downarrow}t, a\rangle\rangle$ and $\theta' = \langle j', \langle {}^{\downarrow}t', a'\rangle\rangle$ be two timed observations. We start by establishing the following transformations:

$$
\begin{aligned}
&\mathsf{MaxD}(\theta, \theta')\\
&\text{per (5.7)}\\
=\ & \max_r \big(\max {}^{\uparrow}T_r(\theta) - \min {}^{\uparrow}T_r(\theta')\big)\\
&\text{per (5.5)}\\
=\ & \max_r \big(r({}^{\downarrow}t) - \mathsf{l}_{\min}(a) - \big(r({}^{\downarrow}t') - \mathsf{l}_{\max}(a')\big)\big)\\
&\text{per elementary arithmetic}\\
=\ & \mathsf{l}_{\max}(a') - \mathsf{l}_{\min}(a) + \max_r \big(r({}^{\downarrow}t) - r({}^{\downarrow}t')\big)
\end{aligned}
$$

Thus, it remains to show that:

$$
\max_r \big(r({}^{\downarrow}t) - r({}^{\downarrow}t')\big) = \begin{cases} (1+\delta)({}^{\downarrow}t - {}^{\downarrow}t') & \text{if } {}^{\downarrow}t \ge {}^{\downarrow}t'\\ (1+\delta)^{-1}({}^{\downarrow}t - {}^{\downarrow}t') & \text{otherwise}\end{cases} \tag{G1}
$$

To show (G1), let us distinguish the cases ${}^{\downarrow}t \geq {}^{\downarrow}t'$ and ${}^{\downarrow}t < {}^{\downarrow}t'$.

Case ${}^{\downarrow}t \geq {}^{\downarrow}t'$. We also have $r({}^{\downarrow}t) \geq r({}^{\downarrow}t')$ for all r since time-remapping functions are strictly monotone (see Definition 5.1.1). Hence, $r({}^{\downarrow}t) - r({}^{\downarrow}t') \geq 0$ for all r. Now, it follows from the slope restriction (5.2) that $\max_r \left(r({}^{\downarrow}t) - r({}^{\downarrow}t')\right) = (1+\delta)({}^{\downarrow}t - {}^{\downarrow}t')$ concluding the proof for the case ${}^{\downarrow}t \geq {}^{\downarrow}t'$.

Case ${}^{\downarrow}t < {}^{\downarrow}t'$. We also have $r({}^{\downarrow}t) < r({}^{\downarrow}t')$ for all r since time-remapping functions are strictly monotone (see Definition 5.1.1). Hence, $r({}^{\downarrow}t) - r({}^{\downarrow}t') < 0$ for all r. Now, it follows from the slope restriction (5.2) that $\max_r \left(r({}^{\downarrow}t) - r({}^{\downarrow}t')\right) = (1+\delta)^{-1}({}^{\downarrow}t - {}^{\downarrow}t')$ concluding the proof for the case ${}^{\downarrow}t < {}^{\downarrow}t'$. □

A.2.3 Proof of Lemma 5.2.3

We restate Lemma 5.2.3 from the body of this thesis (p. 138).

Lemma 5.2.3 *Assume given an observation sequence ω' consistent with some run ρ', i.e., $\varrho(\rho') \rhd \omega'$. For all prefixes $\omega \in \mathsf{Pref}(\omega')$ of the observation sequence ω' there exists a prefix $\rho \in \mathsf{Pref}(\rho')$ of the run ρ' such that $\omega \in \Omega(\rho)$ where Ω is the observation model as per Definition 5.2.4.*

Proof. Let ω' be an observation sequence, ρ' be a run consistent with ω', and $\omega \in \mathsf{Pref}(\omega')$ be a prefix of ω'. We prove Lemma 5.2.3 constructively by defining the prefix $\rho \in \mathsf{Pref}(\rho')$. To this end, let ρ be the minimal prefix of ρ' such that

$$\varrho(\rho) \supseteq \{\, e \in \varrho(\rho') \mid \mathcal{R}(e) \in \omega \,\} \tag{P1}$$

where $\mathcal{R}$ is the bijection witnessing $\varrho(\rho') \rhd \omega'$. Condition (P1) requires that ρ contains all events that have been observed on ω. Note that such a prefix must exist as ρ' is a prefix of itself and it fulfills condition (P1). Hence, it remains to show that the prefix ρ indeed satisfies $\omega \in \Omega(\rho)$. Applying the definition of Ω (see Definition 5.2.4), we obtain the following remaining proof goals:

$$\exists \omega'' \geq \omega : \varrho(\rho) \rhd \omega'' \tag{G1}$$

$$\exists \varsigma \in \mathit{\Pi}(\rho) : \mathsf{Word}(\varsigma) \rhd \omega \tag{G2}$$

We prove both goals separately.

Goal (G1). Let $\omega'' \coloneqq \mathsf{Word}(\{\, \mathcal{R}(e) \mid e \in \rho \,\})$. We have $\omega' \geq \omega'' \geq \omega$ as ρ contains all events corresponding to the observations in ω according to (P1). Furthermore, we have $\rho \rhd \omega''$ as ω'' contains exactly the observations corresponding to the events that ocurred on ρ and the same time-remapping function r witnessing $\varrho(\rho') \rhd \omega'$ also witnesses $\rho \rhd \omega''$. This concludes the proof of (G1).

Goal (G2). To prove (G2), we choose $\varsigma \coloneqq \{\, e \in \varrho(\rho) \mid \mathcal{R}(e) \in \omega \,\}$, i.e., ς must contain exactly the events that have been observed on ω. Again, it is easy to see that $\mathrm{Word}(\varsigma) \rhd \omega$ because ς contains exactly the events which have been observed on ω and the same time-remapping function r witnessing $\varrho(\rho') \rhd \omega'$ also witnesses $\mathrm{Word}(\varsigma) \rhd \omega$. Thus, it remains to show that $\varsigma \in \Pi(\rho)$, i.e.:

$$\{\, e \in \varrho(\rho) \mid \mathcal{R}(e) \in \omega \,\} \in \Pi(\rho) \tag{P2}$$

To this end, we must show:

$$\forall \left\langle \cdot, \left\langle {}^{\uparrow}t, a \right\rangle \right\rangle \in \{\, e \in \varrho(\rho) \mid \mathcal{R}(e) \notin \omega \,\} : \mathrm{Dur}(\rho) \leq {}^{\uparrow}t + \mathsf{l}_{\max}(a) \tag{G3}$$

Condition (G3) requires that all events which have not been observed on ω also do not need to be observed according to the definition of $\Pi(\rho)$, see (5.12). Towards a proof of (G3) by contradiction, let us first establish:

$$\mathrm{Dur}(\rho) \leq \max\left\{\, {}^{\uparrow}t \mid \mathcal{R}\left(\left\langle \cdot, \left\langle {}^{\uparrow}t, \cdot \right\rangle \right\rangle\right) \in \omega \,\right\} \tag{P3}$$

Note that (P3) follows from the fact that ρ is minimal. If (P3) were not to hold, then we could remove the last element of the run and still satisfy condition (P1), i.e., in this case ρ would not be the minimal prefix satisfying (P1). This is the case because if (P3) is not the case, then additional time passed after the events corresponding to ω occurred, which makes the prefix non-minimal.

Now, assume for the sake of contradiction that (G3) does not hold. Hence, there exists an event $e = \left\langle i, \left\langle {}^{\uparrow}t, a \right\rangle \right\rangle \in \varrho(\rho)$ such that $\theta = \mathcal{R}(e) \notin \omega$ but $\mathrm{Dur}(\rho) > {}^{\uparrow}t + \mathsf{l}_{\max}(a)$. Applying (P3), we further obtain the following inequalities:

$${}^{\uparrow}t + \mathsf{l}_{\max}(a) < \mathrm{Dur}(\rho) \leq \max\left\{\, {}^{\uparrow}t \mid \mathcal{R}\left(\left\langle \cdot, \left\langle {}^{\uparrow}t, \cdot \right\rangle \right\rangle\right) \in \omega \,\right\}$$

Hence, we in particular obtain the following inequality:

$${}^{\uparrow}t + \mathsf{l}_{\max}(a) < \max \underbrace{\left\{\, {}^{\uparrow}t \mid \mathcal{R}\left(\left\langle \cdot, \left\langle {}^{\uparrow}t, \cdot \right\rangle \right\rangle\right) \in \omega \,\right\}}_{(*)}$$

Let $e' = \left\langle \cdot, \left\langle {}^{\uparrow}t', a' \right\rangle \right\rangle$ with $\left\langle \cdot, \left\langle {}^{\downarrow}t', a' \right\rangle \right\rangle = \mathcal{R}(e') \in \omega$ be the event corresponding to the maximum of the occurrence times in the set marked with $(*)$ in the inequality above. For the occurrence time ${}^{\uparrow}t'$ of this event, we have:

$${}^{\uparrow}t + \mathsf{l}_{\max}(a) < {}^{\uparrow}t' \tag{P4}$$

The contradiction proving (G3) is obtained by establishing that ${}^{\downarrow}t < {}^{\downarrow}t'$. Thus, $\theta \in \omega$ but $\theta \notin \omega$. Recall that $\varrho(\rho') \rhd \omega'$, hence, we have:

$${}^{\downarrow}t \in {}^{\downarrow}T_r(e) \wedge {}^{\downarrow}t' \in {}^{\downarrow}T_r(e')$$

per (5.4)

$$\implies {}^{\downarrow}t \leq r^{-1}({}^{\uparrow}t + \mathsf{l}_{\max}(a)) \wedge r^{-1}({}^{\uparrow}t' + \mathsf{l}_{\min}(a')) \leq {}^{\downarrow}t'$$

per (P4) and monotonicity of r

$$\implies {}^{\downarrow}t \leq r^{-1}({}^{\uparrow}t + \mathsf{l}_{\max}(a)) < r^{-1}({}^{\uparrow}t' + \mathsf{l}_{\min}(a')) \leq {}^{\downarrow}t'$$

per elementary arithmetic

$$\implies {}^{\downarrow}t < {}^{\downarrow}t'$$

Here, r is the time-remapping function witnessing $\varrho(\rho') \rhd \omega'$. This concludes the proof of (G3) by contradiction and thus the proof of Lemma 5.2.3. □

A.2.4 Proof of Theorem 5.3.1

We restate Theorem 5.3.1 from the body of this thesis (p. 140).

Theorem 5.3.1 *Every VTS $\mathfrak{V}$ satisfying (5.18) is sound.*

Proof. Theorem 5.3.1 states that $\mathsf{V}(\rho') \sqsubseteq \nu(\omega')$ for all $\omega' \in \mathcal{L}\restriction_{\Omega}([\![\mathfrak{T}]\!])$ and $\rho' \in \mathsf{Runs}(\omega')$. Let $\omega' \in \mathcal{L}\restriction_{\Omega}([\![\mathfrak{T}]\!])$ and $\rho' \in \mathsf{Runs}(\omega')$. The proof goal is:

$$\mathsf{V}(\rho') \sqsubseteq \nu(\omega') \tag{G1}$$

Instead of proving (G1) directly, we will prove the following proposition

$$\exists \rho \in \mathsf{Pref}(\rho') : \exists \varsigma \in A(\omega') : \varrho(\rho) \rhd \mathsf{Word}(\varsigma) \tag{G2}$$

echoing (5.18) where $A(\omega')$ is the set of active subsets of ω' as per (5.17). The proof obligation (G2) implies the proof obligation (G1):

$$\overbrace{\exists \rho \in \mathsf{Pref}(\rho') : \exists \varsigma \in A(\omega') : \varrho(\rho) \rhd \mathsf{Word}(\varsigma)}^{\text{(G2)}}$$

per (5.18)

$$\implies \exists \rho \in \mathsf{Pref}(\rho') : \mathsf{V}(\rho) \sqsubseteq \nu(\omega')$$

per Proposition 5.0.1

$$\implies \underbrace{\mathsf{V}(\rho') \sqsubseteq \nu(\omega')}_{\text{(G1)}}$$

The last step uses the fact that $\forall \rho \in \mathsf{Pref}(\rho') : \mathsf{V}(\rho') \sqsubseteq \mathsf{V}(\rho)$, since the verdict oracle is monotonic (see Definition 3.4.2 and Proposition 5.0.1).

It remains to show (G2). Given that $\rho' \in \mathsf{Runs}(\omega')$, we have $\omega' \in \Omega(\rho')$ which according to Definition 5.2.4 means that there exists a $\omega'' \geq \omega'$ such that $\varrho(\rho') \rhd \omega''$, i.e., ω' can be continued such that $\varrho(\rho')$ is consistent with the continuation. From $\varrho(\rho') \rhd \omega''$, we obtain witnesses $\mathcal{R}'$ and r according to Definition 5.2.3, i.e., $\mathcal{R}' : \varrho(\rho') \to \omega''$ is the bijection and r is the time-remapping function witnessing

consistency of $\varrho(\rho')$ and ω''. We establish (G2) by first constructing a prefix ρ of ρ' and then showing that $\varrho(\rho) \rhd \mathrm{Word}(\varsigma)$ for some $\varsigma \in A(\omega')$ thereby instantiating the existential quantification in (G2). Let ρ be the maximal prefix of ρ' such that:

$$\{\mathcal{R}'(e) \mid e \in \varrho(\rho)\} \subseteq \omega'$$

That is, ρ is the maximal prefix that contains only events observed on ω'. To show the existential quantification in (G2), let ς be defined as follows:

$$\varsigma \coloneqq \{\mathcal{R}'(e) \mid e \in \varrho(\rho)\} \tag{P1}$$

That is, ς contains exactly the observations of the events induced by ρ.

This leaves us with the following remaining proof goals:

$$\varrho(\rho) \rhd \mathrm{Word}(\varsigma) \tag{G3}$$

$$\varsigma \in A(\omega') \tag{G4}$$

We prove both goals separately.

Goal (G3). Consistency of $\varrho(\rho)$ and $\mathrm{Word}(\varsigma)$ is witnessed by the functions r and $\mathcal{R} : \varrho(\rho) \to \omega$ with $\mathcal{R}(e) \coloneqq \mathcal{R}'(e)$, where r and $\mathcal{R}'$ are the witnesses of $\varrho(\rho') \rhd \omega''$ as described above. This establishes $\varrho(\rho) \rhd \mathrm{Word}(\varsigma)$ for the prefix ρ.

Goal (G4). Applying (5.17), the proof goal becomes:

$$\mathrm{Settled}(\omega') \subseteq \varsigma \subseteq \omega'$$

By definition (P1) we have $\varsigma \subseteq \omega'$. Thus, it remains to show that:

$$\mathrm{Settled}(\omega') \subseteq \varsigma$$

Suppose for the sake of contradiction that $\mathrm{Settled}(\omega') \not\subseteq \varsigma$, i.e., there exists an observation $\theta \in \mathrm{Settled}(\omega')$ but $\theta \notin \varsigma$. We derive a contradiction by showing that under this assumption ρ is not maximal: Let e be the event corresponding to θ according to $\mathcal{R}'$, i.e., $\theta = \mathcal{R}'(e)$. Note that this event is indeed uniquely defined since $\mathcal{R}'$ is a bijection. Recall that $\theta \in \mathrm{Settled}(\omega')$ means that the event e corresponding to θ must have occurred before any event observed in the future (cf. Section 5.3.1). Hence, all events $e' \in \varrho(\rho')$ occurring before e on ρ' must have already been observed. Formally, that is $\mathcal{R}'(e') \in \omega'$ for all those events e'. Hence, ρ can be extended to contain the already observed events occurring before e on ρ' as well as e itself, i.e., ρ is not maximal. This concludes the contradiction and the proof of Theorem 5.3.1. □

A.2.5 Proof of Theorem 5.3.3

We restate Theorem 5.3.3 from the body of this thesis (p. 141).

Theorem 5.3.3 *Every VTS $\mathfrak{V}$ satisfying (5.21) is sound.*

Proof. The proof proceeds completely analogously to the proof of Theorem 5.3.1 with $\mathsf{Pref}_A(\omega')$ in place of $A(\omega')$ until the proof of the goal (G4). Instead of proving $\varsigma \in A(\omega')$ we must now prove:

$$\varsigma \in \mathsf{Pref}_A(\omega')$$

Applying the definition (5.20) of Pref_A, we obtain two proof goals:

$$\varsigma \in A(\omega') \tag{G1}$$
$$\varsigma \in \mathsf{Pref}_{\prec}(\omega') \tag{G2}$$

Recall, that (G1) has already been established as part of the proof of Theorem 5.3.1. Hence, it remains to prove (G2), i.e., $\varsigma \in \mathsf{Pref}_{\prec}(\omega')$. Applying the definition (5.19) of $\mathsf{Pref}_{\prec}$, the proof goal (G2) becomes:

$$\forall \theta \in \varsigma : \forall \theta' \in \omega' : \theta' \prec \theta \Longrightarrow \theta' \in \varsigma$$

Let $\theta \in \varsigma$ and $\theta' \in \omega'$ such that $\theta' \prec \theta$. Thereby, the proof obligation becomes $\theta' \in \varsigma$. Let e and e' be the events corresponding to the observations θ and θ', i.e., $\theta = \mathcal{R}'(e)$ and $\theta' = \mathcal{R}'(e')$.[40] As $\varsigma(\rho')$ is consistent with ω'' and $\theta' \prec \theta$, the event e' must occur before the event e on ρ'. Hence, the prefix ρ of ρ' must contain the event e' because it contains the event e. By definition (P1) from the inherited proof context of the proof of Theorem 5.3.1, ς contains all the observations corresponding to the events on ρ. Therefore, ς also contains the observation θ'. This concludes the proof of (G2) and with it the proof of Theorem 5.3.3. □

A.2.6 Proof of Theorem 5.3.2

We restate Theorem 5.3.2 from the body of this thesis (p. 140).

Theorem 5.3.2 *Every VTS $\mathfrak{V}$ satisfying (5.18) is Δ-complete for Δ as per (5.14).*

Proof. According to the definition of Δ-completeness (see Definition 3.4.5), given an observation sequence ω, we must show $\nu(\omega') \sqsubseteq \mathsf{V}(\omega)$ for all continuations $\omega' \in \mathsf{Cont}(\omega)$, for which $d(\omega', \omega) > \Delta$ where d is the time difference between ω' and ω according to (5.15). Therefore, for an observation sequence $\omega \in \mathcal{L}|_\Omega([\![\mathfrak{T}]\!])$ and a continuation $\omega' \in \mathsf{Cont}(\omega)$ such that $d(\omega', \omega) > \Delta$, the proof goal is:

$$\nu(\omega') \sqsubseteq \mathsf{V}(\omega) \tag{G1}$$

[40] By starting the proof analogously to the proof of Theorem 5.3.1, we inherit its proof context. Here, $\mathcal{R}'$ is the bijection witnessing $\varsigma(\rho') \rhd \omega''$ as introduced in the proof of Theorem 5.3.1.

Instead of proving (G1) directly, we will prove the following proposition

$$\begin{aligned}&\forall \varsigma \in A(\omega') : \forall \rho'' \in \mathsf{Runs}([\![\mathfrak{T}]\!]) \text{ s.t. } \varrho(\rho'') \rhd \mathsf{Word}(\varsigma) : \\ &\quad \exists \rho \in \mathsf{Pref}(\rho'') : \rho \in \mathsf{Runs}(\omega)\end{aligned} \tag{G2}$$

echoing (5.18) where $A(\omega')$ is the set of active subsets of ω' as per (5.17). The proof obligation (G2) implies the proof obligation (G1):

$$\overbrace{\begin{aligned}&\forall \varsigma \in A(\omega') : \forall \rho'' \in \mathsf{Runs}([\![\mathfrak{T}]\!]) \text{ s.t. } \varrho(\rho'') \rhd \mathsf{Word}(\varsigma) : \\ &\quad \exists \rho \in \mathsf{Pref}(\rho'') : \rho \in \mathsf{Runs}(\omega)\end{aligned}}^{\text{(G2)}}$$

per elementary first-order logic and set theory

$$\iff \begin{aligned}&\forall \rho'' \in \{\, \rho'' \in \mathsf{Runs}([\![\mathfrak{T}]\!]) \mid \varsigma \in A(\omega') \wedge \varrho(\rho'') \rhd \mathsf{Word}(\varsigma) \,\} : \\ &\quad \exists \rho \in \mathsf{Pref}(\rho'') : \rho \in \mathsf{Runs}(\omega)\end{aligned}$$

per Proposition 5.0.1

$$\implies \begin{aligned}&\Big(\bigsqcup \{\, \mathsf{V}(\rho'') \mid \rho'' \in \mathsf{Runs}([\![\mathfrak{T}]\!]) \wedge \varsigma \in A(\omega') \wedge \varrho(\rho'') \rhd \mathsf{Word}(\varsigma) \,\}\Big) \\ &\quad \sqsubseteq \Big(\bigsqcup \{\, \mathsf{V}(\rho) \mid \rho \in \mathsf{Runs}(\omega) \,\}\Big)\end{aligned}$$

per (5.18) and (3.7)

$$\iff \underbrace{\nu(\omega') \sqsubseteq \mathsf{V}(\omega)}_{\text{(G1)}}$$

The last step uses the fact that $\forall \rho \in \mathsf{Pref}(\rho'') : \mathsf{V}(\rho'') \sqsubseteq \mathsf{V}(\rho)$, since the verdict oracle is monotonic (see Definition 3.4.2 and Proposition 5.0.1).

Hence, it remains to show (G2). To this end, let $\varsigma \in A(\omega')$ and $\rho'' \in \mathsf{Runs}([\![\mathfrak{T}]\!])$ such that $\varrho(\rho'') \rhd \mathsf{Word}(\varsigma)$. The proof goal becomes:

$$\exists \rho \in \mathsf{Pref}(\rho'') : \rho \in \mathsf{Runs}(\omega) \tag{G3}$$

Since $\omega' \in \mathsf{Cont}(\omega)$, we have $\omega' \geq \omega$. Since $\omega' \geq \omega$, $\varsigma \in A(\omega')$, and $d(\omega', \omega) > \Delta$, we have $\mathsf{Word}(\varsigma) \geq \omega$, i.e., ς contains all observations of ω as they are settled by now. The following deduction completes the proof:

$$\rho'' \rhd \mathsf{Word}(\varsigma) \wedge \omega \leq \mathsf{Word}(\varsigma)$$

per Lemma 5.2.3

$$\implies \exists \rho \in \mathsf{Pref}(\rho'') : \omega \in \Omega(\rho)$$

per prefix-closure of runs

$$\implies \underbrace{\exists \rho \in \mathsf{Pref}(\rho'') : \rho \in \mathsf{Runs}(\omega)}_{\text{(G3)}}$$

Hence, we established Theorem 5.3.2. □

A.2.7 Proof of Theorem 5.3.4

We restate Theorem 5.3.4 from the body of this thesis (p. 142).

Theorem 5.3.4 *A timed event sequence ϱ and a timed observation sequence ω are consistent iff they are bound-consistent.*

Proof. For a timed event sequence $\varrho = \left(\left\langle {}^{\uparrow}t_i, a_i \right\rangle\right)_{i=1}^{n}$ and a timed observation sequence $\omega = \left(\left\langle {}^{\downarrow}t_j, b_j \right\rangle\right)_{j=1}^{n}$ the proof goal of Theorem 5.3.4 is:

$$\varrho \text{ is consistent with } \omega \iff \varrho \text{ is bound-consistent with } \omega \tag{G1}$$

We prove both directions of the equivalence separately.

ϱ is consistent with ω $\implies$ ϱ is bound-consistent with ω

Let $\mathcal{R} : \varrho \to \omega$ be the bijection and r be the time-remapping function witnessing consistency of ϱ and ω. For each $e = \left\langle i, \left\langle {}^{\uparrow}t, a \right\rangle \right\rangle \in \varrho$, we know ${}^{\uparrow}t \in {}^{\uparrow}T_r(\mathcal{R}(e))$ because ϱ and ω are consistent (cf. Definition 5.2.3). We choose the same bijection $\mathcal{R}$ to prove bound consistency. Recall that the definition of bound consistency comprises two conditions (i) and (ii) (see Definition 5.3.1). Condition (i) requires that $a = b$ for all $\left\langle \cdot, \left\langle {}^{\uparrow}t, a \right\rangle \right\rangle \in \varrho$ where $\left\langle \cdot, \left\langle {}^{\downarrow}t, b \right\rangle \right\rangle = \mathcal{R}\left(\left\langle \cdot, \left\langle {}^{\uparrow}t, a \right\rangle \right\rangle\right)$. It is trivially satisfied by $\mathcal{R}$ as it directly follows from the conditions required by consistency (see Definition 5.2.3). Therefore, condition (ii) remains as proof obligation:

$$\begin{aligned} &\forall \left\langle i, \left\langle {}^{\uparrow}t, a \right\rangle \right\rangle, \left\langle i', \left\langle {}^{\uparrow}t', a' \right\rangle \right\rangle \in \varrho : \\ &\quad {}^{\uparrow}t - {}^{\uparrow}t' \leq \mathrm{MaxD}\left(\mathcal{R}\left(\left\langle i, \left\langle {}^{\uparrow}t, a \right\rangle \right\rangle\right), \mathcal{R}\left(\left\langle i', \left\langle {}^{\uparrow}t', a' \right\rangle \right\rangle\right)\right) \end{aligned} \tag{G2}$$

For proving (G2), let $e = \left\langle i, \left\langle {}^{\uparrow}t, a \right\rangle \right\rangle \in \varrho$ and $e' = \left\langle i', \left\langle {}^{\uparrow}t', a' \right\rangle \right\rangle \in \varrho$. We already established that ${}^{\uparrow}t \in {}^{\uparrow}T_r(\mathcal{R}(e))$ and ${}^{\uparrow}t' \in {}^{\uparrow}T_r(\mathcal{R}(e'))$, hence, ${}^{\uparrow}t - {}^{\uparrow}t' \in {}^{\uparrow}T_r(\mathcal{R}(e)) \boxminus {}^{\uparrow}T_r(\mathcal{R}(e'))$. Therefore, by elementary interval arithmetic (cf. Chapter 2) and maximization over all time-remapping functions we establish:

$$\begin{aligned} & {}^{\uparrow}t - {}^{\uparrow}t' \in {}^{\uparrow}T_r(\mathcal{R}(e)) \boxminus {}^{\uparrow}T_r(\mathcal{R}(e')) \\ \implies & {}^{\uparrow}t - {}^{\uparrow}t' \leq \max\left({}^{\uparrow}T_r(\mathcal{R}(e)) \boxminus {}^{\uparrow}T_r(\mathcal{R}(e'))\right) \\ \implies & {}^{\uparrow}t - {}^{\uparrow}t' \leq \max {}^{\uparrow}T_r(\mathcal{R}(e)) - \min {}^{\uparrow}T_r(\mathcal{R}(e')) \\ \implies & {}^{\uparrow}t - {}^{\uparrow}t' \leq \underbrace{\max_{r'} \left(\max {}^{\uparrow}T_{r'}(\mathcal{R}(e)) - \min {}^{\uparrow}T_{r'}(\mathcal{R}(e'))\right)}_{= \mathrm{MaxD}(\mathcal{R}(e), \mathcal{R}(e')) \text{ as per (5.7)}} \end{aligned} \tag{P1}$$

The crux here is that the specific r witnessing consistency can only lead to a bound tighter than (P1) because (P1) maximizes over all time-remapping functions. This concludes the proof of the $\implies$ direction of (G1).

ϱ is consistent with ω $\impliedby$ ϱ is bound-consistent with ω

The proof of the other direction is much more involved, as we have to show that there always exists a time-remapping function r satisfying the required constraints, if ϱ and ω are bound-consistent. To this end, let $\mathcal{R} : \varrho \to \omega$ be the bijection witnessing bound consistency. We choose the same bijection $\mathcal{R}$ to prove consistency. Condition (i) of bound consistency (cf. Definition 5.3.1) ensures that the actions of events and corresponding observations match. Therefore, it remains to show that there exists a time-remapping function r such that:

$$\forall \langle {}^{\uparrow}t, i, \alpha \rangle \in \varrho : {}^{\uparrow}t \in {}^{\uparrow}T_r\big(\mathcal{R}(\langle {}^{\uparrow}t, i, \alpha \rangle)\big) \tag{G3}$$

We prove the existence of r constructively. That is, we provide an explicit construction of such a time-remapping function r. The time-remapping function we construct is piecewise linear as schematically shown in Figure A.1. To this end, we proceed as follows: For each observation time ${}^{\downarrow}t$ of an observation to be found in ω, we construct a value for $r({}^{\downarrow}t)$ such that (G3) and the slope restriction (5.2) are satisfied. Between the observation times (yellow in Figure A.1) and before the first (respectively after the last) observation time (green in Figure A.1), the values are linearly interpolated, thereby providing a time-remapping function. Figure A.1 depicts the general idea ignoring that observations may be made at the same time.

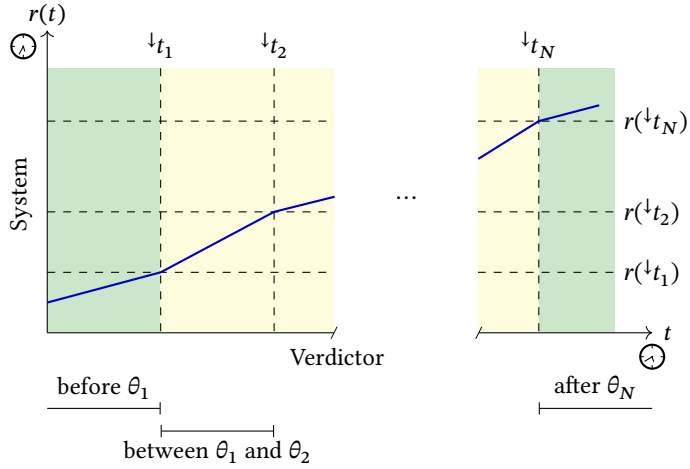


Figure A.1: Schematic depiction of the piecewise-linear time-remapping function r that we construct in the following.

The construction of r proceeds in three steps: (A) We construct an interval $I({}^{\downarrow}t)$ for each observation time ${}^{\downarrow}t$ such that $r({}^{\downarrow}t) \in I({}^{\downarrow}t)$ is sufficient for satisfying (G3). (B) We narrow $I({}^{\downarrow}t)$, to a certain degree which is necessary for the proof to work, by taking into account the slope restriction (5.2). This results in yet another interval

$\bar{r}(\downarrow t)$ for each observation time. (C) Finally, we construct a family of concrete time-remapping functions r (as described above) and prove that each posses the necessary properties, that is, it satisfies (G3) as well as Definition 5.1.1. Along the way, we always prove that the constructed intervals are non-empty as this is crucial for the existence of the final time-remapping function.

(A) Constructing $I(\downarrow t)$ for each observation time $\downarrow t$.

For the purpose of constructing $I(\downarrow t)$, let $E(\downarrow t) \subseteq \varrho$ be the set of all events whose observations are made at time $\downarrow t$. Formally, that is:

$$E(\downarrow t) \coloneqq \{\, e \in \varrho \mid \mathcal{R}(e) = \langle \downarrow t, \cdot \rangle \,\} \tag{P2}$$

Further, let $T_o \subseteq \mathbb{Q}_0^+$ be the set of all observation times. Formally:

$$T_o \coloneqq \{\, \downarrow t \in \mathbb{Q}_0^+ \mid E(\downarrow t) \neq \varnothing \,\} \tag{P3}$$

That is, T_o contains the observation times for which the set of events observed at these times is non-empty. The time-remapping function we construct is piecewise-linear between these times (cf. Figure A.1). For each observation time $\downarrow t \in T_o$ we define the interval $I(\downarrow t) \subseteq \mathbb{R}$ as follows:

$$I(\downarrow t) \coloneqq \bigcap_{\langle i, \langle \uparrow t, a \rangle \rangle \in E(\downarrow t)} \left[\uparrow t + \mathsf{l}_{\min}(a), \uparrow t + \mathsf{l}_{\max}(a) \right] \tag{P4}$$

The same observation times must be mapped to the same system time by r, hence, we define the interval $I(\downarrow t)$ for each observation time $\downarrow t \in T_o$ by intersecting the possible times at which the observations for the corresponding events may be observed relative to the systems clock by intersecting the respective intervals.

Now, $\forall \downarrow t \in T_o : r(\downarrow t) \in I(\downarrow t)$ is sufficient for satisfying (G3):

$$\begin{aligned}
&\forall \downarrow t \in T_o : r(\downarrow t) \in I(\downarrow t) \\
&\quad \text{per (P4)} \\
\iff &\forall \downarrow t \in T_o : r(\downarrow t) \in \bigcap_{\langle i, \langle \uparrow t, a \rangle \rangle \in E(\downarrow t)} \left[\uparrow t + \mathsf{l}_{\min}(a), \uparrow t + \mathsf{l}_{\max}(a) \right] \\
&\quad \text{per elementary arithmetic} \\
\implies &\forall \downarrow t \in T_o : \forall \langle i, \langle \uparrow t, a \rangle \rangle \in E(\downarrow t) : \uparrow t \in [r(\downarrow t) - \mathsf{l}_{\max}(a), r(\downarrow t) - \mathsf{l}_{\min}(a)] \\
&\quad \text{per (5.5), (P2), and (P3)} \\
\implies &\underbrace{\forall \langle i, \langle \uparrow t, a \rangle \rangle \in \varrho : \uparrow t \in \uparrow T_r(\mathcal{R}(\langle i, \langle \uparrow t, a \rangle \rangle))}_{\text{(G3)}}
\end{aligned}$$

Next, we need to show that $I(\downarrow t)$ is non-empty for all $\downarrow t \in T_o$. Otherwise, there is no r such that $\forall \downarrow t \in T_o : r(\downarrow t) \in I(\downarrow t)$ and the construction would fail. This leaves us with the following intermediate proof obligation:

$$\forall \downarrow t \in T_o : I(\downarrow t) \neq \varnothing \tag{G4}$$

Let ${}^{\downarrow}t \in T_o$. There exist $\langle i, \langle {}^{\uparrow}t, a\rangle\rangle, \langle i', \langle {}^{\uparrow}t', a'\rangle\rangle \in E({}^{\downarrow}t)$ such that:

$$\left[{}^{\uparrow}t + \mathsf{l}_{\mathsf{min}}(a), {}^{\uparrow}t' + \mathsf{l}_{\mathsf{max}}(a')\right] = I({}^{\downarrow}t) \tag{P5}$$

This is the case because the lower and upper bounds of $I({}^{\downarrow}t)$ are the maximum and minimum of ${}^{\uparrow}t + \mathsf{l}_{\mathsf{min}}(a)$ and ${}^{\uparrow}t + \mathsf{l}_{\mathsf{max}}(a)$ over $\langle i, \langle {}^{\uparrow}t, a\rangle\rangle \in E({}^{\downarrow}t)$, respectively. Hence, $I({}^{\downarrow}t) \neq \varnothing$ if and only if ${}^{\uparrow}t + \mathsf{l}_{\mathsf{min}}(a) \leq {}^{\uparrow}t' + \mathsf{l}_{\mathsf{max}}(a')$ for the events $\langle i, \langle {}^{\uparrow}t, a\rangle\rangle$ and $\langle i', \langle {}^{\uparrow}t', a'\rangle\rangle$. Thereby, the intermediate proof obligation (G4) becomes:

$${}^{\uparrow}t + \mathsf{l}_{\mathsf{min}}(a) \leq {}^{\uparrow}t' + \mathsf{l}_{\mathsf{max}}(a') \tag{G5}$$

We show (G5) by exploiting the fact that ϱ and ω are bound-consistent:

per bound consistency of ϱ and ω

$${}^{\uparrow}t - {}^{\uparrow}t' \leq \mathsf{MaxD}\big(\mathcal{R}(\langle i, \langle {}^{\uparrow}t, a\rangle\rangle), \mathcal{R}(\langle i', \langle {}^{\uparrow}t', a'\rangle\rangle)\big)$$

per Lemma 5.2.2

$$\implies {}^{\uparrow}t - {}^{\uparrow}t' \leq \mathsf{l}_{\mathsf{max}}(a') - \mathsf{l}_{\mathsf{min}}(a)$$

per elementary arithmetic

$$\iff \underbrace{{}^{\uparrow}t + \mathsf{l}_{\mathsf{min}}(a) \leq {}^{\uparrow}t' + \mathsf{l}_{\mathsf{max}}(a')}_{\text{(G5)}}$$

Note that the application of Lemma 5.2.2 above rests on the fact that the observation times, ${}^{\downarrow}t$ and ${}^{\downarrow}t'$, of the observations corresponding to the two events $\langle i, \langle {}^{\uparrow}t, a\rangle\rangle$ and $\langle i', \langle {}^{\uparrow}t', a'\rangle\rangle$ are identical, hence, $(1+\delta)({}^{\downarrow}t - {}^{\downarrow}t') = 0$.

This leaves us with a non-empty interval $I({}^{\downarrow}t)$ for each observation time ${}^{\downarrow}t$ such that $r({}^{\downarrow}t) \in I({}^{\downarrow}t)$ is sufficient for (G3). However, for constructing r one cannot just choose arbitrary values from these intervals since the slope restriction (5.2) must also be satisfied for a time-remapping function witnessing consistency.

(2.) Constructing $\bar{r}({}^{\downarrow}t)$ for each observation time by narrowing $I({}^{\downarrow}t)$.

In the next step, we narrow $I({}^{\downarrow}t)$ by taking into account the slope restriction (5.2). To this end, let $\vec{t} = t_1 \cdots t_{|T_o|}$ be the permutation of T_o such that $t_k < t_{k+1}$ for all $1 \leq k < |T_o|$. We construct an interval $\bar{r}(t_k) \subseteq \mathbb{R}$ for each observation time $t_k \in T_o$ such that $\bar{r}(t_k) \subseteq I(t_k)$ is inductively defined as follows:

$$\bar{r}(t_1) \coloneqq I(t_1) \qquad \bar{r}(t_{k+1}) \coloneqq I(t_{k+1}) \cap \Bigg(\bar{r}(t_k) \boxplus \underbrace{\left[\frac{t_{k+1} - t_k}{1+\delta}, (1+\delta)(t_{k+1} - t_k)\right]}_{(*)}\Bigg) \tag{P6}$$

Here, (*) is based on the slope restriction (5.2) and projects a cone (cf. Figure 5.3b) from the predecessor interval $\bar{r}(t_k)$. It is easy to see that $\bar{r}(t_k) \subseteq I(t_k)$ for all t_k as it is ensured by the intersection in (P6). We again proceed by showing that $\bar{r}(t_k) \neq \varnothing$

for all t_k. For $k = 1$ this is trivial because $\bar{r}(t_1) = I(t_1)$ and we already established $I(t_1) \neq \emptyset$ previously. Thus, for $k > 1$, it remains to show:

$$\bar{r}(t_k) \neq \emptyset \tag{G6}$$

Let us start with the following observations for $1 \leq k < |T_o|$ leveraging elementary interval arithmetic and set theory:

$$\min \bar{r}(t_{k+1}) = \max \left\{ \begin{array}{c} \min I(t_{k+1}), \\ \min \bar{r}(t_k) + \dfrac{t_{k+1} - t_k}{1+\delta} \end{array} \right\} \tag{P7}$$

$$\max \bar{r}(t_{k+1}) = \min \left\{ \begin{array}{c} \max I(t_{k+1}), \\ \max \bar{r}(t_k) + (1+\delta)(t_{k+1} - t_i) \end{array} \right\} \tag{P8}$$

Exploiting the recursion in (P7) and (P8) we conclude:

$$\forall 1 \leq k \leq |T_o| : \exists k' \leq k : \min \bar{r}(t_k) = \min I(t_{k'}) + \frac{t_k - t_{k'}}{1+\delta} \tag{P9}$$

$$\forall 1 \leq k \leq |T_o| : \exists k'' \leq k : \max \bar{r}(t_k) = \max I(t_{k''}) + (1+\delta)(t_k - t_{k''}) \tag{P10}$$

Notably, (P9) and (P10) also include the case $k = 1$. For $k = 1$, one simply chooses $k' = 1$ and $k'' = 1$. We now prove (G6) analogously to $I(\downarrow t) \neq \emptyset$, by establishing for each $1 \leq k \leq |T_o|$:

$$\min \bar{r}(t_k) \leq \max \bar{r}(t_k) \tag{G7}$$

Let $1 \leq k \leq |T_o|$. Further, let $k' \leq k$ and $k'' \leq k$ be the witnesses which exist according to (P9) and (P10), respectively. Now, by (P4), there exist $\langle i, \langle \uparrow t, a \rangle \rangle \in E(t_{k'})$ and $\langle i', \langle \uparrow t', a' \rangle \rangle \in E(t_{k''})$ such that:

$$\min I(t_{k'}) = \uparrow t + \mathsf{l}_{\min}(a) \qquad \max I(t_{k''}) = \uparrow t' + \mathsf{l}_{\max}(a') \tag{P11}$$

Based on these insights, we establish the following equivalences:

$$\overbrace{\min \bar{r}(t_k) \leq \max \bar{r}(t_k)}^{\text{(G7)}}$$

per (P9) and (P10)

$$\iff \min I(t_{k'}) + \frac{t_k - t_{k'}}{1+\delta} \leq \max I(t_{k''}) + (1+\delta)(t_k - t_{k''})$$

per (P11)

$$\iff \uparrow t + \mathsf{l}_{\min}(a) + \frac{t_k - t_{k'}}{1+\delta} \leq \uparrow t' + \mathsf{l}_{\max}(a') + (1+\delta)(t_k - t_{k''})$$

per elementary arithmetic

$$\iff \underbrace{{}^{\uparrow}t - {}^{\uparrow}t' + \mathsf{l}_{\mathsf{min}}(a) + \frac{t_k - t_{k'}}{1+\delta} \leq \mathsf{l}_{\mathsf{max}}(a') + (1+\delta)(t_k - t_{k''})}_{\text{(G8)}}$$

Therefore, the intermediate proof obligation (G7) becomes:

$$\underbrace{{}^{\uparrow}t - {}^{\uparrow}t' + \mathsf{l}_{\mathsf{min}}(\alpha) + \frac{t_k - t_{k'}}{1+\delta}}_{\text{(G8a)}} \leq \underbrace{\mathsf{l}_{\mathsf{max}}(\beta) + (1+\delta)(t_k - t_{k''})}_{\text{(G8b)}} \tag{G8}$$

Recall that ${}^{\uparrow}t - {}^{\uparrow}t'$ is bounded by MaxD because ϱ and ω are bound-consistent:

$${}^{\uparrow}t - {}^{\uparrow}t' \leq \mathsf{MaxD}\big(\mathcal{R}\big(\langle i, \langle {}^{\uparrow}t, a\rangle\rangle\big), \mathcal{R}\big(\langle i', \langle {}^{\uparrow}t', a'\rangle\rangle\big)\big) \tag{P12}$$

To prove (G8), we now need to distinguish two cases $k' < k''$ and $k' \geq k''$. These cases correspond to the cases of the formula (5.7) for MaxD because $k' < k''$ implies $t_{k'} < t_{k''}$ and $k' \geq k''$ implies $t_{k'} \geq t_{k''}$.

Case $k' < k''$. We have $t_{k'} < t_{k''}$ and establish (G8) as follows:

$$\begin{aligned}
&\overbrace{{}^{\uparrow}t - {}^{\uparrow}t' + \mathsf{l}_{\mathsf{min}}(a) + \frac{t_k - t_{k'}}{1+\delta}}^{\text{(G8a)}} \\
&\quad \text{per (P12)} \\
&\leq \mathsf{MaxD}\big(\mathcal{R}\big(\langle i, \langle {}^{\uparrow}t, a\rangle\rangle\big), \mathcal{R}\big(\langle i', \langle {}^{\uparrow}t', a'\rangle\rangle\big)\big) + \mathsf{l}_{\mathsf{min}}(a) + \frac{t_k - t_{k'}}{1+\delta} \\
&\quad \text{per Lemma 5.2.2 and } t_{k'} < t_{k''} \\
&= \frac{t_{k'} - t_{k''}}{1+\delta} - \mathsf{l}_{\mathsf{min}}(a) + \mathsf{l}_{\mathsf{max}}(a') + \mathsf{l}_{\mathsf{min}}(a) + \frac{t_k - t_{k'}}{1+\delta} \\
&= \frac{t_{k'} - t_{k''}}{1+\delta} + \mathsf{l}_{\mathsf{max}}(a') + \frac{t_k - t_{k'}}{1+\delta} \\
&= \frac{t_{k'} - t_{k''} + t_k - t_{k'}}{1+\delta} + \mathsf{l}_{\mathsf{max}}(a') \\
&= \frac{t_k - t_{k''}}{1+\delta} + \mathsf{l}_{\mathsf{max}}(a') \\
&\leq \underbrace{\mathsf{l}_{\mathsf{max}}(a') + (1+\delta)(t_k - t_{k''})}_{\text{(G8b)}}
\end{aligned}$$

Case $k' \geq k''$. We have $t_{k'} \geq t_{k''}$ and establish (G8) as follows:

$$\overbrace{{}^{\uparrow}t - {}^{\uparrow}t' + \mathsf{l}_{\mathsf{min}}(\alpha) + \frac{t_k - t_{k'}}{1+\delta}}^{\text{(G8a)}}$$

$$
\begin{aligned}
&\text{per (P12)}\\
&\leq \mathrm{MaxD}\big(\mathcal{R}\big(\langle i, \langle {}^{\uparrow}t, a\rangle\rangle\big), \mathcal{R}\big(\langle i', \langle {}^{\uparrow}t', a'\rangle\rangle\big)\big) + \mathsf{I}_{\mathsf{min}}(\alpha) + \frac{t_k - t_{k'}}{1+\delta}\\
&\text{per Lemma 5.2.2 and } t_{k'} \geq t_{k''}\\
&= (1+\delta)(t_{k'} - t_{k''}) - \mathsf{I}_{\mathsf{min}}(\alpha) + \mathsf{I}_{\mathsf{max}}(\beta) + \mathsf{I}_{\mathsf{min}}(\alpha) + \frac{t_k - t_{k'}}{1+\delta}\\
&= (1+\delta)(t_{k'} - t_{k''}) + \mathsf{I}_{\mathsf{max}}(\beta) + \frac{t_k - t_{k'}}{1+\delta}\\
&\leq (1+\delta)(t_{k'} - t_{k''}) + \mathsf{I}_{\mathsf{max}}(\beta) + (t_k - t_{k'})(1+\delta)\\
&= (1+\delta)(t_{k'} - t_{k''} + t_k - t_{k'}) + \mathsf{I}_{\mathsf{max}}(\beta)\\
&= (1+\delta)(t_k - t_{k''}) + \mathsf{I}_{\mathsf{max}}(\beta)\\
&= \underbrace{\mathsf{I}_{\mathsf{max}}(\beta) + (1+\delta)(t_k - t_{k''})}_{\text{(G8b)}}
\end{aligned}
$$

Therefore, we conclude $\bar{r}(t_k) \neq \varnothing$ for all t_k.

(3.) Constructing a family of time-remapping functions.

Hitherto, we have defined a non-empty interval $\bar{r}(t_k)$ for each observation time t_k found in ω taking into account (G3) and, to a degree, also the slope restriction (5.2). It remains to define a concrete time-remapping function r. We instead define a family of concrete time-remapping functions. Each of those time-remapping functions will establish consistency together with the bijection $\mathcal{R}$.

The family is defined such that for each member r:

$$r(t_{|T_o|}) \in \bar{r}\big(t_{|T_o|}\big) \tag{P13}$$

$$r(t_k) \in \bar{r}(t_k) \cap \left(r(t_{k+1}) \boxminus \left[\frac{t_{k+1} - t_k}{1+\delta}, (1+\delta)(t_{k+1} - t_k)\right]\right) \text{ for } k < |T_o| \tag{P14}$$

$$r(t) = r(t_k) + \frac{t_{k+1} - t_k}{r(t_{k+1}) - r(t_k)}(t - t_k) \text{ for } t_k < t < t_{k+1} \tag{P15}$$

$$r(t) = \begin{cases} r(t_{|T_o|}) + t - t_{|T_o|} & \text{if } t > t_{|T_o|} \\ r(t_1) - t + t_1 & \text{if } t < t_1 \end{cases} \tag{P16}$$

Note that the conditions for (P13), (P14), (P15), and (P16) are mutually exclusive, i.e., for each $t \in \mathbb{R}$ only one of them applies. For each $t_k \in T_o$, (P13) applies if $k = |T_o|$, otherwise, (P14) applies. Between the observation times, (P15) applies. Before the first (respectively after the last) observation time, (P16) applies. Concrete values can

be obtained by first fixing $r(t_{|T_o|})$ for the last observation time $t_{|T_o|}$ according to (P13). Based on this decision, values for the remaining observation times can be obtained by iterating over them in decreasing order, applying (P14), and then choosing a value from the interval. Although, for this to work, we have to prove yet again that the interval in (P14) is non-empty. Before we do that as part of the last step of this proof, we show that every function indeed satisfies (G3) and Definition 5.1.1.

By construction, $\forall {}^{\downarrow}t \in T_o : r({}^{\downarrow}t) \in I({}^{\downarrow}t)$ since $\forall {}^{\downarrow}t \in T_o : r({}^{\downarrow}t) \in \bar{r}({}^{\downarrow}t)$ and $\bar{r}({}^{\downarrow}t) \subseteq I({}^{\downarrow}t)$ which is sufficient for (G3), as we have shown before. The function r is also a strictly monotone continuous bijection because the slope in all its segments is strictly positive (cf. (P15) and (P16)) and it approaches the values $r({}^{\downarrow}t)$ for all observation times ${}^{\downarrow}t$ in the limit from both sides (cf. (P15) and (P16)) by construction. The slope before the first and after the last observation time is 1 respectively (cf. (P16)) which trivially fulfills the slope restriction (5.2). For the slope between observation times, we show that the slope restriction applies as follows:

$$
\begin{aligned}
& \overbrace{r(t_k) \in \bar{r}(t_k) \cap \left(r(t_{k+1}) \boxminus \left[\frac{t_{k+1} - t_k}{1+\delta}, (1+\delta)(t_{k+1} - t_k) \right] \right)}^{\text{(P15)}} \\
\implies & r(t_k) \in r(t_{k+1}) \boxminus \left[\frac{t_{k+1} - t_k}{1+\delta}, (1+\delta)(t_{k+1} - t_k) \right] \\
\implies & r(t_k) \in \left[r(t_{k+1}) - (1+\delta)(t_{k+1} - t_k), r(t_{k+1}) - \frac{t_{k+1} - t_k}{1+\delta} \right] \\
\implies & (r(t_{k+1}) - (1+\delta)(t_{k+1} - t_k)) \leq r(t_k) \leq \left(r(t_{k+1}) - \frac{t_{k+1} - t_k}{1+\delta} \right) \\
\implies & -(1+\delta)(t_{k+1} - t_k) \leq (r(t_k) - r(t_{k+1})) \leq -\frac{t_{k+1} - t_k}{1+\delta} \\
\implies & (1+\delta)(t_{k+1} - t_k) \geq (r(t_{k+1}) - r(t_k)) \geq \frac{t_{k+1} - t_k}{1+\delta} \\
\implies & \frac{t_{k+1} - t_k}{1+\delta} \leq (r(t_{k+1}) - r(t_k)) \leq (1+\delta)(t_{k+1} - t_k) \\
& \text{per } t_{k+1} - t_k > 0 \\
\implies & \frac{1}{1+\delta} \leq \frac{r(t_{k+1}) - r(t_k)}{t_{k+1} - t_k} \leq (1+\delta) \\
\implies & \underbrace{\frac{r(t_{k+1}) - r(t_k)}{t_{k+1} - t_k} \in \left[\frac{1}{1+\delta}, 1+\delta \right]}_{\text{slope restriction (5.2)}}
\end{aligned}
$$

Hence, every r as defined above satisfies (G3) as well as Definition 5.1.1, i.e., is a time-remapping function witnessing consistency of ϱ with ω. It remains to show,

that indeed there are r satisfying the conditions given above. That is, the interval in (P14) is non-empty such that a value can be chosen for each $1 \leq k < |T_o|$:

$$\left(\bar{r}(t_k) \cap \left(r(t_{k+1}) \boxminus \left[\frac{t_{k+1}-t_k}{1+\delta}, (1+\delta)(t_{k+1}-t_k)\right]\right)\right) \neq \emptyset \tag{G9}$$

Let $1 \leq k < |T_o|$. As values are chosen in descending order starting from a non-empty interval (cf. (P13)) we can assume that a value for $r(t_{k+1})$ has already been chosen and that $r(t_{k+1}) \in \bar{r}(t_{k+1})$. Therefore, the following transformations apply:

$$\begin{aligned}
& r(t_{k+1}) \in \bar{r}(t_{k+1}) \\
& \text{per (P6)} \\
\iff\ & r(t_{k+1}) \in I(t_{k+1}) \cap \left(\bar{r}(t_k) \boxplus \left[\frac{t_{k+1}-t_k}{1+\epsilon}, (1+\delta)(t_{k+1}-t_k)\right]\right) \\
\implies\ & r(t_{k+1}) \in \left(\bar{r}(t_k) \boxplus \left[\frac{t_{k+1}-t_k}{1+\delta}, (1+\delta)(t_{k+1}-t_k)\right]\right) \\
\implies\ & \exists x \in \bar{r}(t_k), y \in \left[\frac{t_{k+1}-t_k}{1+\delta}, (1+\delta)(t_{k+1}-t_k)\right] : r(t_{k+1}) = x + y \\
\implies\ & \exists x \in \bar{r}(t_k), y \in \left[\frac{t_{k+1}-t_k}{1+\delta}, (1+\delta)(t_{k+1}-t_k)\right] : r(t_{k+1}) - y = x \\
\implies\ & \exists x \in \bar{r}(t_k) : x \in \left(r(t_{k+1}) \boxminus \left[\frac{t_{k+1}-t_k}{1+\delta}, (1+\delta)(t_{k+1}-t_k)\right]\right) \\
\implies\ & \underbrace{\left(\bar{r}(t_k) \cap \left(r(t_{k+1}) \boxminus \left[\frac{t_{k+1}-t_k}{1+\delta}, (1+\delta)(t_{k+1}-t_k)\right]\right)\right) \neq \emptyset}_{\text{(G9)}}
\end{aligned}$$

Therefore, we obtain a non-empty family of time-remapping functions r witnessing consistency. This concludes the proof of Theorem 5.3.4. □

Bibliography

[Aba+16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. "TensorFlow: A System for Large-Scale Machine Learning". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. Savannah, GA, USA: USENIX Association, 2016, pp. 265–283. ISBN: 9781931971331.

[AD91] Rajeev Alur and David L. Dill. "The Theory of Timed Automata". In: *Real-Time: Theory in Practice, REX Workshop, Mook, The Netherlands, June 3-7, 1991, Proceedings*. Ed. by J. W. de Bakker, Cornelis Huizing, Willem P. de Roever, and Grzegorz Rozenberg. Vol. 600. Lecture Notes in Computer Science. Springer, 1991, pp. 45–73. DOI: 10.1007/BFB0031987.

[AFR16] Rajeev Alur, Dana Fisman, and Mukund Raghothaman. "Regular Programming for Quantitative Properties of Data Streams". In: *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Ed. by Peter Thiemann. Vol. 9632. Lecture Notes in Computer Science. Springer, 2016, pp. 15–40. DOI: 10.1007/978-3-662-49498-1_2.

[AFW18] Artur Andrzejak, Gerhard Friedrich, and Franz Wotawa. "Software Configuration Diagnosis - A Survey of Existing Methods and Open Challenges". In: *Proceedings of the 20th Configuration Workshop, Graz, Austria, September 27-28, 2018*. Ed. by Alexander Felfernig, Juha Tiihonen, Lothar Hotz, and Martin Stettinger. Vol. 2220. CEUR Workshop Proceedings. CEUR-WS.org, 2018, pp. 85–92.

[AH90] Rajeev Alur and Thomas A. Henzinger. "Real-time Logics: Complexity and Expressiveness". In: *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990*. IEEE Computer Society, 1990, pp. 390–401. DOI: 10.1109/LICS.1990.113764.

[AK14] Shaull Almagor and Orna Kupferman. "Latticed-LTL Synthesis in the Presence of Noisy Inputs". In: *Foundations of Software Science and Computation Structures*. Ed. by Anca Muscholl. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 226–241. ISBN: 978-3-642-54830-7.

[ALH06] E. Athanasopoulou, Lingxi Li, and C.N. Hadjicostis. "Probabilistic failure diagnosis in finite state machines under unreliable observations". In: *2006 8th International Workshop on Discrete Event Systems*. 2006, pp. 301–306. DOI: 10.1109/WODES.2006.1678446.

[And+24] Roman Andriushchenko, Alexander Bork, Carlos E. Budde, Milan Ceska, Kush Grover, Ernst Moritz Hahn, Arnd Hartmanns, Bryant Israelsen, Nils Jansen, Joshua Jeppson, Sebastian Junges, Maximilian A. Köhl, Bettina Könighofer, Jan Kretínský, Tobias Meggendorfer, David Parker, Stefan Pranger, Tim Quatmann, Enno Ruijters, Landon Taylor, Matthias Volk, Maximilian Weininger, and Zhen Zhang. "Tools at the Frontiers of Quantitative Verification: QComp 2023 Competition Report". In: *TOOLympics 2023*. 2024.

[Ape+13] Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer, 2013. ISBN: 978-3-642-37520-0. DOI: 10.1007/978-3-642-37521-7.

[AS15] Ayse Nur Acar and Klaus Werner Schmidt. "Discrete event supervisor design and application for manufacturing systems with arbitrary faults and repairs". In: *IEEE International Conference on Automation Science and Engineering, CASE 2015, Gothenburg, Sweden, August 24-28, 2015*. IEEE, 2015, pp. 825–830. DOI: 10.1109/CoASE.2015.7294183.

[Åst65] K.J Åström. "Optimal control of Markov processes with incomplete state information". In: *Journal of Mathematical Analysis and Applications* 10.1 (1965), pp. 174–205. ISSN: 0022-247X. DOI: https://doi.org/10.1016/0022-247X(65)90154-X.

[Aud+22] Giorgio Audrito, Ferruccio Damiani, Volker Stolz, Gianluca Torta, and Mirko Viroli. "Distributed runtime verification by past-CTL and the field calculus". In: *J. Syst. Softw.* 187 (2022), p. 111251. DOI: 10.1016/J.JSS.2022.111251.

[Avi+04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. "Basic Concepts and Taxonomy of Dependable and Secure Computing". In: *IEEE Trans. Dependable Secur. Comput.* 1.1 (2004), pp. 11–33. DOI: 10.1109/TDSC.2004.2.

[Bai+18] Christel Baier, Nathalie Bertrand, Clemens Dubslaff, Daniel Gburek, and Ocan Sankur. "Stochastic Shortest Paths and Weight-Bounded Properties in Markov Decision Processes". In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*. Ed. by Anuj Dawar and Erich Grädel. ACM, 2018, pp. 86–94. DOI: 10.1145/3209108.3209184.

[Bai+20] Christel Baier, Clemens Dubslaff, Holger Hermanns, Michaela Klauck, Sascha Klüppelholz, and Maximilian A. Köhl. "Components in Probabilistic Systems: Suitable by Construction". In: *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part I*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 12476. Lecture Notes in Computer Science. Springer, 2020, pp. 240–261. DOI: 10.1007/978-3-030-61362-4_13.

[Bai+21] Christel Baier, Clemens Dubslaff, Florian Funke, Simon Jantsch, Rupak Majumdar, Jakob Piribauer, and Robin Ziemek. "From Verification to Causality-Based Explications (Invited Talk)". In: *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*. Ed. by Nikhil Bansal, Emanuela Merelli, and James Worrell. Vol. 198. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 1:1–1:20. DOI: 10.4230/LIPICS.ICALP.2021.1.

[Bar+04] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. "Rule-Based Runtime Verification". In: *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, Italy, January 11-13, 2004, Proceedings*. Ed. by Bernhard Steffen and Giorgio Levi. Vol. 2937. Lecture Notes in Computer Science. Springer, 2004, pp. 44–57. DOI: 10.1007/978-3-540-24622-0_5.

[Bau+20] Jan Baumeister, Bernd Finkbeiner, Sebastian Schirmer, Maximilian Schwenger, and Christoph Torens. "RTLola Cleared for Take-Off: Monitoring Autonomous Aircraft". In: *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*. Ed. by Shuvendu K. Lahiri and Chao Wang. Vol. 12225. Lecture Notes in Computer Science. Springer, 2020, pp. 28–39. DOI: 10.1007/978-3-030-53291-8_3.

[BB87] Tommaso Bolognesi and Ed Brinksma. "Introduction to the ISO Specification Language LOTOS". In: *Comput. Networks* 14 (1987), pp. 25–59. DOI: 10.1016/0169-7552(87)90085-7.

[BBL01] Guillem Bernat, Alan Burns, and Albert Llamosí. "Weakly Hard Real-Time Systems". In: *IEEE Trans. Computers* 50.4 (2001), pp. 308–321. DOI: 10.1109/12.919277.

[BBR02] I. Broster, A. Burns, and G. Rodriguez-Navas. "Probabilistic analysis of CAN with faults". In: *23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002.* 2002, pp. 269–278. DOI: 10.1109/REAL.2002.1181581.

[BBS95] Andrew G. Barto, Steven J. Bradtke, and Satinder P. Singh. "Learning to act using real-time dynamic programming". In: *Artificial Intelligence* 72.1 (1995), pp. 81–138. ISSN: 0004-3702. DOI: 10.1016/0004-3702(94)00011-O.

[BBV15] Tessa Belder, Maurice H. ter Beek, and Erik P. de Vink. "Coherent branching feature bisimulation". In: *Proceedings 6th Workshop on Formal Methods and Analysis in SPL Engineering, FMSPLE@ETAPS 2015, London, UK, 11 April 2015*. Ed. by Joanne M. Atlee and Stefania Gnesi. Vol. 182. EPTCS. 2015, pp. 14–30. DOI: 10.4204/EPTCS.182.2.

[BCD05] Patricia Bouyer, Fabrice Chevalier, and Deepak D'Souza. "Fault Diagnosis Using Timed Automata". In: *Foundations of Software Science and Computational Structures, 8th International Conference, FOSSACS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*. Ed. by Vladimiro Sassone. Vol. 3441. Lecture Notes in Computer Science. Springer, 2005, pp. 219–233. DOI: 10.1007/978-3-540-31982-5_14.

[Bee+16] Maurice H. ter Beek, Alessandro Fantechi, Stefania Gnesi, and Franco Mazzanti. "Modelling and analysing variability in product families: Model checking of modal transition systems with variability constraints". In: *J. Log. Algebraic Methods Program.* 85.2 (2016), pp. 287–315. DOI: 10.1016/J.JLAMP.2015.11.006.

[Bee+19] Maurice H. ter Beek, Ferruccio Damiani, Michael Lienhardt, Franco Mazzanti, and Luca Paolini. "Static analysis of featured transition systems". In: *Proceedings of the 23rd International Systems and Software Product Line Conference, SPLC 2019, Volume A, Paris, France, September 9-13, 2019*. Ed. by Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnava, Thomas Thüm, and Tewfik Ziadi. ACM, 2019, 9:1–9:13. DOI: 10.1145/3336294.3336295.

[Bee+22] Maurice H. ter Beek, Ferruccio Damiani, Michael Lienhardt, Franco Mazzanti, and Luca Paolini. "Efficient static analysis and verification of featured transition systems". In: *Empir. Softw. Eng.* 27.1 (2022), p. 10. DOI: 10.1007/S10664-020-09930-8.

[BG03] Blai Bonet and Hector Geffner. "Labeled RTDP: Improving the Convergence of Real-Time Dynamic Programming". In: *ICAPS*. 2003, pp. 12–21.

[BG04] Glenn Bruns and Patrice Godefroid. "Model Checking with Multi-valued Logics". In: *Automata, Languages and Programming*. Ed. by Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 281–293. ISBN: 978-3-540-27836-8.

[BG99] Glenn Bruns and Patrice Godefroid. "Model Checking Partial State Spaces with 3-Valued Temporal Logics". In: *Computer Aided Verification*. Ed. by Nicolas Halbwachs and Doron Peled. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 274–287. ISBN: 978-3-540-48683-1.

[BHH12] Jonathan Bogdoll, Arnd Hartmanns, and Holger Hermanns. "Simulation and Statistical Model Checking for Modestly Nondeterministic Models". In: *Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance - 16th International GI/ITG Conference, MMB & DFT 2012, Kaiserslautern, Germany, March 19-21, 2012. Proceedings*. Ed. by Jens B. Schmitt. Vol. 7201. Lecture Notes in Computer Science. Springer, 2012, pp. 249–252. DOI: 10.1007/978-3-642-28540-0_20.

[Bie+21] Sebastian Biewer, Bernd Finkbeiner, Holger Hermanns, Maximilian A. Köhl, Yannik Schnitzer, and Maximilian Schwenger. "RTLola on Board: Testing Real Driving Emissions on your Phone". In: *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II*. Ed. by Jan Friso Groote and Kim Guldstrand Larsen. Vol. 12652. Lecture Notes in Computer Science. Springer, 2021, pp. 365–372. DOI: 10.1007/978-3-030-72013-1_20.

[Bie+23] Sebastian Biewer, Bernd Finkbeiner, Holger Hermanns, Maximilian A. Köhl, Yannik Schnitzer, and Maximilian Schwenger. "On the road with RTLola". In: *International Journal on Software Tools for Technology Transfer (STTT)* 25.2 (2023), pp. 205–218. DOI: 10.1007/s10009-022-00689-5.

[Bir40] Garrett Birkhoff. "Lattice Theory". In: *Journal of Symbolic Logic* 5.4 (1940), pp. 155–157. DOI: 10.2307/2268183.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008. ISBN: 978-0-262-02649-9.

[BKS17] Kevin Baum, Maximilian A. Köhl, and Eva Schmidt. "Two Challenges for CI Trustworthiness and How to Address Them". In: *Proceedings of the 1st Workshop on Explainable Computational Intelligence (XCI 2017)*. Dundee, United Kingdom: Association for Computational Linguistics, Sept. 2017. DOI: 10.18653/v1/W17-3701.

[Blo+15] Roderick Bloem, Bettina Könighofer, Robert Könighofer, and Chao Wang. "Shield Synthesis: - Runtime Enforcement for Reactive Systems". In: *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*. Ed. by Christel Baier and Cesare Tinelli. Vol. 9035. Lecture Notes in Computer Science. Springer, 2015, pp. 533–548. DOI: 10.1007/978-3-662-46681-0_51.

[BLR05] Patricia Bouyer, François Laroussinie, and Pierre-Alain Reynier. "Diagonal Constraints in Timed Automata: Forward Analysis of Timed Systems". In: *Formal Modeling and Analysis of Timed Systems, Third International Conference, FORMATS 2005, Uppsala, Sweden, September 26-28, 2005, Proceedings*. Ed. by Paul Pettersson and Wang Yi. Vol. 3829. Lecture Notes in Computer Science. Springer, 2005, pp. 112–126. DOI: 10.1007/11603009_10.

[BLS06a] Andreas Bauer, Martin Leucker, and Christian Schallhart. "Model-based runtime analysis of distributed reactive systems". In: *17th Australian Software Engineering Conference (ASWEC 2006), 18-21 April 2006, Sydney, Australia*. IEEE Computer Society, 2006, pp. 243–252. DOI: 10.1109/ASWEC.2006.36.

[BLS06b] Andreas Bauer, Martin Leucker, and Christian Schallhart. "Monitoring of Real-Time Properties". In: *FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science, 26th International Conference, Kolkata, India, December 13-15, 2006, Proceedings*. Ed. by S. Arun-Kumar and Naveen Garg. Vol. 4337. Lecture Notes in Computer Science. Springer, 2006, pp. 260–272. DOI: 10.1007/11944836_25.

[BLS07] Andreas Bauer, Martin Leucker, and Christian Schallhart. "The Good, the Bad, and the Ugly, But How Ugly Is Ugly?" In: *Runtime Verification, 7th International Workshop, RV 2007, Vancouver, Canada, March 13, 2007, Revised Selected Papers*. Ed. by Oleg Sokolsky and Serdar Tasiran. Vol. 4839. Lecture Notes in Computer Science. Springer, 2007, pp. 126–138. DOI: 10.1007/978-3-540-77395-5_11.

[BLS11] Andreas Bauer, Martin Leucker, and Christian Schallhart. "Runtime Verification for LTL and TLTL". In: *ACM Trans. Softw. Eng. Methodol.* 20.4 (2011), 14:1–14:64. DOI: 10.1145/2000799.2000800.

[Bod+04] Maik Boden, Manfred Koegst, José Luis Tiburcio Badía, and Steffen Rülke. "Cost-Efficient Implementation of Adaptive Finite State Machines". In: *2004 Euromicro Symposium on Digital Systems Design (DSD 2004), Architectures, Methods and Tools, 31 August - 3 September 2004, Rennes, France*. IEEE Computer Society, 2004, pp. 144–151. DOI: 10.1109/DSD.2004.1333270.

[Bou07] Patricia Bouyer. "Model-checking Timed Temporal Logics". In: *Proceedings of the 5th Workshop on Methods for Modalities, M4M 2007, Cachan, France, November 29-30, 2007*. Ed. by Carlos Areces and Stéphane Demri. Vol. 231. Electronic Notes in Theoretical Computer Science. Elsevier, 2007, pp. 323–341. DOI: 10.1016/J.ENTCS.2009.02.044.

[BRV01] Patrick Blackburn, Maarten de Rijke, and Yde Venema. *Modal Logic*. Vol. 53. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2001. ISBN: 978-1-10705088-4. DOI: 10.1017/CBO9781107050884.

[Bry86] Randal E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". In: *IEEE Trans. Computers* 35.8 (1986), pp. 677–691. DOI: 10.1109/TC.1986.1676819.

[BS14] Laura Bozzelli and César Sánchez. "Foundations of Boolean Stream Runtime Verification". In: *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*. Ed. by Borzoo Bonakdarpour and Scott A. Smolka. Vol. 8734. Lecture Notes in Computer Science. Springer, 2014, pp. 64–79. DOI: 10.1007/978-3-319-11164-3_6.

[BS16] Laura Bozzelli and César Sánchez. "Foundations of Boolean stream runtime verification". In: *Theor. Comput. Sci.* 631 (2016), pp. 118–138. DOI: 10.1016/J.TCS.2016.04.019.

[BS69] Jacobus W de Bakker and Dana Scott. "A theory of programs". In: *IBM seminar, Vienna*. 1969.

[Buc+10] Christian Buckl, Irina Gaponova, Michael Geisinger, Alois Knoll, and Edward A Lee. "Model-based specification of timing requirements". In: *Proceedings of the tenth ACM international conference on Embedded software*. 2010, pp. 239–248.

[Bud+17] Carlos E. Budde, Christian Dehnert, Ernst Moritz Hahn, Arnd Hartmanns, Sebastian Junges, and Andrea Turrini. "JANI: Quantitative Model and Tool Interaction". In: *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 II*. Ed. by Axel Legay and Tiziana Margaria. Vol. 10206. Lecture Notes in Computer Science. 2017, pp. 151–168. ISBN: 978-3-662-54579-9. DOI: 10.1007/978-3-662-54580-5_9.

[Bud+18] Carlos E. Budde, Pedro R. D'Argenio, Arnd Hartmanns, and Sean Sedwards. "A Statistical Model Checker for Nondeterminism and Rare Events". In: *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 II*. Ed. by Dirk Beyer and Marieke Huisman. Vol. 10806. Lecture Notes in Computer Science. Springer, 2018, pp. 340–358. ISBN: 978-3-319-89962-6. DOI: 10.1007/978-3-319-89963-3_20.

[Bud+20] Carlos E. Budde, Arnd Hartmanns, Michaela Klauck, Jan Kretinsky, David Parker, Tim Quatmann, Andrea Turrini, and Zhen Zhang. "On Correctness, Precision, and Performance in Quantitative Verification (QComp 2020 Competition Report)". In: *Proceedings of the 9th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation. Software Verification Tools*. 2020.

[BY03] Johan Bengtsson and Wang Yi. "Timed Automata: Semantics, Algorithms and Tools". In: *Lectures on Concurrency and Petri Nets, Advances in Petri Nets [This tutorial volume originates from the 4th Advanced Course on Petri Nets, ACPN 2003, held in Eichstätt, Germany in September 2003. In addition to lectures given at ACPN 2003, additional chapters have been commissioned]*. Ed. by Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg. Vol. 3098. Lecture Notes in Computer Science. Springer, 2003, pp. 87–124. DOI: 10.1007/978-3-540-27755-2_3.

[Car+13] Lilian K. Carvalho, Marcos Vicente Moreira, João Carlos Basilio, and Stéphane Lafortune. "Robust diagnosis of discrete-event systems against permanent loss of observations". In: *Autom.* 49.1 (2013), pp. 223–231. DOI: 10.1016/j.automatica.2012.09.017.

[Car+19] Thyago Peres Carvalho, Fabrízzio Alphonsus A. M. N. Soares, Roberto Vita, Roberto da Piedade Francisco, João Pedro Tavares Vieira Basto, and Symone G. S. Alcalá. "A systematic literature review of machine

learning methods applied to predictive maintenance". In: *Comput. Ind. Eng.* 137 (2019). DOI: 10.1016/J.CIE.2019.106024.

[CCM08] Miguel Castro, Manuel Costa, and Jean-Philippe Martin. "Better bug reporting with better privacy". In: *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008*. Ed. by Susan J. Eggers and James R. Larus. ACM, 2008, pp. 319–328. DOI: 10.1145/1346281.1346322.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming - methods, tools and applications*. Addison-Wesley, 2000. ISBN: 978-0-201-30977-5.

[CE81] Edmund M. Clarke and E. Allen Emerson. "Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic". In: *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981*. Ed. by Dexter Kozen. Vol. 131. Lecture Notes in Computer Science. Springer, 1981, pp. 52–71. DOI: 10.1007/BFB0025774.

[Cev+14] Kristína Cevorová, Galina Jirásková, Peter Mlynárcik, Matús Palmovský, and Juraj Sebej. "Operations on Automata with All States Final". In: *Proceedings 14th International Conference on Automata and Formal Languages, AFL 2014, Szeged, Hungary, May 27-29, 2014*. Ed. by Zoltán Ésik and Zoltán Fülöp. Vol. 151. EPTCS. 2014, pp. 201–215. DOI: 10.4204/EPTCS.151.14.

[Che+06] Jingdong Chen, Jacob Benesty, Yiteng Arden Huang, and Simon Doclo. "New insights into the noise reduction Wiener filter". In: *IEEE Trans. Speech Audio Process.* 14.4 (2006), pp. 1218–1234. DOI: 10.1109/TSA.2005.860851.

[Che80] Brian F. Chellas. *Modal Logic - An Introduction*. Cambridge University Press, 1980. ISBN: 978-0-51162119-2. DOI: 10.1017/CBO9780511621192.

[Chr+18] Philipp Chrszon, Clemens Dubslaff, Sascha Klüppelholz, and Christel Baier. "ProFeat: feature-oriented engineering for family-based probabilistic model checking". In: *Formal Aspects Comput.* 30.1 (2018), pp. 45–75. DOI: 10.1007/S00165-017-0432-4.

[Cla+10] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. "Model checking lots of systems: efficient verification of temporal properties in software product lines". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*. Ed. by Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel. ACM, 2010, pp. 335–344. DOI: 10.1145/1806799.1806850.

[Cla+13] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-François Raskin. "Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking". In: *IEEE Trans. Software Eng.* 39.8 (2013), pp. 1069–1089. DOI: 10.1109/TSE.2012.86.

[Cla10] Andreas Classen. *Modelling with FTS: a Collection of Illustrative Examples*. Tech. rep. P-CS-TR SPLMC-00000001. Namur, Belgium: PReCISE Research Center, University of Namur, 2010.

[CN02] Paul Clements and Linda M. Northrop. *Software product lines - practices and patterns*. SEI series in software engineering. Addison-Wesley, 2002. ISBN: 978-0-201-70332-0.

[Con+18] Lukas Convent, Sebastian Hungerecker, Martin Leucker, Torben Scheffel, Malte Schmitz, and Daniel Thoma. "TeSSLa: Temporal Stream-Based Specification Language". In: *Formal Methods: Foundations and Applications - 21st Brazilian Symposium, SBMF 2018, Salvador, Brazil, November 26-30, 2018, Proceedings*. Ed. by Tiago Massoni and Mohammad Reza Mousavi. Vol. 11254. Lecture Notes in Computer Science. Springer, 2018, pp. 144–162. DOI: 10.1007/978-3-030-03044-5_10.

[Cor+12] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. "Behavioural modelling and verification of real-time software product lines". In: *16th International Software Product Line Conference, SPLC '12, Salvador, Brazil - September 2-7, 2012, Volume 1*. Ed. by Eduardo Santana de Almeida, Christa Schwanninger, and David Benavides. ACM, 2012, pp. 66–75. DOI: 10.1145/2362536.2362549.

[CPS08] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. "Dynamic Event-Based Runtime Monitoring of Real-Time and Contextual Properties". In: *Formal Methods for Industrial Critical Systems, 13th International Workshop, FMICS 2008, L'Aquila, Italy, September 15-16, 2008, Revised Selected Papers*. Ed. by Darren D. Cofer and Alessandro Fantechi. Vol. 5596. Lecture Notes in Computer Science. Springer, 2008, pp. 135–149. DOI: 10.1007/978-3-642-03240-0_13.

[CTT19] Alessandro Cimatti, Chun Tian, and Stefano Tonetta. "Assumption-Based Runtime Verification with Partial Observability and Resets". In: *Runtime Verification - 19th International Conference, RV 2019, Porto, Portugal, October 8-11, 2019, Proceedings*. Ed. by Bernd Finkbeiner and Leonardo Mariani. Vol. 11757. Lecture Notes in Computer Science. Springer, 2019, pp. 165–184. DOI: 10.1007/978-3-030-32079-9_10.

[CW96] Edmund M. Clarke and Jeannette M. Wing. "Formal Methods: State of the Art and Future Directions". In: *ACM Comput. Surv.* 28.4 (1996), pp. 626–643. DOI: 10.1145/242223.242257.

[DAn+05] Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, and Zohar Manna. "LOLA: Runtime Monitoring of Synchronous Systems". In: *12th International Symposium on Temporal Representation and Reasoning (TIME 2005), 23-25 June 2005, Burlington, Vermont, USA*. IEEE Computer Society, 2005, pp. 166–174. DOI: 10.1109/TIME.2005.26.

[Deu23] Präsidium der Deutschen Forschungsgemeinschaft (DFG). "Stellungnahme des Präsidiums der Deutschen Forschungsgemeinschaft (DFG) zum Einfluss generativer Modelle für die Text- und Bilderstellung auf die Wissenschaftenund das Förderhandeln der DFG". In: (Sept. 2023).

[Dev+14] Xavier Devroey, Gilles Perrouin, Maxime Cordy, Pierre-Yves Schobbens, Axel Legay, and Patrick Heymans. "Towards statistical prioritization for software product lines testing". In: *The Eighth International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '14, Sophia Antipolis, France, January 22-24, 2014*. Ed. by Philippe Collet, Andrzej Wasowski, and Thorsten Weyer. ACM, 2014, 10:1–10:7. DOI: 10.1145/2556624.2556635.

[Dev17] Xavier Devroey. "Behavioural model-based testing of software product lines". PhD thesis. University of Namur, 2017. DOI: 10.5281/zenodo.4105899.

[Di +12] Marco Di Natale, Haibo Zeng, Paolo Giusto, and Arkadeb Ghosal. *Understanding and using the controller area network communication protocol: theory and practice*. Springer Science & Business Media, 2012.

[Dil89] David L. Dill. "Timing Assumptions and Verification of Finite-State Concurrent Systems". In: *Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France, June 12-14, 1989, Proceedings*. Ed. by Joseph Sifakis. Vol. 407. Lecture Notes in Computer Science. Springer, 1989, pp. 197–212. DOI: 10.1007/3-540-52148-8_17.

[DK22] Clemens Dubslaff and Maximilian A. Köhl. "Configurable-by-Construction Runtime Monitoring". In: *Leveraging Applications of Formal Methods, Verification and Validation. Verification Principles - 11th International Symposium, ISoLA 2022, Rhodes, Greece, October 22-30, 2022, Proceedings, Part I*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 13701. Lecture Notes in Computer Science. Springer, 2022, pp. 220–241. DOI: 10.1007/978-3-031-19849-6_14.

[DLS08] Wei Dong, Martin Leucker, and Christian Schallhart. "Impartial Anticipation in Runtime-Verification". In: *Automated Technology for Verification and Analysis, 6th International Symposium, ATVA 2008, Seoul, Korea, October 20-23, 2008. Proceedings*. Ed. by Sung Deok Cha, Jin-Young Choi, Moonzoo Kim, Insup Lee, and Mahesh Viswanathan. Vol. 5311. Lecture Notes in Computer Science. Springer, 2008, pp. 386–396. DOI: 10.1007/978-3-540-88387-6_33.

[DP17] Tom van Dijk and Jaco van de Pol. "Sylvan: multi-core framework for decision diagrams". In: *Int. J. Softw. Tools Technol. Transf.* 19.6 (2017), pp. 675–696. DOI: 10.1007/S10009-016-0433-2.

[Dub+24] Clemens Dubslaff, Kallistos Weis, Christel Baier, and Sven Apel. "Feature causality". In: *J. Syst. Softw.* 209 (2024), p. 111915. DOI: 10.1016/J.JSS.2023.111915.

[Dub19] Clemens Dubslaff. "Compositional Feature-Oriented Systems". In: *Software Engineering and Formal Methods - 17th International Conference, SEFM 2019, Oslo, Norway, September 18-20, 2019, Proceedings*. Ed. by Peter Csaba Ölveczky and Gwen Salaün. Vol. 11724. Lecture Notes in Computer Science. Springer, 2019, pp. 162–180. DOI: 10.1007/978-3-030-30446-1_9.

[Dub22] Clemens Dubslaff. "Quantitative Analysis of Configurable and Reconfigurable Systems". PhD thesis. Dresden University of Technology, Germany, 2022.

[Duf+06] Marie Duflot, Marta Z. Kwiatkowska, Gethin Norman, and David Parker. "A formal analysis of bluetooth device discovery". In: *Int. J. Softw. Tools Technol. Transf.* 8.6 (2006), pp. 621–632. DOI: 10.1007/S10009-006-0014-X.

[Dur+16] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. "Spot 2.0—A Framework for LTL and ω-Automata Manipulation". In: *International Symposium on Automated Technology for Verification and Analysis*. Springer. 2016, pp. 122–129.

[Ell63] E. O. Elliott. "Estimates of error rates for codes on burst-noise channels". In: *The Bell System Technical Journal* 42.5 (1963), pp. 1977–1997. DOI: 10.1002/j.1538-7305.1963.tb00955.x.

[Eri99] Clifton A. Ericson. "Fault Tree Analysis — A History". In: *Proceedings of the 17th International Systems Safety Conference*. 1999.

[Eur16] European Parliament and Council of the European Union. *Commission Regulation (EU) 2016/427*. Mar. 2016. URL: http://data.europa.eu/eli/reg/2016/427/oj (visited on 06/18/2018).

[Eur17] European Parliament and Council of the European Union. *Commission Regulation (EU) 2017/1151*. June 2017. URL: http://data.europa.eu/eli/reg/2017/1151/oj (visited on 06/18/2018).

[Eur98] European Parliament and Council of the European Union. "Directive 98/69/EC of the European Parliament and of the Council". In: *Official Journal of the European Communities* (1998).

[FAI15] Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. "On Verifying Hennessy-Milner Logic with Recursion at Runtime". In: *Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings*. Ed. by Ezio Bartocci and Rupak Majumdar. Vol. 9333. Lecture Notes in Computer Science. Springer, 2015, pp. 71–86. DOI: 10.1007/978-3-319-23820-3_5.

[Fal+11] Yliès Falcone, Laurent Mounier, Jean-Claude Fernandez, and Jean-Luc Richier. "Runtime enforcement monitors: composition, synthesis, and enforcement abilities". In: *Formal Methods Syst. Des.* 38.3 (2011), pp. 223–262. DOI: 10.1007/S10703-011-0114-4.

[Faq+20] Rasha Faqeh, Christof Fetzer, Holger Hermanns, Jörg Hoffmann, Michaela Klauck, Maximilian A. Köhl, Marcel Steinmetz, and Christoph Weidenbach. "Towards Dynamic Dependable Systems Through Evidence-Based Continuous Certification". In: *Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part II*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 12477. Lecture Notes in Computer Science. Springer, 2020, pp. 416–439. DOI: 10.1007/978-3-030-61470-6_25.

[Fay+16] Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Hazem Torfah. "A Stream-Based Specification Language for Network Monitoring". In: *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings*. Ed. by Yliès Falcone and César Sánchez. Vol. 10012. Lecture Notes in Computer Science. Springer, 2016, pp. 152–168. DOI: 10.1007/978-3-319-46982-9_10.

[Fay+19] Peter Faymonville, Bernd Finkbeiner, Malte Schledjewski, Maximilian Schwenger, Marvin Stenger, Leander Tentrup, and Hazem Torfah. "StreamLAB: Stream-based Monitoring of Cyber-Physical Systems". In: *Proceedings of the 31st International Conference on Computer Aided Verification (CAV'19)*. Vol. 11561. LNCS. Springer, 2019, pp. 421–431.

[Fel05] Max Felser. "Real-time ethernet-industry prospective". In: *Proceedings of the IEEE* 93.6 (2005), pp. 1118–1129.

[Fer+21] Angelo Ferrando, Rafael C. Cardoso, Marie Farrell, Matt Luckcuck, Fabio Papacchini, Michael Fisher, and Viviana Mascardi. "Bridging the gap between single- and multi-model predictive runtime verification". In: *Formal Methods Syst. Des.* 59.1 (2021), pp. 44–76. DOI: 10.1007/s10703-022-00395-7.

[Fer+96] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Laurent Mounier, Radu Mateescu, and Mihaela Sighireanu. "CADP - A Protocol Validation and Verification Toolbox". In: *Computer Aided Verification, 8th International Conference, CAV '96, New Brunswick, NJ, USA, July 31 - August 3, 1996, Proceedings*. Ed. by Rajeev Alur and Thomas A. Henzinger. Vol. 1102. Lecture Notes in Computer Science. Springer, 1996, pp. 437–440. ISBN: 3-540-61474-5. DOI: 10.1007/3-540-61474-5_97.

[FFM12] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. "What can you verify and enforce at runtime?" In: *Int. J. Softw. Tools Technol. Transf.* 14.3 (2012), pp. 349–382. DOI: 10.1007/s10009-011-0196-8.

[Fin+19] Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. "Monitoring hyperproperties". In: *Formal Methods Syst. Des.* 54.3 (2019), pp. 336–363. DOI: 10.1007/S10703-019-00334-Z.

[FMY97] Masahiro Fujita, Patrick C. McGeer, and Jerry Chih-Yuan Yang. "Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation". In: *Formal Methods Syst. Des.* 10.2/3 (1997), pp. 149–169. DOI: 10.1023/A:1008647823331.

[Fra90] Paul Martin Frank. "Fault diagnosis in dynamic systems using analytical and knowledge-based redundancy: A survey and some new results". In: *Autom.* 26.3 (1990), pp. 459–474. DOI: 10.1016/0005-1098(90)90018-D.

[Gar+13] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. "CADP 2011: a toolbox for the construction and analysis of distributed processes". In: *Int. J. Softw. Tools Technol. Transf.* 15.2 (2013), pp. 89–107. DOI: 10.1007/s10009-012-0244-z.

[Gar73] Martin Gardner. "Mathematical games". In: *Scientific American* 229 (1973), pp. 118–121.

[Gil60] E. N. Gilbert. "Capacity of a burst-noise channel". In: *The Bell System Technical Journal* 39.5 (1960), pp. 1253–1265. DOI: 10.1002/j.1538-7305.1960.tb03959.x.

[GK15] Hila Gonen and Orna Kupferman. "Inherent Vacuity in Lattice Automata". In: *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*. Ed. by Lev D. Beklemishev, Andreas Blass, Nachum Dershowitz, Bernd Finkbeiner, and

Wolfram Schulte. Vol. 9300. Lecture Notes in Computer Science. Springer, 2015, pp. 174–192. DOI: 10.1007/978-3-319-23534-9_10.

[GLM18] Hubert Garavel, Frédéric Lang, and Laurent Mounier. "Compositional Verification in Action". In: *Formal Methods for Industrial Critical Systems - 23rd International Conference, FMICS 2018, Maynooth, Ireland, September 3-4, 2018, Proceedings*. Ed. by Falk Howar and Jiri Barnat. Vol. 11119. Lecture Notes in Computer Science. Springer, 2018, pp. 189–210. ISBN: 978-3-030-00243-5. DOI: 10.1007/978-3-030-00244-2_13.

[GLS08] Alexander Gruler, Martin Leucker, and Kathrin Danielle Scheidemann. "Modeling and Model Checking Software Product Lines". In: *Formal Methods for Open Object-Based Distributed Systems, 10th IFIP WG 6.1 International Conference, FMOODS 2008, Oslo, Norway, June 4-6, 2008, Proceedings*. Ed. by Gilles Barthe and Frank S. de Boer. Vol. 5051. Lecture Notes in Computer Science. Springer, 2008, pp. 113–131. DOI: 10.1007/978-3-540-68863-1_8.

[GN92] Naresh Gupta and Dana S. Nau. "On the Complexity of Blocks-World Planning". In: *Artif. Intell.* 56.2-3 (1992), pp. 223–254. DOI: 10.1016/0004-3702(92)90028-V.

[Gro+20a] Timo P. Gros, David Großand Stefan Gumhold, Jörg Hoffmann, Michaela Klauck, and Marcel Steinmetz. "TraceVis: Towards Visualization for Deep Statistical Model Checking". In: *Proceedings of the 9th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation. From Verification to Explanation*. 2020.

[Gro+20b] Timo P. Gros, Holger Hermanns, Jörg Hoffmann, Michaela Klauck, and Marcel Steinmetz. "Deep Statistical Model Checking". In: *Formal Techniques for Distributed Objects, Components, and Systems - 40th IFIP WG 6.1 International Conference, FORTE 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings*. Ed. by Alexey Gotsman and Ana Sokolova. Vol. 12136. Lecture Notes in Computer Science. Springer, 2020, pp. 96–114. ISBN: 978-3-030-50085-6. DOI: 10.1007/978-3-030-50086-3_6.

[Gro+22] Timo P. Gros, Holger Hermanns, Jörg Hoffmann, Michaela Klauck, Maximilian A. Köhl, and Verena Wolf. "MoGym: Using Formal Models for Training and Verifying Decision-making Agents". In: *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II*. Ed. by Sharon Shoham and Yakir Vizel. Vol. 13372. Lecture Notes in Computer Science. Springer, 2022, pp. 430–443. DOI: 10.1007/978-3-031-13188-2_21.

[GS18] Felipe Gorostiaga and César Sánchez. "Striver: Stream Runtime Verification for Real-Time Event-Streams". In: *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings*. Ed. by Christian Colombo and Martin Leucker. Vol. 11237. Lecture Notes in Computer Science. Springer, 2018, pp. 282–298. DOI: 10.1007/978-3-030-03769-7_16.

[Hah+13] Ernst Moritz Hahn, Arnd Hartmanns, Holger Hermanns, and Joost-Pieter Katoen. "A compositional modelling and analysis framework for stochastic hybrid systems". In: *Formal Methods Syst. Des.* 43.2 (2013), pp. 191–232. DOI: 10.1007/s10703-012-0167-z.

[Hah+14] Ernst Moritz Hahn, Yi Li, Sven Schewe, Andrea Turrini, and Lijun Zhang. "iscasMc: A Web-Based Probabilistic Model Checker". In: *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*. Ed. by Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun. Vol. 8442. Lecture Notes in Computer Science. Springer, 2014, pp. 312–317. ISBN: 978-3-319-06409-3. DOI: 10.1007/978-3-319-06410-9_22.

[Hah+19] Ernst Moritz Hahn, Arnd Hartmanns, Christian Hensel, Michaela Klauck, Joachim Klein, Jan Kretínský, David Parker, Tim Quatmann, Enno Ruijters, and Marcel Steinmetz. "The 2019 Comparison of Tools for the Analysis of Quantitative Formal Models - (QComp 2019 Competition Report)". In: *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III*. Ed. by Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen. Vol. 11429. Lecture Notes in Computer Science. Springer, 2019, pp. 69–92. ISBN: 978-3-030-17501-6. DOI: 10.1007/978-3-030-17502-3_5.

[Har+19] Arnd Hartmanns, Michaela Klauck, David Parker, Tim Quatmann, and Enno Ruijters. "The Quantitative Verification Benchmark Set". In: *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I*. Ed. by Tomás Vojnar and Lijun Zhang. Vol. 11427. Lecture Notes in Computer Science. Springer, 2019, pp. 344–350. ISBN: 978-3-030-17461-3. DOI: 10.1007/978-3-030-17462-0_20.

[Hav+10] Klaus Havelund, Martin Leucker, Martin Sachenbacher, Oleg Sokolsky, and Brian C. Williams, eds. *Runtime Verification, Diagnosis, Planning and Control for Autonomous Systems, 07.11. - 12.11.2010*. Vol. 10451. Dagstuhl Seminar Proceedings. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany, 2010.

[Hen+22] Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. "The probabilistic model checker Storm". In: *Int. J. Softw. Tools Technol. Transf.* 24.4 (2022), pp. 589–610. DOI: 10.1007/S10009-021-00633-Z.

[Hen98] Thomas A. Henzinger. "It's About Time: Real-Time Logics Reviewed". In: *CONCUR '98: Concurrency Theory, 9th International Conference, Nice, France, September 8-11, 1998, Proceedings*. Ed. by Davide Sangiorgi and Robert de Simone. Vol. 1466. Lecture Notes in Computer Science. Springer, 1998, pp. 439–454. DOI: 10.1007/BFB0055640.

[Her+18] Holger Hermanns, Sebastian Biewer, Pedro R. D'Argenio, and Maximilian A. Köhl. "Verification, Testing, and Runtime Monitoring of Automotive Exhaust Emissions". In: *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16-21 November 2018*. Ed. by Gilles Barthe, Geoff Sutcliffe, and Margus Veanes. Vol. 57. EPiC Series in Computing. EasyChair, 2018, pp. 1–17. DOI: 10.29007/6zxt.

[HG05] Klaus Havelund and Allen Goldberg. "Verify Your Runs". In: *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*. Ed. by Bertrand Meyer and Jim Woodcock. Vol. 4171. Lecture Notes in Computer Science. Springer, 2005, pp. 374–383. DOI: 10.1007/978-3-540-69149-5_40.

[HH14] Arnd Hartmanns and Holger Hermanns. "The Modest Toolset: An Integrated Environment for Quantitative Modelling and Verification". In: *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*. Ed. by Erika Ábrahám and Klaus Havelund. Vol. 8413. Lecture Notes in Computer Science. Springer, 2014, pp. 593–598. DOI: 10.1007/978-3-642-54862-8_51.

[HK11] Shulamit Halamish and Orna Kupferman. "Minimizing Deterministic Lattice Automata". In: *Foundations of Software Science and Computational Structures - 14th International Conference, FOSSACS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*. Ed. by Martin Hofmann. Vol. 6604. Lecture Notes in Computer Science. Springer, 2011, pp. 199–213. DOI: 10.1007/978-3-642-19805-2_14.

[HK12] Shulamit Halamish and Orna Kupferman. "Approximating Deterministic Lattice Automata". In: *Automated Technology for Verification and Analysis - 10th International Symposium, ATVA 2012, Thiruvananthapuram, India, October 3-6, 2012. Proceedings*. Ed. by Supratik Chakraborty and Madhavan Mukund. Vol. 7561. Lecture Notes in Computer Science. Springer, 2012, pp. 27–41. DOI: 10.1007/978-3-642-33386-6_4.

[HK15] Shulamit Halamish and Orna Kupferman. "Minimizing Deterministic Lattice Automata". In: *ACM Trans. Comput. Log.* 16.1 (2015), 1:1–1:21. DOI: 10.1145/2631915.

[HLP01] Klaus Havelund, Michael R. Lowry, and John Penix. "Formal Analysis of a Space-Craft Controller Using SPIN". In: *IEEE Trans. Software Eng.* 27.8 (2001), pp. 749–765. DOI: 10.1109/32.940728.

[HMU01] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 2nd Edition*. Addison-Wesley series in computer science. Addison-Wesley-Longman, 2001. ISBN: 978-0-201-44124-6.

[HMU07] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 3rd Edition*. Pearson international edition. Addison-Wesley, 2007. ISBN: 978-0-321-47617-3.

[Hoa78] C. A. R. Hoare. "Communicating Sequential Processes". In: *Commun. ACM* 21.8 (1978), pp. 666–677. DOI: 10.1145/359576.359585.

[Hof+20] Jörg Hoffmann, Holger Hermanns, Michaela Klauck, Marcel Steinmetz, Erez Karpas, and Daniele Magazzeni. "Let's Learn Their Language? A Case for Planning with Automata-Network Languages from Model Checking". In: *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 2020, pp. 13569–13575. ISBN: 978-1-57735-823-7.

[Hol97] Gerard J. Holzmann. "The Model Checker SPIN". In: *IEEE Trans. Software Eng.* 23.5 (1997), pp. 279–295. DOI: 10.1109/32.588521.

[Hop71] John Hopcroft. "AN n log n ALGORITHM FOR MINIMIZING STATES IN A FINITE AUTOMATON". In: *Theory of Machines and Computations*. Ed. by Zvi Kohavi and Azaria Paz. Academic Press, 1971, pp. 189–196. ISBN: 978-0-12-417750-5. DOI: https://doi.org/10.1016/B978-0-12-417750-5.50022-1.

[HP01] Joseph Y. Halpern and Judea Pearl. "Causes and Explanations: A Structural-Model Approach - Part II: Explanations". In: *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4-10, 2001*. Ed. by Bernhard Nebel. Morgan Kaufmann, 2001, pp. 27–34.

[Hus+24] Nils Husung, Clemens Dubslaff, Holger Hermanns, and Maximilian A. Köhl. "OxiDD - A Safe, Concurrent, Modular, and Performant Decision Diagram Framework in Rust". In: *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part III*. Ed. by Bernd Finkbeiner and Laura Kovács. Vol. 14572. Lecture Notes in Computer Science. Springer, 2024, pp. 255–275. DOI: 10.1007/978-3-031-57256-2_13.

[Ise05] Rolf Isermann. "Model-based fault-detection and diagnosis - status and applications". In: *Annu. Rev. Control.* 29.1 (2005), pp. 71–85. DOI: 10.1016/J.ARCONTROL.2004.12.002.

[Ive+00] Torsten K. Iversen, Kåre J. Kristoffersen, Kim Guldstrand Larsen, Morten Laursen, Rune G. Madsen, Steffen K. Mortensen, Paul Pettersson, and Chris B. Thomasen. "Model-checking real-time control programs: verifying Lego(R) Mindstorms^{TM} systems using UPPAAL". In: *12th Euromicro Conference on Real-Time Systems (ECRTS 2000), 19-21 June 2000, Stockholm, Sweden, Proceedings*. IEEE Computer Society, 2000, pp. 147–155. DOI: 10.1109/EMRTS.2000.854002.

[Jan+20] Nils Jansen, Bettina Könighofer, Sebastian Junges, Alex Serban, and Roderick Bloem. "Safe Reinforcement Learning Using Probabilistic Shields (Invited Paper)". In: *31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference)*. Ed. by Igor Konnov and Laura Kovács. Vol. 171. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 3:1–3:16. DOI: 10.4230/LIPICS.CONCUR.2020.3.

[JJS21] Sebastian Junges, Nils Jansen, and Sanjit A. Seshia. "Enforcing Almost-Sure Reachability in POMDPs". In: *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*. Ed. by Alexandra Silva and K. Rustan M. Leino. Vol. 12760. Lecture Notes in Computer Science. Springer, 2021, pp. 602–625. DOI: 10.1007/978-3-030-81688-9_28.

[JM11] Galina Jirásková and Tomás Masopust. “State Complexity of Projected Languages”. In: *Descriptional Complexity of Formal Systems - 13th International Workshop, DCFS 2011, Gießen/Limburg, Germany, July 25-27, 2011. Proceedings*. Ed. by Markus Holzer, Martin Kutrib, and Giovanni Pighizzini. Vol. 6808. Lecture Notes in Computer Science. Springer, 2011, pp. 198–211. DOI: 10.1007/978-3-642-22600-7_16.

[Jun+21] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. “Safe systems programming in Rust”. In: *Commun. ACM* 64.4 (2021), pp. 144–152. DOI: 10.1145/3418295.

[KAK08] Christian Kästner, Sven Apel, and Martin Kuhlemann. “Granularity in software product lines”. In: *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*. Ed. by Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn. ACM, 2008, pp. 311–320. DOI: 10.1145/1368088.1368131.

[Kan+90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. rep. Carnegie-Mellon University Software Engineering Institute, 1990.

[KDH24] Maximilian A. Köhl, Clemens Dubslaff, and Holger Hermanns. “Configuration Monitor Synthesis”. In: *Automated Technology for Verification and Analysis, ATVA 2024*. Lecture Notes in Computer Science. Accepted for publication. 2024.

[Kel76] Robert M. Keller. “Formal Verification of Parallel Programs”. In: *Commun. ACM* 19.7 (1976), pp. 371–384. DOI: 10.1145/360248.360251.

[KH23] Maximilian A. Köhl and Holger Hermanns. “Model-Based Diagnosis of Real-Time Systems: Robustness Against Varying Latency, Clock Drift, and Out-of-Order Observations”. In: *ACM Transactions on Embedded Computing Systems* 22.4 (2023), 68:1–68:48. DOI: 10.1145/3597209.

[KHB18] Maximilian A. Köhl, Holger Hermanns, and Sebastian Biewer. “Efficient Monitoring of Real Driving Emissions”. In: *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings*. Ed. by Christian Colombo and Martin Leucker. Vol. 11237. Lecture Notes in Computer Science. Springer, 2018, pp. 299–315. DOI: 10.1007/978-3-030-03769-7_17.

[Kim+10] Chang Hwan Peter Kim, Eric Bodden, Don Batory, and Sarfraz Khurshid. “Reducing Configurations to Monitor in a Software Product Line”. In: *Proceedings of the 10th International Conference on Runtime Verification (RV’10)*. Berlin, Heidelberg: Springer, 2010, pp. 285–299.

[KKH21] Maximilian A. Köhl, Michaela Klauck, and Holger Hermanns. "Momba: JANI Meets Python". In: *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II*. Ed. by Jan Friso Groote and Kim Guldstrand Larsen. Vol. 12652. Lecture Notes in Computer Science. Springer, 2021, pp. 389–398. DOI: 10.1007/978-3-030-72013-1_23.

[KL07] Orna Kupferman and Yoad Lustig. "Lattice Automata". In: *Verification, Model Checking, and Abstract Interpretation, 8th International Conference, VMCAI 2007, Nice, France, January 14-16, 2007, Proceedings*. Ed. by Byron Cook and Andreas Podelski. Vol. 4349. Lecture Notes in Computer Science. Springer, 2007, pp. 199–213. DOI: 10.1007/978-3-540-69738-1_14.

[Kla+20] Michaela Klauck, Marcel Steinmetz, Jörg Hoffmann, and Holger Hermanns. "Bridging the Gap Between Probabilistic Model Checking and Probabilistic Planning: Survey, Compilations, and Empirical Comparison". In: *J. Artif. Intell. Res.* 68 (2020), pp. 247–310. DOI: 10.1613/jair.1.11595.

[KLC98] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. "Planning and Acting in Partially Observable Stochastic Domains". In: *Artif. Intell.* 101.1-2 (1998), pp. 99–134. DOI: 10.1016/S0004-3702(98)00023-X.

[Kle38] Stephen Cole Kleene. "On Notation for Ordinal Numbers". In: *J. Symb. Log.* 3.4 (1938), pp. 150–155. DOI: 10.2307/2267778.

[KLS22] Hannes Kallwies, Martin Leucker, and César Sánchez. "Symbolic Runtime Verification for Monitoring Under Uncertainties and Assumptions". In: *Automated Technology for Verification and Analysis - 20th International Symposium, ATVA 2022, Virtual Event, October 25-28, 2022, Proceedings*. Ed. by Ahmed Bouajjani, Lukás Holík, and Zhilin Wu. Vol. 13505. Lecture Notes in Computer Science. Springer, 2022, pp. 117–134. DOI: 10.1007/978-3-031-19992-9_8.

[KLS23] Hannes Kallwies, Martin Leucker, and César Sánchez. "General Anticipatory Monitoring for Temporal Logics on Finite Traces". In: *Runtime Verification - 23rd International Conference, RV 2023, Thessaloniki, Greece, October 3-6, 2023, Proceedings*. Ed. by Panagiotis Katsaros and Laura Nenzi. Vol. 14245. Lecture Notes in Computer Science. Springer, 2023, pp. 106–125. DOI: 10.1007/978-3-031-44267-4_6.

[Klu+16] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. "Jupyter Notebooks-a publishing format for reproducible computational workflows." In: *ELPUB*. 2016, pp. 87–90.

[KNP11] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. "PRISM 4.0: Verification of Probabilistic Real-Time Systems". In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 585–591. DOI: 10.1007/978-3-642-22110-1_47.

[KNP12] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. "The PRISM Benchmark Suite". In: *Ninth International Conference on Quantitative Evaluation of Systems, QEST 2012, London, United Kingdom, September 17-20, 2012*. IEEE Computer Society, 2012, pp. 203–204. DOI: 10.1109/QEST.2012.14.

[Köh+19] Maximilian A. Köhl, Kevin Baum, Markus Langer, Daniel Oster, Timo Speith, and Dimitri Bohlender. "Explainability as a Non-Functional Requirement". In: *27th IEEE International Requirements Engineering Conference, RE 2019, Jeju Island, Korea (South), September 23-27, 2019*. Ed. by Daniela E. Damian, Anna Perini, and Seok-Won Lee. IEEE, 2019, pp. 363–368. DOI: 10.1109/RE.2019.00046.

[Koz82] Dexter Kozen. "Results on the Propositional -Calculus". In: *Automata, Languages and Programming, 9th Colloquium, Aarhus, Denmark, July 12-16, 1982, Proceedings*. Ed. by Mogens Nielsen and Erik Meineche Schmidt. Vol. 140. Lecture Notes in Computer Science. Springer, 1982, pp. 348–359. DOI: 10.1007/BFB0012782.

[KPR98] Yonit Kesten, Amir Pnueli, and Li-on Raviv. "Algorithmic Verification of Linear Temporal Logic Specifications". In: *Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13-17, 1998, Proceedings*. Ed. by Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel. Vol. 1443. Lecture Notes in Computer Science. Springer, 1998, pp. 1–16. DOI: 10.1007/BFB0055036.

[Kri63] Saul Kripke. "Semantical Considerations on Modal Logic". In: *Acta Philosophica Fennica* 16 (1963), pp. 83–94.

[KRS09] Jui-Yi Kao, Narad Rampersad, and Jeffrey O. Shallit. "On NFAs where all states are final, initial, or both". In: *Theor. Comput. Sci.* 410.47-49 (2009), pp. 5010–5021. DOI: 10.1016/J.TCS.2009.07.049.

[Kup22] Orna Kupferman. "Multi-Valued Reasoning about Reactive Systems". In: *Found. Trends Theor. Comput. Sci.* 15.2 (2022), pp. 126–228. DOI: 10.1561/0400000083.

[KV01] Orna Kupferman and Moshe Y. Vardi. "Model Checking of Safety Properties". In: *Formal Methods Syst. Des.* 19.3 (2001), pp. 291–314. DOI: 10.1023/A:1011254632723.

[KW87] Johan de Kleer and Brian C. Williams. "Diagnosing Multiple Faults". In: *Artif. Intell.* 32.1 (1987), pp. 97–130. DOI: 10.1016/0004-3702(87)90063-4.

[Lar+97] K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. "Efficient verification of real-time systems: compact data structure and state-space reduction". In: *Proceedings Real-Time Systems Symposium*. 1997, pp. 14–24. DOI: 10.1109/REAL.1997.641265.

[LBW09] Jay Ligatti, Lujo Bauer, and David Walker. "Run-Time Enforcement of Nonsafety Policies". In: *ACM Trans. Inf. Syst. Secur.* 12.3 (2009), 19:1–19:41. DOI: 10.1145/1455526.1455532.

[Leu+18] Martin Leucker, César Sánchez, Torben Scheffel, Malte Schmitz, and Alexander Schramm. "TeSSLa: runtime verification of non-synchronized real-time streams". In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018*. Ed. by Hisham M. Haddad, Roger L. Wainwright, and Richard Chbeir. ACM, 2018, pp. 1925–1933. DOI: 10.1145/3167132.3167338.

[Leu+19] Martin Leucker, César Sánchez, Torben Scheffel, Malte Schmitz, and Daniel Thoma. "Runtime Verification for Timed Event Streams with Partial Information". In: *Runtime Verification - 19th International Conference, RV 2019, Porto, Portugal, October 8-11, 2019, Proceedings*. Ed. by Bernd Finkbeiner and Leonardo Mariani. Vol. 11757. Lecture Notes in Computer Science. Springer, 2019, pp. 273–291. DOI: 10.1007/978-3-030-32079-9_16.

[Leu12] Martin Leucker. "Sliding between Model Checking and Runtime Verification". In: *Runtime Verification, Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers*. Ed. by Shaz Qadeer and Serdar Tasiran. Vol. 7687. Lecture Notes in Computer Science. Springer, 2012, pp. 82–87. DOI: 10.1007/978-3-642-35632-2_10.

[Lew73] David Lewis. "Causation". In: *Journal of Philosophy* 70.17 (1973), pp. 556–567. DOI: 10.2307/2025310.

[LFM20] Giovanni Lugaresi, Nicla Frigerio, and Andrea Matta. "A New Learning Factory Experience Exploiting LEGO For Teaching Manufacturing Systems Integration". In: *Procedia Manufacturing* 45 (2020). Learning Factories across the value chain – from innovation to service – The 10th

Conference on Learning Factories 2020, pp. 271–276. ISSN: 2351-9789. DOI: https://doi.org/10.1016/j.promfg.2020.04.106.

[LS09] Martin Leucker and Christian Schallhart. "A brief account of runtime verification". In: *J. Log. Algebraic Methods Program.* 78.5 (2009), pp. 293–303. DOI: 10.1016/j.jlap.2008.08.004.

[MB06] Katell Morin-Allory and Dominique Borrione. "On-line Monitoring of Properties Built on Regular Expressions". In: *Forum on specification and Design Languages, FDL 2006, September 19-22, 2006, Darmstadt, Germany, Proceedings*. ECSI, 2006, pp. 249–255.

[MG05] H. Brendan McMahan and Geoffrey J. Gordon. "Fast Exact Planning in Markov Decision Processes". In: *ICAPS*. 2005, pp. 151–160.

[Mha+17] Lotfi Mhamdi, Chakib Ben Njima, Hedi Dhouibi, and Hassani Messaoud. "Using timed automata and fuzzy logic for diagnosis of multiple faults in DES". In: *International Conference on Control, Automation and Diagnosis, ICCAD 2017, Hammamet, Tunisia, January 19-21, 2017*. IEEE, 2017, pp. 457–463. DOI: 10.1109/CADIAG.2017.8075702.

[MHC99] Omid Madani, Steve Hanks, and Anne Condon. "On the Undecidability of Probabilistic Planning and Infinite-Horizon Partially Observable Markov Decision Problems". In: *Proceedings of the Sixteenth National Conference on Artificial Intelligence and Eleventh Conference on Innovative Applications of Artificial Intelligence, July 18-22, 1999, Orlando, Florida, USA*. Ed. by Jim Hendler and Devika Subramanian. AAAI Press / The MIT Press, 1999, pp. 541–548.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*. Vol. 92. Lecture Notes in Computer Science. Springer, 1980. ISBN: 3-540-10235-3. DOI: 10.1007/3-540-10235-3.

[Moo56] Edward F. Moore. "Gedanken-Experiments on Sequential Machines". In: *Automata Studies. (AM-34), Volume 34*. Ed. by C. E. Shannon and J. McCarthy. Princeton: Princeton University Press, 1956, pp. 129–154. ISBN: 9781400882618. DOI: doi:10.1515/9781400882618-006.

[Myh57] John Myhill. "Finite automata and the representation of events". In: *WADD Technical Report* 57 (1957), pp. 112–137.

[Ner58] A. Nerode. "Linear Automaton Transformations". In: *Proceedings of the American Mathematical Society* 9.4 (1958), pp. 541–544. ISSN: 00029939, 10886826.

[Orf95] Sophocles J Orfanidis. *Introduction to signal processing*. Prentice-Hall, Inc., 1995.

[Ped+11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. "Scikit-learn: Machine learning in Python". In: *the Journal of machine Learning research* 12 (2011), pp. 2825–2830.

[Pin+13] Luis Enrique Pineda, Yi Lu, Shlomo Zilberstein, and Claudia V. Goldman. "Fault-Tolerant Planning under Uncertainty". In: *IJCAI*. 2013, pp. 2350–2356.

[Pin+16] Srinivas Pinisetty, Viorel Preoteasa, Stavros Tripakis, Thierry Jéron, Yliès Falcone, and Hervé Marchand. "Predictive runtime enforcement". In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016*. Ed. by Sascha Ossowski. ACM, 2016, pp. 1628–1633. DOI: 10.1145/2851613.2851827.

[Pin+17a] Srinivas Pinisetty, Thierry Jéron, Stavros Tripakis, Yliès Falcone, Hervé Marchand, and Viorel Preoteasa. "Predictive runtime verification of timed properties". In: *J. Syst. Softw.* 132 (2017), pp. 353–365. DOI: 10.1016/j.jss.2017.06.060.

[Pin+17b] Srinivas Pinisetty, Partha S. Roop, Steven Smyth, Nathan Allen, Stavros Tripakis, and Reinhard von Hanxleden. "Runtime Enforcement of Cyber-Physical Systems". In: *ACM Trans. Embed. Comput. Syst.* 16.5s (2017), 178:1–178:25. DOI: 10.1145/3126500.

[PM22] Paolo Pazzaglia and Martina Maggio. "Characterizing the Effect of Deadline Misses on Time-Triggered Task Chains". In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 41.11 (2022), pp. 3957–3968. DOI: 10.1109/TCAD.2022.3199146.

[Pnu77] Amir Pnueli. "The Temporal Logic of Programs". In: *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32.

[Pop+06] Traian Pop, Paul Pop, Petru Eles, Zebo Peng, and Alexandru Andrei. "Timing Analysis of the FlexRay Communication Protocol". In: *18th Euromicro Conference on Real-Time Systems, ECRTS'06, 5-7 July 2006, Dresden, Germany, Proceedings*. IEEE Computer Society, 2006, pp. 203–216. DOI: 10.1109/ECRTS.2006.31.

[PSJ18] Sven Peldszus, Daniel Strüber, and Jan Jürjens. "Model-based security analysis of feature-oriented software product lines". In: *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE 2018. Boston, MA, USA: Association

for Computing Machinery, 2018, pp. 93–106. ISBN: 9781450360456. DOI: 10.1145/3278122.3278126.

[Pur98] Anuj Puri. "Dynamical Properties of Timed Automata". In: *Formal Techniques in Real-Time and Fault-Tolerant Systems, 5th International Symposium, FTRTFT'98, Lyngby, Denmark, September 14-18, 1998, Proceedings*. Ed. by Anders P. Ravn and Hans Rischel. Vol. 1486. Lecture Notes in Computer Science. Springer, 1998, pp. 210–227. DOI: 10.1007/BFB0055349.

[PZ14] Luis Enrique Pineda and Shlomo Zilberstein. "Planning Under Uncertainty Using Reduced Models: Revisiting Determinization". In: *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling, ICAPS 2014, Portsmouth, New Hampshire, USA, June 21-26, 2014*. Ed. by Steve A. Chien, Minh Binh Do, Alan Fern, and Wheeler Ruml. AAAI, 2014. ISBN: 978-1-57735-660-8.

[QS82] Jean-Pierre Queille and Joseph Sifakis. "Specification and verification of concurrent systems in CESAR". In: *International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings*. Ed. by Mariangiola Dezani-Ciancaglini and Ugo Montanari. Vol. 137. Lecture Notes in Computer Science. Springer, 1982, pp. 337–351. DOI: 10.1007/3-540-11494-7_22.

[Rei87] Raymond Reiter. "A Theory of Diagnosis from First Principles". In: *Artif. Intell.* 32.1 (1987), pp. 57–95. DOI: 10.1016/0004-3702(87)90062-2.

[Rin04] Jussi Rintanen. "Complexity of Planning with Partial Observability". In: *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004), June 3-7 2004, Whistler, British Columbia, Canada*. Ed. by Shlomo Zilberstein, Jana Koehler, and Sven Koenig. AAAI, 2004, pp. 345–354.

[RM87] Richard A Roberts and Clifford T Mullis. *Digital signal processing*. Addison-Wesley Longman Publishing Co., Inc., 1987.

[RN10] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach, Third International Edition*. Pearson Education, 2010. ISBN: 978-0-13-207148-2.

[RS02] C. R. Ramakrishnan and R. Sekar. "Model-based analysis of configuration vulnerabilities". In: *Journal of Computer Security* 10.1-2 (2002), pp. 189–209. DOI: 10.3233/JCS-2002-101-209.

[RS15] Enno Ruijters and Mariëlle Stoelinga. "Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools". In: *Comput. Sci. Rev.* 15 (2015), pp. 29–62. DOI: 10.1016/j.cosrev.2015.03.001.

[RS59] Michael O. Rabin and Dana S. Scott. "Finite Automata and Their Decision Problems". In: *IBM J. Res. Dev.* 3.2 (1959), pp. 114–125. DOI: 10.1147/rd.32.0114.

[RW89] Peter J. Ramadge and Walter Murray Wonham. "The control of discrete event systems". In: *Proc. IEEE* 77.1 (1989), pp. 81–98. DOI: 10.1109/5.21072.

[Sam+95] Meera Sampath, Raja Sengupta, Stéphane Lafortune, Kasim Sinnamohideen, and Demosthenis Teneketzis. "Diagnosability of discrete-event systems". In: *IEEE Trans. Autom. Control.* 40.9 (1995), pp. 1555–1575. DOI: 10.1109/9.412626.

[Sam+96] Meera Sampath, Raja Sengupta, Stéphane Lafortune, Kasim Sinnamohideen, and Demosthenis Teneketzis. "Failure diagnosis using discrete-event models". In: *IEEE Trans. Control. Syst. Technol.* 4.2 (1996), pp. 105–124. DOI: 10.1109/87.486338.

[Sco82] Dana S. Scott. "Domains for Denotational Semantics". In: *Automata, Languages and Programming, 9th Colloquium, Aarhus, Denmark, July 12-16, 1982, Proceedings*. Ed. by Mogens Nielsen and Erik Meineche Schmidt. Vol. 140. Lecture Notes in Computer Science. Springer, 1982, pp. 577–613. DOI: 10.1007/BFB0012801.

[SCT10] Andreas Schranzhofer, Jian-Jia Chen, and Lothar Thiele. "Timing Analysis for TDMA Arbitration in Resource Sharing Systems". In: *16th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2010, Stockholm, Sweden, April 12-15, 2010*. Ed. by Marco Caccamo. IEEE Computer Society, 2010, pp. 215–224. DOI: 10.1109/RTAS.2010.24.

[SFK08] Mani Swaminathan, Martin Fränzle, and Joost-Pieter Katoen. "The Surprising Robustness of (Closed) Timed Automata against Clock-Drift". In: *Fifth IFIP International Conference On Theoretical Computer Science - TCS 2008, IFIP 20th World Computer Congress, TC 1, Foundations of Computer Science, September 7-10, 2008, Milano, Italy*. Ed. by Giorgio Ausiello, Juhani Karhumäki, Giancarlo Mauri, and C.-H. Luke Ong. Vol. 273. IFIP. Springer, 2008, pp. 537–553. DOI: 10.1007/978-0-387-09680-3_36.

[SHT06] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. "Feature Diagrams: A Survey and a Formal Semantics". In: *14th IEEE International Conference on Requirements Engineering (RE 2006), 11-15 September 2006, Minneapolis/St.Paul, Minnesota, USA*. IEEE Computer Society, 2006, pp. 136–145. DOI: 10.1109/RE.2006.23.

[SKY] SKYbrary. *Electronic Centralized Aircraft Monitor (ECAM)*. URL: https://web.archive.org/web/20240421150245/https://skybrary.aero/articles/electronic-centralized-aircraft-monitor-ecam.

[Som15] Fabio Somenzi. *CUDD: CU Decision Diagram Package*. Tech. rep. University of Colorado at Boulder, 2015.

[TFC90] Jeffrey J. P. Tsai, Kwang-Ya Fang, and Horng-Yuan Chen. "A Noninvasive Architecture to Monitor Real-Time Distributed Systems". In: *Computer* 23.3 (1990), pp. 11–23. DOI: 10.1109/2.50269.

[THE93] THE COUNCIL OF THE EUROPEAN COMMUNITIES. *Council Directive 93/42/EECof 14 June 1993 concerning medical devices*. June 1993. URL: http://data.europa.eu/eli/dir/1993/42/oj (visited on 04/24/2024).

[Thr02] Sebastian Thrun. "Probabilistic robotics". In: *Commun. ACM* 45.3 (2002), pp. 52–57. DOI: 10.1145/504729.504754.

[Thü+14] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. "A Classification and Survey of Analysis Strategies for Software Product Lines". In: *ACM Comput. Surv.* 47.1 (2014), 6:1–6:45. DOI: 10.1145/2580950.

[THW94] Ken Tindell, H. Hanssmon, and Andy J. Wellings. "Analysing Real-Time Communications: Controller Area Network (CAN)". In: *Proceedings of the 15th IEEE Real-Time Systems Symposium (RTSS '94), San Juan, Puerto Rico, December 7-9, 1994*. IEEE Computer Society, 1994, pp. 259–263. DOI: 10.1109/REAL.1994.342710.

[Tri02] Stavros Tripakis. "Fault Diagnosis for Timed Automata". In: *Formal Techniques in Real-Time and Fault-Tolerant Systems, 7th International Symposium, FTRTFT 2002, Co-sponsored by IFIP WG 2.2, Oldenburg, Germany, September 9-12, 2002, Proceedings*. Ed. by Werner Damm and Ernst-Rüdiger Olderog. Vol. 2469. Lecture Notes in Computer Science. Springer, 2002, pp. 205–224. DOI: 10.1007/3-540-45739-9_14.

[Tsa+90] Jeffrey J. P. Tsai, Kwang-Ya Fang, Horng-Yuan Chen, and Yao-Dong Bi. "A Noninterference Monitoring and Replay Mechanism for Real-Time Software Testing and Debugging". In: *IEEE Trans. Software Eng.* 16.8 (1990), pp. 897–916. DOI: 10.1109/32.57626.

[TV99] Eduardo Tovar and Francisco Vasques. "Real-time fieldbus communications using Profibus networks". In: *IEEE Trans. Ind. Electron.* 46.6 (1999), pp. 1241–1251. DOI: 10.1109/41.808018.

[TYG08] David Thorsley, Tae-Sic Yoo, and Humberto E. Garcia. "Diagnosability of stochastic discrete-event systems under unreliable observations". In: *2008 American Control Conference*. 2008, pp. 1158–1165. DOI: 10.1109/ACC.2008.4586649.

[Uni] United States Environmental Protection Agency. URL: https://web.archive.org/web/20240718170737/https://www.epa.gov/greenvehicles/electric-plug-hybrid-electric-vehicles.

[VL08] Antti Valmari and Petri Lehtinen. "Efficient Minimization of DFAs with Partial Transition". In: *25th International Symposium on Theoretical Aspects of Computer Science*. Ed. by Susanne Albers and Pascal Weil. Vol. 1. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2008, pp. 645–656. ISBN: 978-3-939897-06-4. DOI: 10.4230/LIPIcs.STACS.2008.1328.

[WB95] Greg Welch and Gary Bishop. *An introduction to the Kalman filter*. 1995.

[Wul+08] Martin De Wulf, Laurent Doyen, Nicolas Markey, and Jean-François Raskin. "Robust safety of timed automata". In: *Formal Methods Syst. Des.* 33.1-3 (2008), pp. 45–84. DOI: 10.1007/S10703-008-0056-7.

[Yan96] Steve Jennhwa Yang. "Debugging for Timing-Constraint Violations". In: *IEEE Softw.* 13.2 (1996), pp. 89–99. DOI: 10.1109/52.506465.

[Zav00] Pamela Zave. "Feature-Oriented Description, Formal Methods, and DFC". In: *Language Constructs for Describing Features, Proceedings of the FIREworks Workshop, Glasgow, UK, May 2000*. Ed. by Stephen Gilmore and Mark Ryan. Springer, 2000, pp. 11–26. DOI: 10.1007/978-1-4471-0287-8_2.

[Zha+22] Peng Zhang, Shijun Zhang, Shang Li, Jin Zhang, Shaoxun Liu, and Youjun Bu. "FRA-FPGA: Fast Reconfigurable Automata Processing on FPGAs". In: *32nd International Conference on Field-Programmable Logic and Applications, FPL 2022, Belfast, United Kingdom, August 29 - Sept. 2, 2022*. IEEE, 2022, pp. 313–321. DOI: 10.1109/FPL57034.2022.00055.

[ZLD12] Xian Zhang, Martin Leucker, and Wei Dong. "Runtime Verification with Predictive Semantics". In: *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings*. Ed. by Alwyn Goodloe and Suzette Person. Vol. 7226. Lecture Notes in Computer Science. Springer, 2012, pp. 418–432. DOI: 10.1007/978-3-642-28891-3_37.

[Zon+20] Tiago Zonta, Cristiano André da Costa, Rodrigo da Rosa Righi, Miromar José de Lima, Eduardo Silveira da Trindade, and Guann-Pyng Li. "Predictive maintenance in the Industry 4.0: A systematic literature review". In: *Comput. Ind. Eng.* 150 (2020), p. 106889. DOI: 10.1016/J.CIE.2020.106889.