THEORY AND APPLICATIONS OF RUNTIME MONITORING METRIC FIRST-ORDER TEMPORAL LOGIC

Samuel Müller

DISS. ETH NO. 18445

THEORY AND APPLICATIONS OF RUNTIME MONITORING METRIC FIRST-ORDER TEMPORAL LOGIC

A dissertation
submitted to
ETH ZURICH
for the degree of
Doctor of Sciences

presented by
Samuel Müller,
Dipl. Inform. et lic. oec. publ.,
University of Zurich
born on 12 December 1979
citizen of Winterthur ZH

accepted on the recommendation of
Prof. Dr. David Basin, examiner
Prof. Dr. Birgit Pfitzmann, co-examiner
Prof. Dr. Peter Widmayer, co-examiner

2009

Abstract

RUNTIME MONITORING is a systematic approach to verifying system properties at execution time by using an algorithm to check whether a system execution satisfies a temporal property. Whereas novel application areas such as compliance or business activity monitoring require expressive property specification languages, current monitoring techniques are restricted in the properties they can handle. They either support properties expressed in propositional temporal logics and thus cannot cope with variables ranging over infinite domains, do not provide both universal and existential quantification or only in restricted ways, do not allow arbitrary quantifier alternation, cannot handle unrestricted negation, do not provide quantitative temporal operators, or cannot simultaneously handle both past and future temporal operators.

In this dissertation, we present a runtime monitoring approach for an expressive fragment of metric first-order temporal logic (MFOTL) that overcomes all these limitations. The fragment consists of formulae of the form $\Box\,\phi$, where ϕ is bounded, i.e., its temporal operators refer only finitely into the future. Our monitor uses automatic structures to finitely represent infinite structures, which allows for the unrestricted use of negation and quantification in monitored formulae. Moreover, our monitor supports the arbitrary nesting of both past and bounded future operators. This means that complex properties can be specified more naturally than with only past operators.

We also show how to adapt our monitoring approach to the common case where all relations are required to be finite and hence relational databases can serve as an alternative to automata. Under the additional restriction that time increases after at most a fixed number of time points, our incremental construction ensures that our monitor requires only polynomial space in the cardinality of the data appearing in the processed prefix of the monitored timed temporal structure. This is in contrast to complexity results for other approaches, such as a logical data expiration technique proposed for two-sorted first-order logic (2-FOL). While this logic is at least as expressive as MFOTL, the space required for monitoring 2-FOL formulae is non-elementary in the cardinality of the data in the processed prefix.

We prototypically implemented our approach for the setting with finite relations and validated its practical feasibility by means of a statistical steady-state analysis. Moreover, we further assessed the practical usefulness of our monitoring approach in two case studies in the areas of compliance and separation of duty.

Kurzfassung

RUNTIME MONITORING ist ein Verfahren zur algorithmischen Überprüfung, ob eine beobachtete Systemausführung eine temporale Eigenschaft erfüllt. Obwohl neue Anwendungen wie Compliance oder Geschäftsaktivitätsüberwachung erhöhte Ausdrucksmächtigkeit erfordern, sind existierende Verfahren diesbezüglich eingeschränkt. Sie unterstützen nur Eigenschaften, welche ohne Variablen über unendlichen Wertebereichen definiert werden können, bieten keine oder nur eingeschränkte universelle und existentielle Quantifikation, haben Probleme mit Negation, unterstützen keine quantitativen temporalen Operatoren oder lassen nicht gleichzeitig temporale Vergangenheits- und Zukunfts-Operatoren zu.

Diese Dissertation beschreibt ein Runtime Monitoring-Verfahren für ein ausdrucksmächtiges Fragment metrischer Temporallogik erster Stufe (MFOTL), das alle obgenannten Einschränkungen beseitigt. Das Fragment besteht aus Formeln der Art $\Box\, \phi$, wobei ϕ begrenzt ist, d.h. alle Operatoren referenzieren Zeitpunkte, welche höchstens endlich weit in der Zukunft liegen. Unser Verfahren repräsentiert unendliche Strukturen durch endliche Zustandsautomaten und erlaubt dadurch die uneingeschränkte Verwendung von Negation und Quantifikation. Zudem unterstützt das Verfahren die beliebige Verschachtelung von Zukunfts- und Vergangenheits-Operatoren und damit die natürliche Spezifikation komplexer Eigenschaften.

Für den verbreiteten Fall mit endlichen Relationen erläutern wir die nötigen Anpassungen, um relationale Datenbanken anstelle von Zustandsautomaten zu verwenden. Dann zeigen wir, dass der Speicherbedarf unseres Verfahrens (unter einer technischen Annahme) polynomiell in der Kardinalität der Menge der bereits beobachteten Datenwerte begrenzt ist. Dies steht im Gegensatz zu Komplexitätsresultaten alternativer Verfahren wie beispielsweise einer Datenexpirations-Technik für Logik erster Stufe über zwei Trägermengen (2-FOL). Während 2-FOL mindestens so ausdrucksmächtig ist wie MFOTL, so ist der Speicherbedarf zur Überwachung beliebiger 2-FOL-Formeln nicht-elementar in der Kardinalität der Menge der Datenwerte im bereits verarbeiteten Präfix der überwachten temporalen Struktur.

Für den Fall mit endlichen Relationen haben wir einen Prototypen unseres Verfahrens in Java entwickelt. Dessen praktischer Nutzen wurde mittels einer statistischen Stationäritäts-Analyse sowie anhand zweier Fallstudien in den Bereichen Compliance und Separation of Duty (d.h. organisatorische Aufgabentrennung) validiert.

Acknowledgments

This dissertation would not exist without the great support and continual inspiration that I received from many colleagues, friends, and family members. On these few lines, I want to express my deep gratitude to all those who supported me on this endeavor.

First of all, I thank David Basin for giving me the opportunity of pursuing this dissertation, for his instructive comments, and his generous support. Likewise, I thank Birgit Pfitzmann for introducing me to the world of research and her valuable feedback to numerous drafts of my work. I also want to express my appreciation to Peter Widmayer for his great flexibility and his willingness to serve as my co-examiner.

Second, I thank my IBM friends and colleagues for making the past four years such a tremendous and lasting experience. Specifically, I thank my former office mates Dieter Sommer, Alexandru Caracas, and Thomas Groß for many inspiring and conspiring discussions; Samuel Burri, Christopher Giblin, and Günter Karjoth for sharing their perspectives on compliance, separation of duty, and more worldly topics; Ulrich Schimpel and Richard Bödi for their statistical advice and discussions on the nature of risk; all my jogging partners for many breathtaking discussions; the ZRL predocs for their empathic company; and Andreas Wespi and Matthias Schunter for their support.

Third and to no lesser extent, I want to thank my current and former research colleagues at ETH for maintaining a pleasant and inspiring work environment and for providing valuable suggestions to further improve my work. I am particularly deeply indebted to Felix Klaedtke, whose rigorous and diligent reviews of many of my drafts helped me in making my ideas more precise. Felix not only was an invaluable sparring partner in many long and fruitful discussions, he is a true scientific role model and a great colleague. I am also grateful to Christof Roduner for numerous motivating discussions and for sharing many joys, tears, and beers.

Last and closest to my heart, I want to thank those many people around me who mostly work in professions other than research but are such invaluable sources of mental support and continual inspriation. In particular, I thank my parents, my sisters, my close friends, and other relatives for their unconditional support, their sustained confidence in my abilities, and simply for being there. A special thank goes to Colin Schälli for helping me with the cover page. My final and most heartfelt thank goes to Rahel, whose limitless love, care, and understanding supplied me with the exhaustless amount of energy that I required to complete this work.

Contents

List of Figures

List of Tables

"There is nothing permanent except change."

Heraclitus

Chapter 1

Introduction

COMPUTER systems pervade all aspects of our society. Ranging from communication networks to personal consumer goods, they enable real-time communication across geographies, ensure timely payments of goods and services, and simplify our daily lives to a large extent. While differing in implementation technology, scale, or application domain, most of these computer systems share one or several of the following basic attributes:

- *Reactive behavior:* The system is embedded in and interacts with a changing environment for an unknown, possibly infinite amount of time. For example, a web server responds to user requests as long as its administrator does not stop it or until it crashes.
- *Unbounded state space:* Depending on data variables ranging over infinite domains, the state space used by a computer system over time often cannot be sensibly bounded a priori. For example, a transaction processing system may have to deal with a possibly infinite number of transactions and data elements over time.
- *Time-constrained operation:* Computer-based applications are often subject to timing constraints. For example, a web server is usually expected to respond to a request within a given time frame, say within at most 30 seconds.
- *Distributed architecture:* In our networked world and even on single platforms, computer systems are often distributed in the sense that different parts of a computation are performed by different physical or logical units. Moreover, some computations may be performed in parallel. For example, in a service-oriented architecture, services are hosted on different servers and may perform their computations simultaneously.

As computer systems continue to represent, support, or enable ever larger parts of our daily lives, we increasingly depend on their correct operation. The ability to verify or guarantee that a system satisfies certain properties, i.e., that it operates according to its specification, is thus essential.

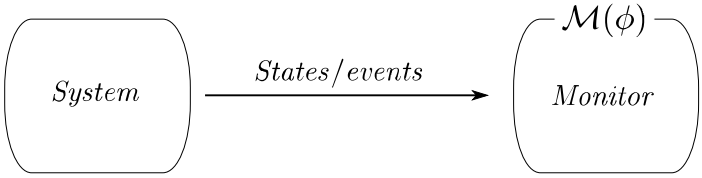


Figure 1.1: Conceptual overview of runtime monitoring.

There is a broad spectrum of techniques to assess whether a system correctly implements its specification. On one end of the spectrum, there are manual reviews and testing. These techniques are typically informal, have incomplete coverage, and only provide limited guarantees regarding the correctness of the analyzed system. On the other end of the spectrum, we find verification techniques such as model checking (i.e., algorithmic verification) and theorem proving (i.e., deductive verification). These approaches can provide comprehensive correctness guarantees. However, they both require a complete formal model of the system under scrutiny—often too strong a requirement for a real-world system. Moreover, the practical application of model checking is essentially limited to finite state systems. In this dissertation, we thus investigate another technique, which resides in the middle of the spectrum and integrates the simplicity of testing with the rigor and strength of automated verification: *runtime monitoring*.

1.1 The Runtime Monitoring Problem

Runtime monitoring is a systematic approach to verifying system properties at execution time by using an algorithm to check whether a system execution satisfies a temporal property. By verifying properties at execution time, the approach contrasts with model checking or deductive verification techniques, where properties are verified before the system under scrutiny is executed. Because runtime monitoring only verifies individual executions, it is often called a light-weight verification technique.

Conceptually, a typical (online) runtime monitoring approach works as follows: To begin with, a property ϕ is formally specified using an appropriate specification language. Often, a variant of temporal logic or a similar declarative language is used for this purpose. In a next step, a runtime monitor $\mathcal{M}(\phi)$ is automatically generated from the formal property specification. The generated runtime monitor is then executed in parallel with the system under scrutiny. As the monitored system changes its state or generates new events, the runtime monitor incrementally consumes this information and determines at each time point whether the system satisfies or violates the monitored property (see Figure 1.1).

A rigorous approach to runtime monitoring temporal properties of a reactive system thus requires that the following aspects be addressed:

1. *Execution model $\mathcal{E}$:* First, an adequate formal model for representing system executions along with an appropriate model of time must be selected.

2. *Property specification language $\mathcal{L}$:* Second, a language to formally and unambiguously specify temporal system properties must be selected. Both the syntax and the semantics of the language $\mathcal{L}$ must be formally defined with respect to the chosen execution model.

3. *Monitoring method $\mathcal{M}$:* Finally, a method for determining whether arbitrary system executions (represented in the selected execution model) satisfy a temporal property (formulated in the selected property specification language) must be provided.

In other words, runtime monitoring is the problem of checking whether or not a property ϕ, formulated in a specification language $\mathcal{L}$, is satisfied by a system execution, expressed in an execution model $\mathcal{E}$. Any correct monitoring method $\mathcal{M}$ for a given execution model $\mathcal{E}$ and a property specification language $\mathcal{L}$ then constitutes a valid solution to the runtime monitoring problem.

1.2 A Glimpse at Applications

Runtime monitoring is more than a program verification technique. Indeed, methods to verify properties at execution time have many interesting applications that go beyond system verification. For example, runtime monitoring can be applied in the following contexts:

- *Verification:* In the area of software program verification, runtime monitoring can complement static verification techniques in situations where model checking fails (e.g., due to an infinite state space or the unavailability of a complete system specification). In cases where model checking is feasible for the control part of the system, runtime monitoring may be useful to dynamically check the integrity of the data-dependent part of the system. Moreover, in the area of testing, runtime monitoring approaches can generate reliable test oracles for use during simulations.

- *Compliance, security, and dependability:* The areas of compliance, security, or dependability often require the monitoring of temporal system properties during system execution. For example, runtime monitoring techniques can enable the continuous auditing of compliance requirements, the enforcement of history-based access control or dynamic separation of duty rules, and the monitoring of dynamic intrusion detection rules.

- *Business intelligence:* Making intelligent decisions often relies on the ability to analyze and detect temporal properties over continuous streams of input data. Being able to quickly detect temporal properties and react upon them in an adequate manner can even provide a competitive advantage, for example, in the area of algorithmic stock trading. Here, runtime monitoring offers methods and tools for the specification and detection of complex properties in data streams.

1.3 A Running Example

To motivate our approach and illustrate a possible application context, we now introduce a running example.

1.3.1 Scenario and Assumptions

Let us consider a reactive transaction processing system (TPS) that processes customer transactions of a financial institution. We make the following assumptions: At each time point, the TPS processes an arbitrary number of transactions. Every so often, some of the transactions (e.g., those whose financial value exceeds a certain threshold) are reported as suspicious. Transactions and reports may be processed in parallel and neither the number of transactions, customers, nor reports can be bounded a priori. As the TPS uses unique identification numbers (IDs) as representations of the respective real-world objects, we need to consider an infinite domain of such transaction IDs, customer IDs, and report IDs. We further assume that the time between any two consecutive customer transactions or reports can vary from being arbitrarily small to an arbitrarily large but finite amount of time.

1.3.2 An Example Property

We expect the TPS to satisfy certain correctness properties. Specifically, we shall consider the following property, which is motivated by anti-money laundering regulations such as the Bank Secrecy Act [BSA70] or the USA Patriot Act [Pat01]:

> *Every transaction t of a customer c, who has within the last 30 days been involved in a suspicious transaction t', must be reported as suspicious. Suspicious transactions must be reported within 2 days.*

To verify whether the TPS satisfies this property, we use runtime monitoring. If at any time point during system execution, the property is indeed violated, further steps may be taken. For example, the monitoring system can notify another system component or a person to take corrective action, execute a predefined operation, or possibly stop the processing of further transactions altogether.

Throughout this dissertation, we shall revisit this running example several times. In particular, when presenting different models of system execution, we demonstrate how to represent executions of the TPS. Moreover, in the context of our discussion of different property specification languages, we show how to formally and unambiguously express our example property. Finally, when presenting and validating our runtime monitoring approach for complex system properties, the running example will often serve as a concrete illustration.

1.4 Contributions

The core contributions of this dissertation are the following:

- We provide an approach for runtime monitoring complex system properties formulated within an expressive safety fragment of metric first-order temporal logic over timed temporal (automatic) first-order structures. The supported fragment allows for arbitrary nesting of both temporal past and bounded future operators. Moreover, by using automatic structures, the fragment makes no restrictions on the use of negation and quantification in monitored formulae.

  Details are presented in Chapter 4 and were published in [BKMP08a].

- We provide a special treatment of our approach for the setting where all relations are finite. In particular, we show how to restrict negation and quantification in monitored formulae such that relational databases can be used as an efficient alternative to automata for implementing our monitoring approach. Moreover, we extended an existing rewrite procedure to handle a richer class of temporal formulae. Despite the restricted use of negation and quantification limitation, the resulting fragment still handles arbitrary nesting of temporal past and bounded future operators. Furthermore, we present space optimizations and show that the space consumption of our monitoring approach is polynomially bounded by the cardinality of the data appearing in the processed prefix.

  Details are presented in Chapter 5 and were published in [BKMP08a, BKMP08b].

- We practically validated our monitoring approach for finite relations based on a prototypical implementation in Java. In particular, we investigated the practical feasibility of our approach by means of a general analysis and two case studies in the areas of compliance and separation of duty as follows:

  - We conducted a general analysis of the steady-state mean space consumption and the steady-state mean processing performance of our approach for monitoring several practically relevant formula classes and found strong evidence for their practical feasibility.

    Details are presented in Chapter 6.

- Our first case study demonstrates that complex compliance requirements can be expressed as bounded metric first-order temporal logical formulae and monitored using our monitor for finite relations.

  Details are presented in Chapter 7.

- Our second case study shows that several variants of separation of duty constraints can be expressed in past-only (metric) first-order temporal logic. Moreover, we demonstrate that simple dynamic separation of duty constraints and object-based separation of duty constraints can be practically monitored using our monitor for finite relations.

  Details are presented in Chapter 8.

Earlier work that influenced the case studies was reported in [GLM+05, GMP06, Mül06, AvKM+06, AYL06].

In addition to the above contributions, this dissertation makes explicit the strong ties between various streams of research, notably software program monitoring, dynamic integrity checking for databases, monitoring for security and compliance, and data stream management. In particular, we provide an overview of common concepts and a classification of related approaches from all the above research fields in terms of a common typology.

1.5 Structure

This dissertation consists of a theoretical and a practical part. In the following, we give a brief summary of what lies ahead.

Part I: Theory

In the theoretical part, we address the theory of runtime monitoring complex system properties with a focus on properties expressed in metric first-order temporal logic.

Chapter 2 presents the mathematical notation, technical preliminaries, and relevant previous results. These definitions and results are necessary to understand the contents and results of the work at hand. Section 2.1 fixes the notation and reviews basic mathematical concepts. In Section 2.2, we summarize elementary automata-theoretic definitions and results. In Section 2.3, we introduce a general model to formally capture untimed and timed system behavior and relate it to alternative models. In Section 2.4, we define the notion of a property and review several formalisms for property specification. In particular, we introduce *metric first-order temporal logic*, an expressive

formalism for specifying temporal system properties. Finally, in Section 2.5, we summarize automatic structures and alternative means for the finite representation of infinite structures.

Chapter 3 provides an overview of the field of runtime monitoring. After providing definitions and sketching basic concepts in Section 3.1, we present a classification of existing runtime monitoring approaches in Section 3.2. We discuss different types of applications of runtime monitoring in Section 3.3 and conclude with a brief summary in Section 3.4.

Chapter 4 is the theoretical core chapter of this dissertation. It introduces a novel monitoring approach for complex properties expressed using metric first-order temporal logic. In Section 4.1, we first describe the necessary restrictions required to apply our approach. Section 4.2 then presents a high-level overview of our approach. In Sections 4.3 and 4.4, we provide the full theoretical details of our incremental monitoring technique. Subsequently, in Section 4.5, we illustrate our constructions by walking through an example step by step. Finally, in Section 4.6, we present our algorithm and prove its correctness. The chapter ends with a summary in Section 4.7.

Chapter 5 discusses a practically relevant special case of our runtime monitoring approach, namely the case where relations are always finite. In Section 5.1, we first present an overview and sketch the difficulties induced by finite relations. Section 5.2 then explains how the notion of domain independence as known from database theory is extended to our temporal setting. Subsequently, in Section 5.3, we provide an approach based on rewriting to ensure that relations always remain finite. Optimizations of our approach are presented in Section 5.4. Finally, in Section 5.5, we prove that the space consumption our runtime monitoring approach is polynomially bounded by the cardinality of the data in the processed prefix. We then discuss our results in the context of related work and conclude the chapter with a discussion in Section 5.6.

Part II: Applications

In the practical part, we present how we validated our monitoring approach. We first describe our prototypical monitoring framework that implements our monitoring approach for finite relations. We then evaluate the practical feasibility of the approach by means of a general analysis and two short case studies.

Chapter 6 describes our prototypical monitoring framework and presents general results of an experimental validation of our implementation. In Section 6.1, we summarize the architecture of the monitoring framework. In Section 6.2, we then describe our validation methodology and present the results of an experimental analysis of the

space consumption of our monitors for several formula classes. The chapter concludes with a summary and some further remarks in Section 6.3.

Chapter 7 presents the results of our first case study. In Section 7.1, we clarify our objectives and present a set of regulatory requirements as an example. In Section 7.2, we demonstrate how the presented requirements can be formalized by means of bounded metric first-order temporal logical formulae. In Section 7.3, we investigate to what extent the formalized requirements can be monitored with our monitoring framework. We conclude the chapter with a summary in Section 7.4.

Chapter 8 reports the results of our second case study. In Section 8.1, we state our objectives, review the notion of role-based access control, and give an brief overview of different variants of separation of duty. In Section 8.2, we demonstrate how separation of duty constraints can be formalized by means of past-only and TSF domain independent MFOTL formulae. In Section 8.3, we then investigate whether monitoring the formalized constraints is practically feasible. We conclude the chapter with a summary in Section 8.4.

Chapter 9 discusses related work. In Section 9.1, we give a general overview and summarize the limitations of logics supported by competing runtime monitoring approaches. In Section 9.2, we review related work from the area of software program monitoring in more detail. In Section 9.3, we distinguish our approach from work in the area of dynamic integrity checking for databases. Finally, in Section 9.4, we review and distinguish our approach from related work in the field of data stream management.

Chapter 10 concludes the dissertation. We first summarize our core results in Section 10.1 before we discuss future work and open problems in Section 10.2.

Part I

Theory

"[...] nanos gigantium humeris insidentes [...]"

Bernard de Chartres

Chapter 2

Background

THIS chapter presents the mathematical notation, concepts, and tools used throughout this dissertation. We first review some basic mathematical concepts and fix our notation. We then present a brief summary of standard automata theory along with results that we shall use in the forthcoming chapters. In the subsequent section, we introduce a general model of timed and untimed system behavior and highlight that several commonly used models of system execution are special cases thereof. We then introduce the notion of a property and review alternative means to specify properties. In particular, we present metric first-order temporal logic, our formalism of choice for specifying temporal system properties. Finally, we review automatic structures and explain how infinite structures can be finitely represented using finite state automata.

2.1 Basic Definitions and Notation

Let us recall the following set-theoretical notation and terminology.

2.1.1 Sets, Sequences, and Tuples

We make standard use of sets and sequences (see, e.g., [Sip96] or [Tru99]). Set membership is expressed using the symbols $\in$ and $\notin$. We write $\emptyset$ to denote the empty set. For sets A and B, we write $\mathsf{A} \subseteq \mathsf{B}$ if A is a *subset* of B and $\mathsf{A} \subsetneq \mathsf{B}$ if A is a *proper subset* of B. We write $\mathcal{P}(\mathsf{A})$ or 2^{A} to denote the *power set* of A. We write $\mathsf{A} \cup \mathsf{B}$ for the *union* and $\mathsf{A} \cap \mathsf{B}$ for the *intersection* of A and B. The *cardinality* of A, written $|\mathsf{A}|$, is the number of elements contained in the set. The set A is called a *singleton* if $|\mathsf{A}| = 1$. The sets A and B are called *disjoint* if $\mathsf{A} \cap \mathsf{B} = \emptyset$.

We write $\mathbb{N}$ for the set of natural numbers, $\mathbb{Q}$ for the set of rational numbers, and $\mathbb{R}$ for the set of real numbers. Let $\mathbb{I}$ be the set of nonempty intervals over $\mathbb{N}$. We often write an interval in $\mathbb{I}$ as $[c, d)$, where $c \in \mathbb{N}$, $d \in \mathbb{N} \cup \{\infty\}$, and $c < d$, i.e, $[c, d) := \{a \in \mathbb{N} \mid c \leq a < d\}$.

We represent finite or infinite sequences by writing the objects in it within parentheses, separated by commas. For example, we write (a, c, b) to denote the sequence of the elements a, c, and b. Finite sequences are also called *tuples*. A sequence with k elements is called an k*-tuple* or a sequence of *length* k. A 2-tuple is also called a *pair*, a 3-tuple a *triple*, a 4-tuple a *quadruple*, etc. The *empty tuple* (also called *empty word*) ϵ is the special sequence of length 0 that does not contain any elements.

For two sets A and B, the *cross product* or *Cartesian product*, denoted by $\mathsf{A}\times\mathsf{B}$, is the set of all pairs (a, b), where $a \in \mathsf{A}$ and $b \in \mathsf{B}$. Likewise, for k sets $\mathsf{A}_1, \ldots, \mathsf{A}_k$, the Cartesian product, written $\mathsf{A}_1 \times \ldots \times \mathsf{A}_k$, is the set of all k-tuples $(a_1, \ldots, a_k)$ with $a_i \in \mathsf{A}_i$ for all i with $1 \leq i \leq k$. To express the Cartesian product of a set A with itself, instead of writing $\underbrace{\mathsf{A} \times \cdots \times \mathsf{A}}_{k}$, we often use the shorthand A^k.

2.1.2 Words and Languages

An *alphabet* Σ is an finite set of *symbols*. A *word over* Σ (also called a string) is a finite or infinite sequence of symbols from Σ. The *length* $|w|$ of a word w is the number of symbols $k \geq 0$ occurring in the sequence. We write $|w| = \infty$ if the length of a word w is infinite. Infinite words are also called ω-words. We denote as Σ^* and Σ^ω the sets of all finite and infinite words, respectively, as built from symbols in Σ including the empty word ϵ. We also define $\Sigma^\infty := \Sigma^* \cup \Sigma^\omega$. Note that we typically write words without the parentheses and commas used to denote sequences. Consider the following example. Let $\Sigma = \{0, 1\}$ be an alphabet. The set Σ^∞ then includes the words $\{\epsilon, 0, 1, 01, 10, \ldots\}$. The *lexicographic ordering* of words is the same as the dictionary ordering with the exception that shorter words precede longer words.

For a word $w \in \Sigma^\infty$, we write $w_0 w_1 \cdots$ to denote its elements. We write $w[i, j]$ with $i, j \in \mathbb{N}$ to denote the word (i.e., subsequence) $w_i \cdots w_j$ and $w[i, \infty]$ to denote the subsequence $w_i w_{i+1} \cdots$. For $w \in \Sigma^*$ and $\beta \in \Sigma^\infty$, we often use juxtaposition and write $w\beta$ to denote concatenation of w and β. A word w is called a prefix or suffix of a word β if there exists a sequence α such that $\alpha = w\beta$ or $\alpha = \beta w$, respectively. We sometimes use the same notion and terminology also for sequences other than words.

A *language* L over an alphabet Σ is a subset of Σ^∞. A *finitary* language over Σ is a subset of Σ^* and an ω*-language* over Σ is a subset of Σ^ω.

Languages can be manipulated using specific operations. Let L and L' be languages. We define $L \cup L' = \{w \mid w \in L \text{ or } w \in L'\}$ (*union*), $L \cap L' = \{w \mid w \in L \text{ and } w \in L'\}$ (*intersection*), $\bar{L} = \{w \mid w \notin L\}$ (*complementation*), $L \backslash L' = \{w \mid w \in L \text{ and } w \notin L'\}$ (*difference*), $L \circ L' = \{uv \mid u \in L \text{ and } v \in L'\}$ (*concatenation*), and $L^* = \{w_1 w_2 \ldots w_k \mid k \geq 0 \text{ and each } w_i \in L\}$ (*Kleene star*).

2.1.3 Timed Words and Timed Languages

A *timed word* over some alphabet Σ is a pair $(\bar{w}, \bar{t})$, where $\bar{w} = w_0 \cdots w_k$ is a finite word over Σ and $\bar{t} = (t_0, \ldots, t_k)$ is a *monotonous* sequence of real-numbered *time stamps*, i.e., $t_i \in \mathbb{R}$ and $t_i \leq t_{i+1}$, for all $i, 0 \leq i < k$. Similarly, an *infinite timed word* or *timed ω-word* over Σ is a pair (w, t), where $w = w_0 w_1 \cdots$ is an ω-word over Σ and $t = (t_0, t_1, \ldots)$ is an infinite sequence of real-numbered *time stamps*, i.e., $t_i \in \mathbb{R}$, for all $i \geq 0$. The sequence t is monotonous, i.e., $t_i \leq t_{i+1}$, for all $i \geq 0$, and *progressing*, i.e., for every $t' \in \mathbb{R}$, there is some $i \leq 0$ such that $t_i > t'$. As an alternative, finite and infinite timed words are sometimes also defined with the natural numbers as time stamps. The respective definitions are analogous. Choosing a monotonically increasing sequence of real numbers as time stamps results in a proper *real-time semantics*. In contrast, the choice of a monotonically increasing sequence of natural numbers gives rise to a so-called *fictitious clock semantics* [AH92]. In a fictitious clock semantics, time stamps are recorded with finite precision only but two symbols sharing the same time stamps may still be distinguished by their temporal ordering.

Instead of as a pair of sequences, we often also write finite or infinite timed words as sequences of pairs. For example, for the timed ω-word (w, t) with $w = w_0 w_1 \cdots$ and $t = (t_0, t_1, \ldots)$ we write $(w_0, t_0)(w_1, t_1) \ldots$.

A *timed language* over Σ is a set of timed words over Σ. A *timed ω-language* over Σ is a set of timed ω-words over Σ.

2.1.4 Relations and Functions

A (k-*ary*) *relation* $\mathsf{R} \subseteq \mathsf{A}_1 \times \cdots \times \mathsf{A}_k$ is a set of k-tuples, with $a_i \in \mathsf{A}_i$ for all i with $1 \leq i \leq k$. We call k the *arity* of R and say a relation is *nullary*, *unary*, or *binary* if $k = 0$, $k = 1$, or $k = 2$, respectively. If R is a binary relation, we often write $a_1 \mathsf{R}\, a_2$ instead of $(a_1, a_2) \in \mathsf{R}$. Moreover, for $(a_1, \ldots, a_k) \in \mathsf{R}$, we sometimes also write $R(a_1, \ldots, a_k) = \mathit{true}$. We also use the term *predicate* as a synonym of relation.

A mathematical *function* or *mapping* is a special type of binary relation, where no element of the domain occurs more than once as the first element of any pair contained in it. Formally, a relation $\mathsf{R} \subseteq \mathsf{A} \times \mathsf{B}$ is a function if $(a, b) \in \mathsf{R}$ and $(a, c) \in \mathsf{R}$ implies $b = c$, for all pairs in R. We often use the italicized letters f, g, or h to denote functions and write $f : \mathsf{A} \to \mathsf{B}$ to specify that f is a function mapping elements in the domain A to a subset of B. Moreover, we write $f(a) = b$ to express that $(a, b) \in f$. The function $f : \mathsf{A} \to \mathsf{B}$ is called *surjective* if for every $b \in \mathsf{B}$ there is an element $a \in \mathsf{A}$ such that $b = f(a)$. And it is called *injective* if for all $a, a' \in \mathsf{A}$ $f(a) = f(a')$ implies $a = a'$. If f is both injective and surjective, f is also called *bijective*.

2.1.5 Signatures and First-order Structures

A (first-order) *signature* S is a quadruple $(\mathsf{C}, \mathsf{R}, \mathsf{F}, a)$, where C is a set of constant symbols, R is a finite set of relation symbols, F is a finite set of function symbols, and $a : \mathsf{R} \cup \mathsf{F} \to \mathbb{N}$ associates each relation or function symbol $s \in \mathsf{R} \cup \mathsf{F}$ with an arity $a(s) \in \mathbb{N}$. Instead of the function a, we sometimes also use a superscript to indicate the arity of a relation or function symbol. For example, we write r^3 to indicate that the arity associated with some relation symbol r is 3.

Let $S = (\mathsf{C}, \mathsf{R}, \mathsf{F}, a)$ be a signature. A *(first-order) structure* D over S, also called an S-structure, consists of a domain $|D| \neq \emptyset$, *interpretations* $c^D \in |D|$ and $r^D \subseteq |D|^{a(r)}$, for each $c \in \mathsf{C}$ and $r \in \mathsf{R}$, and functions $f : |D|^{a(f)} \to |D|$ for each $f \in \mathsf{F}$. A *relational* structure is a first-order structure without functions. We obtain the *relational variant* of each structure by replacing each function $f : |D|^{a(f)} \to |D|$ by its *graph* $G_f := \{(a_1, \ldots, a_{a(f)}, b) \in |D|^{a(f)+1} \mid f(a_1, \ldots, a_{a(f)}) = b\}$. As functions can be represented as relations, for the rest of this dissertation, we only consider relational structures and simply write $(\mathsf{C}, \mathsf{R}, a)$ for a signature.

In a relational database context, a signature is called a *(relational) database schema*. A *relational database* is a first-order structure over a relational database schema S and a (data) domain, where all relations are finite. The *active domain* $adom(D)$ of a relational database D over a relational schema $S = (\mathsf{C}, \mathsf{R}, a)$ is the set of constants and domain elements appearing in its relations, i.e., $adom(D) := \{c^D \mid c \in \mathsf{C}\} \cup \bigcup_{r \in \mathsf{R}} \{d_j \mid (d_1, \ldots, d_{a(r)}) \in r^D \text{ and } 1 \leq j \leq a(r)\}$ [Cod70, Cho94, AHV95].

2.2 Automata Theory

Let us also recall some basic definitions and results from automata theory.

2.2.1 Finite State Automata and Regular Languages

A (deterministic) *finite state automaton* (DFA) is a quintuple $(\mathsf{Q}, \Sigma, \delta, q_0, \mathsf{F})$, where Q is a finite set of states, Σ is an alphabet, $\delta : \mathsf{Q} \times \Sigma \to \mathsf{Q}$ is a transition function, $q_0 \in \mathsf{Q}$ is the start state, and $\mathsf{F} \subseteq \mathsf{Q}$ is the set of final states. We now define what it means for a DFA to accept a word. Let $\mathcal{A} = (\mathsf{Q}, \Sigma, \delta, q_0, \mathsf{F})$ be a finite state automaton and $w = w_1 w_2 \cdots w_n$ a word over Σ. Then $\mathcal{A}$ *accepts* w if and only if there is a sequence of states, called a *run*, $(r_0, r_1, \ldots, r_n)$ with $r_i \in \mathsf{Q}$ such that $r_0 = q_0$, $\delta(r_i, w_{i+1}) = r_{i+1}$, for all $i \in \{0, \ldots, n-1\}$, and $r_n \in \mathsf{F}$. The language *recognized* by $\mathcal{A}$ is $L(\mathcal{A}) := \{w \mid \mathcal{A} \text{ accepts } w\}$.

A *nondeterministic finite state automaton (NFA)* is a quintuple $(\mathsf{Q}, \Sigma, \delta, q_0, \mathsf{F})$, where Q is a finite set of states, Σ is an alphabet, $\delta : \mathsf{Q} \times \Sigma \to 2^{\mathsf{Q}}$ is a transition function, $q_0 \in \mathsf{Q}$ is the start state, and $\mathsf{F} \subseteq \mathsf{Q}$ is the set of final states. Let $\mathcal{A} = (\mathsf{Q}, \Sigma, \delta, q_0, \mathsf{F})$ be an NFA and $w = w_1 w_2 \cdots w_n$ a word over Σ. Then $\mathcal{A}$ *accepts* w if and only if there is a *run*

$(r_0, r_1, \ldots, r_n)$ with $r_i \in \mathsf{Q}$ such that $r_0 = q_0$, $r_{i+1} \in \delta(r_i, w_{i+1})$, for $i \in \{0, \ldots, n-1\}$, and $r_n \in \mathsf{F}$. The language *recognized* by $\mathcal{A}$ is $L(\mathcal{A}) := \{w \mid \mathcal{B} \text{ accepts } w\}$.

Let $\mathcal{B}$ be an NFA. There is a DFA $\mathcal{A}$ such that $L(\mathcal{A}) = L(\mathcal{B})$. A language L is called *regular* if there is a DFA $\mathcal{A}$ such that $L = L(\mathcal{A})$. If L and L' are regular languages, then so are $L \cup L$, $L \cap L$, $\bar{L}$, $L \backslash L'$, $L \circ L'$, and L^*. We say that the class of regular languages is *closed* under these operations. We remark that the operations are *effective*, i.e., there are algorithms to construct the regular languages resulting from applying the operations. For more details or proofs, see, e.g., [Sip96].

2.2.2 Büchi Automata and ω-Regular Languages

We also review the notion of a (nondeterministic) *Büchi automaton (NBA)*, which is a nondeterministic finite automaton taking *infinite* words as inputs [Büc62].

A *(nondeterministic) Büchi automaton* is a quintuple $(\mathsf{Q}, \Sigma, \delta, q_0, \mathsf{F})$, where Q is a finite set of states, Σ is an alphabet, $\delta : \mathsf{Q} \times \Sigma \to 2^{\mathsf{Q}}$ is a transition function, $q_0 \in \mathsf{Q}$ is an initial state, and $\mathsf{F} \subseteq \mathsf{Q}$ is a set of accepting states. Let $\mathcal{A} = (\mathsf{Q}, \Sigma, \delta, q_0, \mathsf{F})$ be a Büchi automaton. Then $\mathcal{A}$ is called a *deterministic Büchi automaton (DBA)* if and only if for all $q \in \mathsf{Q}$ and $a \in \Sigma$ it holds that $|\delta(q, a)| = 1$.

Let $\mathcal{A} = (\mathsf{Q}, \Sigma, \delta, q_0, \mathsf{F})$ be a nondeterministic Büchi automaton and $w = w_1 w_2 \cdots$ over Σ be an infinite word, i.e., $w \in \Sigma^\omega$. Then $\mathcal{A}$ *accepts* w if and only if there is a *run* $r = (r_0, r_1, \ldots)$ with $r_i \in \mathsf{Q}$ such that $r_0 = q_0$, $r_{i+1} \in \delta(r_i, w_{i+1})$, for all $i \in \mathbb{N}$, and $\mathsf{Inf}(r) \cap \mathsf{F} \neq \emptyset$, where $\mathsf{Inf}(r)$ denotes the set of states in r that are visited infinitely often.

A language $L \subseteq \Sigma^\omega$ is called ω-regular if it is recognized by some nondeterministic Büchi automaton. Apart from Büchi automata, other types of finite state automata, in particular, Rabin automata, Streett automata, parity automata, and Muller automata, also recognize ω-regular languages. Their main distinguishing feature with respect to Büchi automata is that they use different acceptance conditions. Because all these automata accept ω-words, they are also referred to as ω-automata. Note that the class of deterministic Büchi automata does not recognize all ω-regular languages.

The class of ω-regular languages is closed under union, intersection, and complementation. For more details and proofs, see, e.g., [Tho90, Muk96].

2.2.3 Timed Automata and Timed ω-Regular Languages

While ω-automata are prominent tools for the verification and *qualitative* reasoning about infinite behaviors of finite state reactive systems, *timed automata* provide a theory for modeling and verifying *quantitative* aspects of timed systems [AD90, AD94, BY03]. Essentially, a timed automaton is a Büchi automaton extended with a set of real-valued variables modeling clocks. Constraints on these variables are then used to restrict the set of timed words accepted by the automaton and Büchi acceptance conditions are used to enforce progress properties.

We now present a formal definition of timed automata. We first introduce clock constraints. Let X be a set of clock variables. The set $\Phi(\mathsf{X})$ of *clock constraints* δ is defined inductively as follows: For $x \in \mathsf{X}$ and $c \in \mathbb{Q}$, $(x \leq c)$ and $(c \leq x)$ are clock constraints. If δ_1 and δ_2 are clock constraints, then $(\neg\delta_1)$ and $(\delta_1 \wedge \delta_2)$ are clock constraints.

A *clock assignment* for a set of clock variables X assigns a real number to every $x \in \mathsf{X}$. A clock assignment ν for X satisfies a clock constraint δ over X if and only if δ evaluates to true using the values given by ν. For $t \in \mathbb{R}$, $\nu + t$ denotes the clock assignment that maps every clock $x \in \mathsf{X}$ to $\nu(x) + t$. Likewise, the clock assignment $t \cdot \nu$ assigns to each clock $x \in \mathsf{X}$ the value $t \cdot \nu(x)$. For $\mathsf{Y} \subseteq \mathsf{X}$, $\nu[\mathsf{Y}/t]$ is the clock assignment assigning t to each $x \in \mathsf{Y}$ and agrees with ν over the rest of the clocks.

A *timed (Büchi) automaton* is a tuple $(\mathsf{Q}, \Sigma, q_0, \mathsf{X}, \Delta, \mathsf{F})$, where Q is a finite set of states, Σ is an alphabet, $q_0 \in \mathsf{Q}$ is an initial state, X is a finite set of clock variables, $\Delta \subseteq \mathsf{Q} \times \mathsf{Q} \times \Sigma \times 2^{\mathsf{X}} \times \Phi(\mathsf{X})$ is a transition relation, and $\mathsf{F} \subseteq \mathsf{Q}$ is a set of accepting states. An element $(q, q', a, \lambda, \delta) \in \Delta$ represents a transition from state q to state q' upon reading the input symbol a. The set $\lambda \subseteq \mathsf{X}$ identifies the set of clocks to be reset and δ is a clock constraint over X.

We can now define what it means for a timed Büchi automaton to accept a timed ω-word. Let $\mathcal{A} = (\mathsf{Q}, \Sigma, q_0, \mathsf{X}, \Delta, \mathsf{F})$ be a timed Büchi automaton and (w, t), with $w = w_1 w_2 \cdots$ and $t = (t_1, t_2, \dots)$ a timed ω-word over Σ. Then $\mathcal{A}$ *accepts* (w, t) if and only if there is a *run* (r, n) with $r = (r_0, r_1, \dots)$, $r_i \in \mathsf{Q}$, and $n = (\nu_0, \nu_1, \dots)$ with each ν_i being a clock assignment, for all $i \geq 0$, such that $r_0 = q_0$ and $\nu_0(x) = 0$ for all $x \in \mathsf{X}$, $(r_i, r_{i+1}, w_{i+1}, \lambda_{i+1}, \delta_{i+1}) \in \Delta$ such that $(\nu_{i+1} + t_{i+1} - t_i)$ satisfies δ_{i+1} and ν_{i+1} equals $(\nu_{i+1} + t_{i+1} - t_i)[\lambda_{i+1}/0]$, for all $i \geq 0$, and $\mathsf{Inf}(r) \cap \mathsf{F} \neq \emptyset$, where $\mathsf{Inf}(r)$ denotes the set of states in r that are visited infinitely often.

A timed ω-language L is called timed ω-regular if it is recognized by a timed Büchi automaton. The class of timed ω-regular languages is closed under union and intersection but not under complementation [AD94].

2.3 Models of System Behavior

We now describe an untimed and a timed variant of a general model for representing executions of reactive systems. We model executions (or traces) of a reactive system by infinite (timed) sequences of first-order structures—so-called (timed) temporal first-order structures—over some signature S. This model is specifically suited for representing possibly infinite executions of *infinite* state systems. Many other popular models of system execution are special cases of our model.

2.3.1 Temporal First-order Structures

Roughly speaking, a temporal (first-order) structure over a signature S, also called a temporal S-structure, is an infinite sequence of first-order structures over S.

Definition 2.3.1. *Let $S = (\mathsf{C}, \mathsf{R}, a)$ be a signature. A* temporal (first-order) structure *over S is an infinite sequence $D = (D_0, D_1, \dots)$ of structures over S, where:*

1. *D has constant domains, i.e., $|D_i| = |D_{i+1}|$, for all $i \geq 0$. We denote the domain by $|D|$ and require that $|D|$ is linearly ordered by the relation $<$.*
2. *Each constant symbol $c \in \mathsf{C}$ has a rigid interpretation, i.e., $c^{D_i} = c^{D_{i+1}}$, for all $i \geq 0$. We denote the interpretation of c by c^D.*

At each time point, the relations of a first-order structure store those tuples that are true at the given time point. Under the closed world assumption, at each time point only domain values stored in a relation are considered true at this time point [Ull88].

In contrast to other models of system execution such as ω-words, note that a temporal structure can represent a possibly infinite number of observable states. To understand how a temporal structure can represent executions of an infinite state system, consider Example 2.3.2.

Example 2.3.2 (Running example)**.** *For the sake of concreteness, we consider our running example from Section 1.3. To formally capture the processed transactions and reports, we define the signature of the* TPS *as $S_{\text{TPS}} = (\mathsf{C}, \mathsf{R}, a)$, where $\mathsf{R} = \{trans, report\}$ and $a(trans) = a(report) = 2$. Figure 2.1 displays an example temporal structure $D = (D_0, D_1, \dots)$ over S_{TPS}, representing an infinite execution of the* TPS*. The domain $|D|$ consists of the set of strings over the alphabet of ASCII characters. At each time point $j \geq 0$, the structure D_j consists of the two relations $trans^{D_j}$ and $report^{D_j}$ (represented as tables in Figure 2.1), which provide interpretations for the predicate symbols $trans$ and $report$, respectively. For example, the relation $trans^{D_0}$ models the fact that the system processed the transactions 34IKU for customer Bob and 13IRN for customer Alice at the time point 0. Likewise, at the time point $i = 1$, the relation $report^{D_1}$ captures the fact that the transaction 34IKU of customer Bob was reported as suspicious. Note that the empty relation $report^{D_0}$ represents the information that at the time point 0 the system did not process any report for any customer.*

2.3.2 Timed Temporal First-order Structures

In a temporal structure, time is modeled qualitatively, i.e., by the relative ordering of individual structures in the infinite sequence. In many situations, however, also *quantitative* information about the time that elapses between any two time points in an execution is required. To address this requirement, we must also record the physical times at which events or states occur. We assume that in a reactive system, the time difference between any two adjacent time points may vary from zero to an arbitrarily large but finite amount of time.

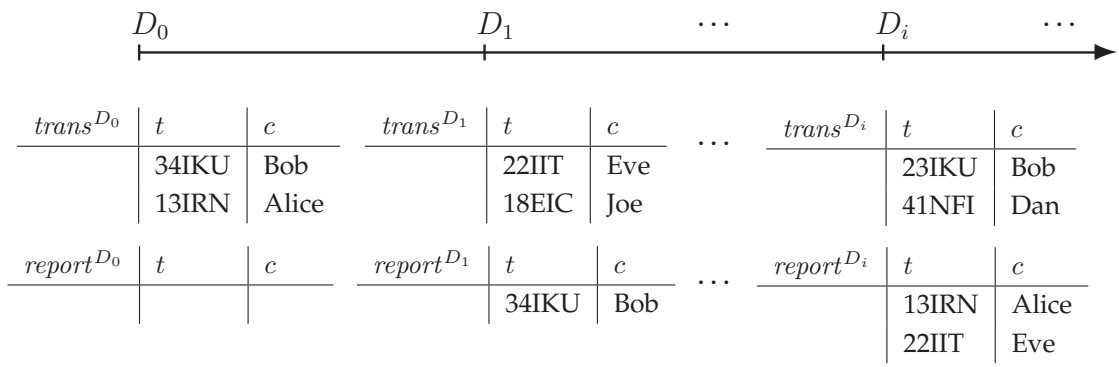


Figure 2.1: Example of a temporal structure over S_{TPS}.

To represent the behavior of a reactive system together with the physical times at which states hold or events appear, we introduce *timed temporal (first-order) structures*.

Definition 2.3.3. *Let $S = (\mathsf{C}, \mathsf{R}, a)$ be a signature. A* timed temporal (first-order) structure *over S, also called a timed temporal S-structure, is a pair (D, τ), where $D = (D_0, D_1, \dots)$ is a temporal structure over S and $\tau = (\tau_0, \tau_1, \dots)$ is a sequence of time stamps $\tau_i \in \mathbb{N}$ with the following properties:*

1. *Monotonicity: $\tau_i \leq \tau_{i+1}$, for all $i \geq 0$.*
2. *Progress: For every $i \geq 0$, there is some $j > i$ such that $\tau_j > \tau_i$.*

Note that our choice of a monotonous sequence of natural numbers as time stamps yields a fictitious clock semantics. Other time domains such as the sets of rational or real numbers, respectively, would have been possible options, too. Our choice of $\mathbb{N}$ as the time domain, has both practical and theoretical reasons. While physical time is essentially continuous, practically we can measure it with finite precision only. Moreover, the choice of $\mathbb{N}$ paired with the assumptions of monotonicity and progress allow for conveniently modeling sequences of structures where the time stamps of adjacent time points may be the same or lie arbitrarily but finitely far apart. Complementing these practical aspects, in Section 5.5, we also give a theoretical justification for our choice.

The time semantics of timed temporal structures as given in Definition 2.3.3 is *point-based*. This means that we associate with each time point i in a timed temporal structure D a single time stamp $\tau_i \in \mathbb{N}$. As an alternative, one might also chose an *interval-based* time semantics, where each structure D_i in D is associated with an interval from the appropriate time domain. While the point-based semantics is more adequate for modeling system events, the interval-based semantics lends itself better to the representation of system states. We remark that (untimed) temporal structures are equally suitable for modeling both sequences of states or events. See [AH92] and [HR04c] for an in-depth discussion of this and other alternatives.

Example 2.3.4 (Running example)**.** *Let us again consider our running example. A concrete example of a timed temporal structure representing an execution the system* TPS *would be the*

pair (D, τ), where $D = (D_0, D_1, D_2, D_3, \dots)$ is a temporal structure over the signature S as defined in Example 2.3.2 and $\tau = (0, 3, 3, 7, \dots)$ is a sequence of time stamps. In addition to the qualitative temporal ordering between individual structures in D, the sequence of time stamps τ now associates with each time point i also a time stamp τ_i. For example, the structure D_0 has an associated time stamp of 0, whereas the structures D_1 and D_2, while linearly ordered, share the same (finite-precision) time stamp with the value 3.

2.3.3 Special Cases

Many other models of system execution are special cases of our general model. We discuss some important special cases below.

Temporal Databases and Database Histories

If all relations are finite, a temporal first-order structure is also known as an abstract temporal snapshot database [CT05].

Definition 2.3.5. *Let $S = (\mathsf{C}, \mathsf{R}, a)$ be a signature. An* (abstract) temporal (snapshot) database *is a temporal first-order structure $D = (D_0, D_1, \dots)$ over S where all r^{D_i}s are finite, for all $r \in \mathsf{R}$ and $i \geq 0$.*

In other words, a temporal database is a infinite sequence of relational databases over the same database schema.

Apart from abstract temporal snapshot databases, there are other types of temporal databases. For example, so-called abstract temporal timestamp databases are defined by augmenting all tuples in relations by an additional temporal attribute. The resulting relations are called *temporal relations*. It is easy to see, however, that each abstract temporal timestamp database over a signature $S = (\mathsf{C}, \mathsf{R}, a)$, with $a(r) \geq 1$ for all $r \in \mathsf{R}$, can be mapped into equivalent abstract temporal snapshot database over a signature $S' = (\mathsf{C}', \mathsf{R}', a')$, where $\mathsf{C}' = \mathsf{C}$, $\mathsf{R}' = \mathsf{R}$, and $a'(r) = a(r) - 1$ for all $r \in \mathsf{R}$. Other types of temporal databases consider multi-dimensional time and cannot be seen as special cases of our execution model. This dissertation focusses on temporal data with one time dimension. Hence, we do not discuss these models here and merely refer to [ÖS95] and [CT98] for more details.

If both the sequence and the relations are finite, a temporal first-order structure is also known as a database history [CT98].

Definition 2.3.6. *A* database history *is a* finite-length *abstract temporal snapshot database.*

In other words, a database history is a finite sequence of relational databases over the same database schema. Intuitively speaking, a database history models the evolution of a relational database as induced by updates applied to it.

Data Streams

The field of *data stream management*, also referred to as complex event processing, has recently experienced a surge of interest, see, e.g. [BBD+02, GO03]. While no uniform consensus regarding the formal semantics of data streams has yet emerged, a data stream is typically defined as a possibly infinite sequence of k-tuples with associated time stamps.

Definition 2.3.7. *Let $S = (\mathsf{C}, \mathsf{R}, a)$ with $\mathsf{R} = \{r\}$, $a(r) = k$, and $k > 0$ be a signature. A* data stream *is a timed first-order temporal structure over S with singleton interpretations for r at each time point, i.e., $|r^{D_i}| = 1$, for all $i \geq 0$.*

In other words, a data stream is a timed temporal structure over a signature containing exactly one k-ary predicate r, whose interpretation r^{D_i}, at each time point i, is a singleton relation of arity k, i.e., a relation containing exactly one k-tuple, for all $i \geq 0$.

ω-Words and Timed ω-Words

In the automated verification community, system executions are typically represented by (timed) ω-words over an alphabet $\Sigma = 2^{\mathsf{AP}}$, where AP is a finite set of proposition symbols. This representation, however, is only feasible to represent infinite behaviors of *finite* state systems or infinite behaviors of infinite state systems with a finite set of observable propositions only. In fact, ω-words and timed ω-words over a finite alphabet are a special case of our general execution model, too.

Proposition 2.3.8. *Let $S = (\mathsf{C}, \mathsf{R}, a)$ be a signature with $a(r) = 0$, for all $r \in \mathsf{R}$. Each temporal structure over S can equivalently be expressed as an* ω-word *over $\Sigma = 2^{\mathsf{R}}$.*

The validity of Proposition 2.3.8 can be seen as follows: if all relation symbols of a signature are nullary, the signature merely specifies a finite set of proposition symbols. Each finite set of proposition symbols R can also be expressed by an alphabet $\Sigma = 2^{\mathsf{R}}$. At each time point i, a temporal structure $D = (D_0, D_1, \dots)$ over S associates a truth value with each of its nullary predicate symbols. Specifically, a propositional symbol $r \in \mathsf{R}$ is interpreted as true at the time point i if the relation r^{D_i} contains the empty tuple ϵ. If r^{D_i} is empty, the proposition symbol is interpreted as false. Equivalently, a timed ω-word $w = w_0 w_1 \cdots$ contains a symbol $w_i \in \Sigma$, where w_i specifies the set of those proposition symbols $r \in \mathsf{R}$ that are true at i. The choice of w_i thus provides an encoding for which of the proposition symbols in R are to be interpreted as true at i.

Similarly, a timed ω-word is a timed temporal first-order structure over a signature with nullary relational symbols only.

Proposition 2.3.9. *Let $S = (\mathsf{C}, \mathsf{R}, a)$ be a signature with $a(r) = 0$, for all $r \in \mathsf{R}$. Each timed temporal structure over S can equivalently be expressed as a timed ω-word over $\Sigma = 2^{\mathsf{R}}$.*

Note that in the automated verification community, timed ω-words are often defined with time stamps from a real-numbered time domain. Because timed temporal structures are defined with natural-numbered time stamps, only timed ω-words defined with natural-numbered time stamps are a special case of our execution model. It is easy to see, however, that timed ω-words with time stamps from a time domain other than the natural numbers are special cases of timed temporal first-order structures with the appropriate time semantics.

2.4 Property Specification

In the previous section, we presented several ways to represent executions of reactive systems. In this section, we first explain how *policies* and *properties* can be understood as sets of system executions. In the subsequent sections, we then review a range of different formalisms to specify such properties. Our main focus lies on temporal logic. In particular, we present *metric first-order temporal logic*, an expressive formalism to specify complex properties of timed temporal structures. We also briefly sketch some alternatives to temporal logics to specify system properties.

2.4.1 Policies and Properties

For the purpose of this section, executions of reactive systems are represented by infinite sequences of states or events. The concrete representation of the elements in these sequences is not important. For example, an execution of the system TPS could be represented as a timed temporal structure over the signature S_{TPS} (see Example 2.3.2). Let S denote the set of all states or events. We write S^* and S^ω to denote the sets of all possible finite and infinite executions, respectively. For a reactive system T, the set $\Sigma_T \subseteq \mathsf{S}^\omega$ denotes the executions corresponding to T.

We now define the notions of a policy and a property. In particular, we recall the formal characterizations of safety and liveness properties and highlight some important results. See [Lam77, AS85, Sch00] for more details.

Policy

In computer security, verification, or requirements engineering, a specification of the forbidden, permitted, or required system behavior is often called a *policy*. Semantically, a policy $\Pi \subseteq S^\omega$ then is simply the set of all system executions that are forbidden, permitted, or required. It is formally specified by defining a predicate P on sets of executions. A system T then is said to *satisfy* a policy Π if and only if $P(\Sigma_T) = \mathit{true}$, or equivalently, if and only if $\Sigma_T \subseteq \Pi$.

Two example applications for security policies in the above sense are access control and information flow. In access control, a policy typically constrains the set of opera-

tions that subjects may perform on objects. In information flow, a policy restricts how much information a principal may learn about system objects from observing system executions.

The above definition of a policy is very general. While this generality allows for capturing most typical requirements, it also rules out the possibility to verify whether or not a system satisfies an arbitrary policy by observing individual executions only. To enable verification based on single executions, a policy must be a property.

Property

A *property* is a special kind of policy. Specifically, a property is a set of system executions that allows for determining set membership by looking at each execution alone.

Definition 2.4.1. *Let P be a predicate on sets of executions. The policy Π defined by P is called a* property *if there is a predicate $\hat{P}$ on individual executions such that $\hat{P}(\sigma) = true$, for all $\sigma \in \Pi$.*

For a given predicate $\hat{P}$ defined on individual executions, we say that $\hat{P}$ *induces* or *defines* the property Π. We say that an execution $\sigma \in \mathsf{S}^\omega$ *satisfies* the property Π if $\sigma \in \Pi$.

While access control policies are properties, information flow policies are not. The reason is that the former can be decided by observing single system executions, whereas the latter may depend on the set of all possible system executions.

Two main types of properties can be distinguished [Lam77, AS85]: *safety properties* and *liveness properties*.

Safety Properties. Intuitively, a safety property stipulates that "nothing bad happens" at any time point during an execution. In other words, for a property Π to be a safety property it must be the case that if Π is not satisfied by some execution, there must be a specific time point when the "bad thing" happens.

Before giving a formal definition of safety properties, we introduce the concept of good and bad prefixes [KV01].

Definition 2.4.2. *Let $\Pi \subseteq \mathsf{S}^\omega$ be a property. A finite sequence $\bar{\sigma} \in \mathsf{S}^*$ is called*

- *a* bad prefix *for Π if we have that $\bar{\sigma}\sigma \notin \Pi$ for all $\sigma \in \mathsf{S}^\omega$, or*
- *a* good prefix *for Π if we have that $\bar{\sigma}\sigma \in \Pi$ for all $\sigma \in \mathsf{S}^\omega$.*

Observe that every finite extension of a bad or good prefix is again a bad or good prefix, respectively. A bad or good prefix $\bar{\sigma}$ for a property Π is *minimal* if there is no proper prefix of $\bar{\sigma}$ that is also a bad or good prefix for Π, respectively.

With this, we can now formally define safety properties as follows.

Definition 2.4.3. *A property $\Pi \subseteq \mathsf{S}^\omega$ is called a* safety property *if for all $\sigma \notin \Pi$ there is a finite sequence $\bar{\sigma} \in \mathsf{S}^*$ which is a bad prefix for Π.*

In other words, a property Π is a safety property if and only if it can be characterized by the set of finite prefixes of all executions excluded from Π. Obviously, every finitary property $\Pi \subseteq S^*$ is a safety property.

A classical example of a safety property is mutual exclusion, i.e., the requirement that "two processes must not enter some critical section at the same time". Each execution that does not satisfy this property, i.e., each execution of a system where at some point in time two processes execute in some critical section simultaneously, has a bad prefix. In this example, the bad prefix is the finite sequence up to the point where the two processes entered the critical section. Note that this requirement actually expresses an *invariant property*. Invariant properties are a subclass of safety properties, whose occurrence can be checked by looking at a single execution step (e.g., a single state). Observe that many access control policies represent invariant properties, too. As an example of a safety property that is not an invariant property, consider the following *history-based* access control policy: "money can only be withdrawn once a correct PIN has been provided". Here, observing only the most recent step in a finite execution sequence does not suffice to detect an occurrence of this property. Also previous execution steps must be considered instead.

Liveness Properties. Intuitively, a *liveness property* stipulates that "something good" eventually happens during an execution [Lam77, AS85]. In other words, a liveness property asserts that no finite execution is irremediable in that the required "good thing" may always still occur later in the future.

More formally, liveness properties are characterized as follows.

Definition 2.4.4. *A property Π is a liveness property if and only if for all finite sequences $\sigma \in \mathsf{S}^*$ there is a sequence $\tau \in \mathsf{S}^\omega$ such that $\sigma\tau \in \Pi$.*

A classical example of a liveness property is starvation freedom, i.e., the requirement that "a process makes progress infinitely often". In contrast to safety properties, finite executions are never irremediable as the "good thing" defined by some liveness property (e.g., make progress infinitely often) might still occur in the future.

Other Properties. Many properties are neither safety nor liveness properties. For example, the property "eventually an event of type T_2 will happen and all preceding events are of type T_1" is an intersection of a liveness property and a safety property. In fact, it can be shown that *every* property is an intersection of a safety and a liveness property [AS85]. In this sense, safety and liveness properties provide an essential characterization of all properties.

While liveness properties have no bad prefixes (by definition), observe that some liveness properties have good prefixes though. These are called co-safety properties.

Definition 2.4.5. *A property* $\Pi \subseteq \mathsf{S}^\omega$ *is called a* co-safety property *if for all* $\sigma \in \Pi$ *there is a finite sequence* $\bar{\sigma} \in \mathsf{S}^*$ *which is a good prefix for* Π.

For example, the property "eventually the system makes progress" is a co-safety property. While violations of this property cannot be observed in finite time, any execution that satisfies this property does so in finite time. Note that not all co-safety properties are liveness properties [BJ08].

2.4.2 Propositional Temporal Logics

Propositional temporal logics are a convenient and well understood means for specifying properties of reactive systems in a declarative way. Its most prominent representative, propositional linear temporal logic, was first introduced by Prior [Pri67] in the context of language theory. Pnueli then pioneered its application to the specification of reactive systems and their properties [Pnu77].

In this section, we present an overview of several flavors of propositional temporal logic. More specifically, we first review linear temporal logic, which is interpreted over infinite words. We then present metric temporal logic and timed propositional temporal logic, which are both interpreted over timed ω-words.

Linear Temporal Logic

Propositional linear temporal logic (LTL) is a popular tool for the specification of finite state reactive systems. We now present linear temporal logic.

Syntax. Let AP be a finite set of atomic propositions. The syntax of LTL formulae is inductively defined as follows.

Definition 2.4.6. *Each* $p \in \mathsf{AP}$ *is an* LTL *formula. If* ϕ *and* ψ *are* LTL *formulae, then* $(\neg\phi)$, $(\phi \wedge \psi)$, $(\bullet\, \phi)$, $(\bigcirc\, \phi)$, $(\phi\, \mathcal{S}\, \psi)$, *and* $(\phi\, \mathcal{U}\, \psi)$ *are* LTL *formulae.*

The temporal operators $\bullet$ (previous) and $\mathcal{S}$ (since) are called *past* operators, whereas the temporal operators $\bigcirc$ (next) and $\mathcal{U}$ (until) are called *future* operators.

Semantics. The semantics of LTL is defined over ω-words over the alphabet $\Sigma = 2^{\mathsf{AP}}$.

Definition 2.4.7. *Let* $w = w_0 w_1 \ldots$ *be an infinite word over the alphabet* $\Sigma = 2^{\mathsf{AP}}$, $i \in \mathbb{N}$, *and*

ϕ an LTL *formula. The relation $(w,i) \models_{\text{LTL}} \phi$ is inductively defined as follows:*

$$
\begin{array}{ll}
(w,i) \models_{\text{LTL}} p & \textit{iff } p \in w_i \\
(w,i) \models_{\text{LTL}} (\neg\phi) & \textit{iff } (w,i) \not\models_{\text{LTL}} \phi \\
(w,i) \models_{\text{LTL}} (\phi_1 \wedge \phi_2) & \textit{iff } (w,i) \models_{\text{LTL}} \phi_1 \textit{ and } (w,i) \models_{\text{LTL}} \phi_2 \\
(w,i) \models_{\text{LTL}} (\bullet\, \phi_1) & \textit{iff } i > 0 \textit{ and } (w,i-1) \models_{\text{LTL}} \phi_1 \\
(w,i) \models_{\text{LTL}} (\circ\, \phi_1) & \textit{iff } (w,i+1) \models_{\text{LTL}} \phi_1 \\
(w,i) \models_{\text{LTL}} (\phi_1\, \mathcal{S}\, \phi_2) & \textit{iff for some } j \leq i,\ (w,j) \models_{\text{LTL}} \phi_2, \\
 & \quad \textit{and } (w,k) \models_{\text{LTL}} \phi_1, \textit{ for all } k \in [j+1, i+1) \\
(w,i) \models_{\text{LTL}} (\phi_1\, \mathcal{U}\, \phi_2) & \textit{iff for some } j \geq i,\ (w,j) \models_{\text{LTL}} \phi_2, \\
 & \quad \textit{and } (w,k) \models_{\text{LTL}} \phi_1, \textit{ for all } k \in [i,j)
\end{array}
$$

Notation and Terminology. It is standard to define additional connectives as syntactic sugar, e.g., $(\phi \vee \psi) := (\neg((\neg\phi) \wedge (\neg\psi)))$ (disjunction) and $(\phi \rightarrow \psi) := ((\neg\phi) \vee \psi)$ (implication). In addition, it is convenient to define the temporal operators $(\Diamond\, \phi) := (\mathit{true}\, \mathcal{U}\, \phi)$ (eventually), $(\Box\, \phi) := (\neg(\Diamond(\neg\phi)))$ (always), $(\blacklozenge\, \phi) := (\mathit{true}\, \mathcal{S}\, \phi)$ (once), and $(\blacksquare\, \phi) := (\neg(\blacklozenge(\neg\phi)))$ (always in the past), where true abbreviates $(p \vee (\neg p))$, for some $p \in \mathsf{AP}$.

To simplify the presentation, standard conventions concerning the binding strength of operators are used to omit parentheses. For example, $\neg$ binds stronger than $\wedge$, which binds stronger than $\vee$. Moreover, temporal operators bind weaker than Boolean connectives.

We denote as *past-only* LTL or *future-only* LTL the fragments of LTL, where besides the Boolean connectives only the past or the future temporal operators, respectively, are used to construct formulae. We shall also use the same convention to refer to past-only and future-only fragments of all other temporal logics discussed throughout this dissertation. We remark that in the literature the abbreviation LTL is normally used to refer to the future-only fragment, whereas the acronym LTL+PAST is used to refer to the full fragment.

Example 2.4.8. *Let $\mathsf{AP} = \{p, q\}$ be a set of propositions. The property "it is always the case that every p is eventually followed by a q" can be expressed by the* LTL *formula*

$$\Box\, p \rightarrow (\Diamond\, q).$$

Note that this formula defines a liveness property. This can be seen as follows: Let $\bar{w} \in \Sigma^*$ be a finite word over Σ. Independent on whether p occurs in $\bar{w} \in \Sigma^*$, we can extend $\bar{w}$ with an ω-word $w = w_0 w_1 \ldots$ over Σ, with $q \in w_j$ for all $j \geq 0$, such that $(\bar{w}w, 0) \models_{\text{LTL}} \Box\, p \rightarrow (\Diamond\, q)$.

Metric Temporal Logic

Metric temporal logic (MTL) is an extension of linear temporal logic with the capability of expressing quantitative timing constraints as required for the specification of

real-time systems [Koy90, AH92]. Specifically, MTL extends LTL with metric temporal operators that constrain the temporal operators by intervals over the natural numbers.

Syntax. We now formally define the syntax of MTL. Let AP be a finite set of propositions and let $\mathbb{I}$ be the set of nonempty intervals over $\mathbb{N}$.

Definition 2.4.9. *Each $p \in \mathsf{AP}$ is an* MTL *formula. For $I \in \mathbb{I}$, if ϕ and ψ are* MTL *formulae, then $(\neg\phi)$, $(\phi \wedge \psi)$, $(\bullet_I \phi)$, $(\bigcirc_I \phi)$, $(\phi \,\mathcal{S}_I\, \psi)$, and $(\phi \,\mathcal{U}_I\, \psi)$ are* MTL *formulae.*

Semantics. We define the semantics of MTL with respect to timed ω-words over the alphabet $\Sigma = 2^{\mathsf{AP}}$. Note that we use timed ω-words with a fictitious clock semantics in our presentation. Other semantics are possible too. See Section 2.1.3 for details.

Definition 2.4.10. *Let (w, t) with $w = w_0 w_1 \dots$ and $\tau = (t_0, t_1, \dots)$ be an infinite timed word over the alphabet $\Sigma = 2^{\mathsf{AP}}$, ϕ an* MTL *formula, and $i \in \mathbb{N}$. The relation $(w, t, i) \models_{\text{MTL}} \phi$ is defined as follows:*

$$
\begin{array}{ll}
(w,t,i) \models_{\text{MTL}} p & \textit{iff } p \in \sigma_i \\
(w,t,i) \models_{\text{MTL}} (\neg\phi) & \textit{iff } (w,t,i) \not\models_{\text{MTL}} \phi \\
(w,t,i) \models_{\text{MTL}} (\phi_1 \wedge \phi_2) & \textit{iff } (w,t,i) \models_{\text{MTL}} \phi_1 \textit{ and } (w,t,i) \models_{\text{MTL}} \phi_2 \\
(w,t,i) \models_{\text{MTL}} (\bullet_I \phi_1) & \textit{iff } i > 0,\ \tau_i - \tau_{i-1} \in I, \textit{ and } (w,t,i-1) \models_{\text{MTL}} \phi_1 \\
(w,t,i) \models_{\text{MTL}} (\bigcirc_I \phi_1) & \textit{iff } \tau_{i+1} - \tau_i \in I \textit{ and } (w,t,i+1) \models_{\text{MTL}} \phi_1 \\
(w,t,i) \models_{\text{MTL}} (\phi_1 \,\mathcal{S}_I\, \phi_2) & \textit{iff for some } j \le i,\ \tau_i - \tau_j \in I,\ (w,t,j) \models_{\text{MTL}} \phi_2, \\
 & \quad \textit{and } (w,t,k) \models_{\text{MTL}} \phi_1, \textit{ for all } k \in [j+1, i+1) \\
(w,t,i) \models_{\text{MTL}} (\phi_1 \,\mathcal{U}_I\, \phi_2) & \textit{iff for some } j \ge i,\ \tau_j - \tau_i \in I,\ (w,t,j) \models_{\text{MTL}} \phi_2, \\
 & \quad \textit{and } (w,t,k) \models_{\text{MTL}} \phi_1, \textit{ for all } k \in [i, j)
\end{array}
$$

Notation and Terminology. Analogous to the case of LTL, additional Boolean and temporal connectives are defined as syntactic sugar in the standard way. Moreover, additional metric operators are defined as follows: $(\Diamond_I \phi) := (\mathit{true}\, \mathcal{U}_I\, \phi)$, $(\Box_I \phi) := (\neg(\Diamond_I(\neg\phi)))$, $(\blacklozenge_I \phi) := (\mathit{true}\, \mathcal{S}_I\, \phi)$, and $(\blacksquare_I \phi) := (\neg(\blacklozenge_I(\neg\phi)))$, where true again abbreviates $(p \vee (\neg p))$. Also the non-metric variants of the temporal operators are easily defined, e.g., $(\Box\, \theta) := (\Box_{[0,\infty)}\, \theta)$. Again, standard conventions concerning the binding strength of operators can be used to omit parentheses.

Example 2.4.11. *Let $\mathsf{AP} = \{p, q\}$ be a set of propositions. The property "it is always the case that every p is followed by a q within at most 5 time steps" can be expressed by the* MTL *formula*

$$\Box\, p \rightarrow (\Diamond_{[0,6)}\, q).$$

Properties defined by formulae of this kind are called time-bounded liveness properties. *Note that, in contrast to the formula of Example 2.4.8, they define safety properties.*

Timed Propositional Temporal Logic

In contrast to MTL, where metric temporal operators are used to express quantitative timing constraints, timed propositional temporal logic (TPTL) uses *freeze* quantification to achieve the same goal. Because freeze quantification is essentially a restricted form of first-order quantification for time variables, TPTL is sometimes also called a half-order logic [AH92, AH94].

Syntax. We now formally define the syntax of TPTL. Let AP be a finite set of propositions and X be a finite set of clock variables.

Definition 2.4.12. *Each $p \in \mathsf{AP}$ is a* TPTL *formula. If $x \in \mathsf{X}$, $c \in \mathbb{N}$, then c and $x + c$ are timing expressions. If π_1 and π_2 are timing expressions, then $(\pi_1 \approx \pi_2)$ and $(\pi_1 \prec \pi_2)$ are* TPTL *formulae. If $x \in \mathsf{X}$ and ϕ is a* TPTL *formula, then $(x.\phi)$, $(\neg\phi)$, $(\bullet\, \phi)$, and $(\bigcirc\, \phi)$ are* TPTL *formulae. If ϕ_1 and ϕ_2 are* TPTL *formulae, then $(\phi_1 \wedge \phi_2)$, $(\phi_1\, \mathcal{S}\, \phi_2)$, and $(\phi_1\, \mathcal{U}\, \phi_2)$ are* TPTL *formulae.*

Semantics. The semantics of TPTL formulae is defined with respect to timed ω-words over the alphabet $\Sigma = 2^{\mathsf{AP}}$. Again, we use a fictitious clock semantics. Let $\alpha : \mathsf{X} \to \mathbb{N}$ be a clock valuation. For $x \in \mathsf{X}$ and $c \in \mathbb{N}$, we have $\alpha(x + c) = \alpha(x) + c$ and $\alpha(c) = c$. Moreover, $\alpha[x/c]$ is the clock valuation that maps value c to the clock variable x and leaves the other variables unaltered.

Definition 2.4.13. *Let (w, t) with $w = w_0 w_1 \ldots$ and $t = (t_0, t, \ldots)$ be a timed word over the alphabet $\Sigma = 2^{\mathsf{AP}}$, ϕ a* TPTL *formula, α a clock valuation, and $i \in \mathbb{N}$. The relation $(w, t, \alpha, i) \models_{\text{TPTL}} \phi$ is inductively defined as follows:*

$$
\begin{array}{ll}
(w,t,\alpha,i) \models_{\text{TPTL}} p & \textit{iff } p \in \sigma_i \\
(w,t,\alpha,i) \models_{\text{TPTL}} \pi_1 \approx \pi_2 & \textit{iff } \alpha(\pi_1) = \alpha(\pi_2) \\
(w,t,\alpha,i) \models_{\text{TPTL}} \pi_1 \prec \pi_2 & \textit{iff } \alpha(\pi_1) < \alpha(\pi_2) \\
(w,t,\alpha,i) \models_{\text{TPTL}} x.\phi & \textit{iff } (w,t,\alpha[x/t_i],i) \models_{\text{TPTL}} \phi \\
(w,t,\alpha,i) \models_{\text{TPTL}} (\neg\phi) & \textit{iff } (w,t,\alpha,i) \not\models_{\text{TPTL}} \phi \\
(w,t,\alpha,i) \models_{\text{TPTL}} (\phi_1 \wedge \phi_2) & \textit{iff } (w,t,\alpha,i) \models_{\text{TPTL}} \phi_1 \textit{ and } (w,t,\alpha,i) \models_{\text{TPTL}} \phi_2 \\
(w,t,\alpha,i) \models_{\text{TPTL}} (\bullet\, \phi_1) & \textit{iff } i > 0 \textit{ and } (w,t,\alpha,i-1) \models_{\text{TPTL}} \phi_1 \\
(w,t,\alpha,i) \models_{\text{TPTL}} (\bigcirc\, \phi_1) & \textit{iff } (w,t,\alpha,i+1) \models_{\text{TPTL}} \phi_1 \\
(w,t,\alpha,i) \models_{\text{TPTL}} (\phi_1\, \mathcal{S}\, \phi_2) & \textit{iff for some } j \leq i,\ (w,t,\alpha,j) \models_{\text{TPTL}} \phi_2, \\
 & \quad \textit{and } (w,t,\alpha,k) \models_{\text{TPTL}} \phi_1, \textit{ for all } k \in [j+1, i+1) \\
(w,t,\alpha,i) \models_{\text{TPTL}} (\phi_1\, \mathcal{U}\, \phi_2) & \textit{iff for some } j \geq i,\ (w,t,\alpha,j) \models_{\text{TPTL}} \phi_2, \\
 & \quad \textit{and } (w,t,\alpha,k) \models_{\text{TPTL}} \phi_1, \textit{ for all } k \in [i, j)
\end{array}
$$

Notation and Terminology. Analogous to the previous logics, the usual syntactic sugar and operator precedence rules can be defined.

Example 2.4.14. *Let* $\mathsf{AP} = \{p, q\}$ *be a set of propositions. As an alternative to the formula of Example 2.4.11, the property "it is always the case that every* p *is followed by a* q *within at most* 5 *time steps" can also be expressed by the* TPTL *formula*

$$\Box\, t_1.(p \rightarrow (\Diamond\, t_2.(q \wedge t_2 \leq t_1 + 5))).$$

Note that the freeze quantified variables t_1 *and* t_2 *are bound to the time stamps of the time points at which the enclosed subformula is satisfied.*

Remark 2.4.15 (Running example)**.** *Regarding the suitability of using* LTL, MTL, *or* TPTL *to formalize the example property of the running example, we make the following observations:*

- *Because there is no a priori bound on the number of transactions, customers, and reports processed by the* TPS *over time, they cannot be adequately represented using a finite set of propositions. Moreover, even if such a (large) bound existed, the propositional formula would have to deal with a large number of case distinctions, resulting in long, unintelligible, and error-prone property specification.*
- *While the need to express quantitative time constraints rules out* LTL*, the temporal operators of* MTL *and* TPTL *offer enough expressiveness to capture such time constraints.*
- *A last comment concerns the ability to refer to both past and future time points. Because it is not evident whether or how our example property could be specified with only temporal past or future operators, it is justified to require a logical fragment that supports both types of temporal operators.*

In the next section, we present a (first-order) temporal logic that is expressive enough to concisely formalize our example property.

2.4.3 Metric First-order Temporal Logic

We now introduce metric first-order temporal logic (MFOTL) [Cho95], which extends propositional metric temporal logic [Koy90, AH92] by providing predicates and quantification over an infinite domain of individuals.

While propositional temporal logics are often used in the verification community, first-order temporal logics are mainly studied in the context of a temporal database context. In this dissertation, we use metric first-order temporal logic for specifying properties of timed temporal structures. We later present a runtime monitoring approach for properties formulated within a safety fragment of MFOTL.

Syntax. Let $S = (\mathsf{C}, \mathsf{R}, a)$ be a signature and V denote a countably infinite set of variables, where we assume $\mathsf{V} \cap \mathsf{R} = \emptyset$ and $\mathsf{V} \cap \mathsf{C} = \emptyset$.

Definition 2.4.16. *The* (MFOTL) formulae *over* S *are inductively defined: (i) For* $t, t' \in \mathsf{V} \cup \mathsf{C}$*,* $(t \approx t')$ *and* $(t \prec t')$ *are formulae. (ii) For* $r \in \mathsf{R}$ *and* $t_1, \ldots, t_{a(r)} \in \mathsf{V} \cup \mathsf{C}$*,* $r(t_1, \ldots, t_{a(r)})$ *is a formula. (iii) For* $x \in \mathsf{V}$*, if* ϕ_1 *and* ϕ_2 *are formulae then* $(\neg\phi_1)$*,* $(\phi_1 \wedge \phi_2)$*, and* $(\exists x.\, \phi_1)$

are formulae. (iv) For $I \in \mathbb{I}$, if ϕ_1 and ϕ_2 are formulae then $(\bullet_I\, \phi_1)$, $(\bigcirc_I\, \phi_1)$, $(\phi_1\, \mathcal{S}_I\, \phi_2)$, and $(\phi_1\, \mathcal{U}_I\, \phi_2)$ are formulae.

Semantics. The semantics of MFOTL is defined with respect to timed temporal structures. A *valuation* is a mapping $v : \mathsf{V} \to |D|$. We abuse notation by applying a valuation v also to constant symbols $c \in \mathsf{C}$, with $v(c) = c^D$. For a valuation v, a variable vector $\bar{x} = (x_1, \ldots, x_n)$, and $\bar{d} = (d_1, \ldots, d_n) \in |D|^n$, $v[\bar{x}/\bar{d}]$ is the valuation that maps x_i to d_i, for i such that $1 \leq i \leq n$, and the valuation of the other variables is unaltered.

Definition 2.4.17. *Let (D, τ) be a timed temporal structure over S, with $D = (D_0, D_1, \ldots)$ and $\tau = (\tau_0, \tau_1, \ldots)$, ϕ an* MFOTL *formula over S, v a valuation, and $i \in \mathbb{N}$. We define the relation $(D, \tau, v, i) \models_{\text{MFOTL}} \phi$ as follows:*

$$
\begin{array}{ll}
(D,\tau,v,i) \models_{\text{MFOTL}} t \approx t' & \textit{iff } v(t) = v(t') \\
(D,\tau,v,i) \models_{\text{MFOTL}} t \prec t' & \textit{iff } v(t) < v(t') \\
(D,\tau,v,i) \models_{\text{MFOTL}} r(t_1, \ldots, t_{a(r)}) & \textit{iff } (v(t_1), \ldots, v(t_{a(r)})) \in r^{D_i} \\
(D,\tau,v,i) \models_{\text{MFOTL}} (\neg \phi_1) & \textit{iff } (D,\tau,v,i) \not\models_{\text{MFOTL}} \phi_1 \\
(D,\tau,v,i) \models_{\text{MFOTL}} (\phi_1 \wedge \phi_2) & \textit{iff } (D,\tau,v,i) \models_{\text{MFOTL}} \phi_1 \textit{ and } (D,\tau,v,i) \models_{\text{MFOTL}} \phi_2 \\
(D,\tau,v,i) \models_{\text{MFOTL}} (\exists x.\, \phi_1) & \textit{iff } (D,\tau,v[x/d],i) \models_{\text{MFOTL}} \phi_1, \textit{for some } d \in |D| \\
(D,\tau,v,i) \models_{\text{MFOTL}} (\bullet_I\, \phi_1) & \textit{iff } i > 0,\ \tau_i - \tau_{i-1} \in I, \textit{ and } (D,\tau,v,i-1) \models_{\text{MFOTL}} \phi_1 \\
(D,\tau,v,i) \models_{\text{MFOTL}} (\bigcirc_I\, \phi_1) & \textit{iff } \tau_{i+1} - \tau_i \in I \textit{ and } (D,\tau,v,i+1) \models_{\text{MFOTL}} \phi_1 \\
(D,\tau,v,i) \models_{\text{MFOTL}} (\phi_1\, \mathcal{S}_I\, \phi_2) & \textit{iff for some } j \leq i,\ \tau_i - \tau_j \in I,\ (D,\tau,v,j) \models_{\text{MFOTL}} \phi_2, \\
 & \quad \textit{and } (D,\tau,v,k) \models_{\text{MFOTL}} \phi_1, \textit{ for all } k \in [j+1, i+1) \\
(D,\tau,v,i) \models_{\text{MFOTL}} (\phi_1\, \mathcal{U}_I\, \phi_2) & \textit{iff for some } j \geq i,\ \tau_j - \tau_i \in I,\ (D,\tau,v,j) \models_{\text{MFOTL}} \phi_2, \\
 & \quad \textit{and } (D,\tau,v,k) \models_{\text{MFOTL}} \phi_1, \textit{ for all } k \in [i, j)
\end{array}
$$

Note that the temporal operators are augmented with lower and upper bounds. A temporal formula is only satisfied at some time point i if it is satisfied within the bounds given by the temporal operator, which are relative to the current time stamp τ_i.

Notation and Terminology. As syntactic sugar, the standard Boolean connectives and temporal operators are defined as for LTL and MTL. For convenience, we recall them here again: For example, we define $(\theta_1 \vee \theta_2) := (\neg((\neg\theta_1) \wedge (\neg\theta_2)))$ and $(\theta_1 \rightarrow \theta_2) := ((\neg\theta_1) \vee \theta_2)$. Moreover, we define the universal quantifier $(\forall x.\, \theta) := (\neg(\exists x.\, \neg\theta)$ and the temporal operators $(\blacklozenge_I\, \theta) := (\mathit{true}\, \mathcal{S}_I\, \theta)$, $(\blacksquare_I\, \theta) := (\neg(\blacklozenge_I(\neg\theta)))$, $(\lozenge_I\, \theta) := (\mathit{true}\, \mathcal{U}_I\, \theta)$, and $(\square_I\, \theta) := (\neg(\lozenge_I(\neg\theta)))$, where $I \in \mathbb{I}$ and *true* abbreviates $(\exists x.\, x \approx x)$. The non-metric variants of the temporal operators are easily defined, e.g., $(\square\, \theta) := (\square_{[0,\infty)}\, \theta)$. We use standard conventions concerning the binding strength of operators to omit parentheses. Recall that $\neg$ binds stronger than $\wedge$, which binds stronger than $\vee$, which in turn binds stronger than $\exists$. Moreover, temporal operators bind weaker than Boolean connectives and quantifiers.

Let $\bar{x} = (x_1, \ldots, x_n)$, where $x_i \in \mathsf{V}$ and $1 \leq i \leq n$, be a variable vector and let ϕ be a formula. We often write $\forall \bar{x}.\phi$ or $\forall x_1, \ldots, x_n.\phi$ instead of $\forall x_1.\ldots.\forall x_n.\phi$. Moreover, for a predicate T with $a(T) = n$, we define $\forall x_1, \ldots, x_n : T.\phi := \forall \bar{x}.T(x_1, \ldots, x_n) \rightarrow \phi$ and $\exists x_1, \ldots, x_n : T.\phi := \exists \bar{x}.T(x_1, \ldots, x_n) \wedge \phi$ (relativized quantification). For two vectors $\bar{t} = (t_1, \ldots, t_n)$ and $\bar{t}' = (t_1, \ldots, t_k)$ with $t_i, t_j \in \mathsf{V} \cup \mathsf{C}$ and $1 \leq i \leq n$ and $1 \leq j \leq k$, we define $\bar{t} \approx \bar{t}' := t_1 \approx t'_1 \wedge \ldots \wedge t_n \approx t_k$.

We call formulae of the form $t \approx t'$, $t \prec t'$, and $r(t_1, \ldots, t_{a(r)})$ *atomic*, and formulae with no temporal operators *first-order*. The outermost connective occurring in a formula θ is called the *main connective* of θ. A formula that has a temporal operator as its main connective is a *temporal* formula. An MFOTL formula θ is *bounded* if the interval I of every temporal operator $\mathcal{U}_I$ occurring in θ is finite. A formula with free variables is called *open*, whereas a formula without free variables is called *closed* or a *sentence*.

MFOTL denotes the set of MFOTL formulae and FOL the set of first-order formulae. Moreover, the set FOTL denotes the non-metric fragment of first-order temporal logic (FOTL), i.e., the MFOTL fragment that only uses non-metric temporal operators. Because the non-metric temporal operators are independent of the time stamps, the semantics of FOTL is normally defined with respect to temporal structures [Cho95].

Definition 2.4.18. *For $\theta \in$ MFOTL, we define its immediate temporal subformulae $tsub(\theta)$ as*

$$tsub(\theta) := \begin{cases} tsub(\alpha) & \text{if } \theta = \neg\alpha \text{ or } \theta = \exists x.\alpha, \\ tsub(\alpha) \cup tsub(\beta) & \text{if } \theta = \alpha \wedge \beta, \\ \{\theta\} & \text{if } \theta \text{ is a temporal formula,} \\ \emptyset & \text{otherwise.} \end{cases}$$

For example, for $\theta := (\bullet\, \alpha) \wedge ((\bigcirc \beta)\ \mathcal{S}_{[1,9)}\ \gamma)$, we have that $tsub(\theta) = \{\bullet\, \alpha, (\bigcirc \beta)\ \mathcal{S}_{[1,9)}\ \gamma\}$.

Note that the truth value of a closed formula does not depend on the valuation. If a formula $\theta \in$ MFOTL is closed, we thus simply write $(D, \tau, i) \models_{\text{MFOTL}} \theta$. The set of *models* $L(\theta)$ of a closed MFOTL formula θ is defined as

$$L(\theta) := \left\{ (D, \tau) \,\middle|\, (D, \tau, 0) \models_{\text{MFOTL}} \theta \right\}.$$

In other words, the set of models $L(\theta)$ thus represents the property induced by θ.

An open formula can also be seen as a *query*. For $\theta \in$ MFOTL with the free variables given by the vector $\bar{x} = (x_1, \ldots, x_n)$, we define the set of satisfying assignments at time instance i as

$$\theta^{(D,\tau,i)} := \left\{ \bar{d} \in |D|^n \,\middle|\, (D, \tau, v[\bar{x}/\bar{d}], i) \models_{\text{MFOTL}} \theta, \text{ for some valuation } v \right\}.$$

If the formula θ is in FOL, we write $(D_i, v) \models_{\text{FOL}} \theta$ instead of $(D, \tau, v, i) \models_{\text{MFOTL}} \theta$ and θ^{D_i} for $\theta^{(D,\tau,i)}$. Note that $(D_i, v) \models_{\text{FOL}} \theta$ agrees with the standard definition of satisfaction in first-order logic. For the rest of this dissertation and when clear from the context, we will simply write $\models$ instead of $\models_{\text{MFOTL}}$, etc.

Example 2.4.19. *Let $S = (\mathsf{C}, \mathsf{R}, a)$ be a signature with $\mathsf{R} = \{open, close\}$ and $a(open) = 1$ and $a(open) = 1$. The property "it is always the case that every element a satisfying predicate $open$ eventually satisfies predicate $close$" can be expressed by the* FOTL *formula*

$$\Box \forall x. open(x) \rightarrow (\Diamond\, close(x)).$$

With the ability of MFOTL to quantify over variables ranging over an infinite domain and the availability of metric temporal past and future operators, we can now also naturally express our example property from Section 1.3.

Example 2.4.20 (Running example)**.** *Let $S_{\text{TPS}} = (\mathsf{C}, \mathsf{R}, a)$ be the signature of the* TPS*, where $\mathsf{R} = \{trans, report\}$ and $a(trans) = a(report) = 2$. The example property from Section 1.3 can be expressed by the* MFOTL *formula*

$$\Box \forall t. \forall c. trans(t, c) \wedge \left(\blacklozenge_{[0,31)} \exists t'. trans(t', c) \wedge (\Diamond_{[0,3)}\, report(t', c)) \right) \rightarrow (\Diamond_{[0,3)}\, report(t, c)).$$

2.4.4 Two-sorted First-order Logic

Two-sorted first-order logic (2-FOL), also known as temporal relational calculus, is an expressive alternative to temporal logics for the specification of temporal properties [Tom03]. Instead of dedicated temporal operators, 2-FOL equips its predicate symbols with the capability to explicitly reference time points.

Syntax. The syntax of 2-FOL is defined as follows. Let $S = (\mathsf{C}, \mathsf{R}, a)$ be a signature. Let V denote a countably infinite set of (data) variables and T be a countably infinite set of temporal variables, disjunct from V, where we assume $(\mathsf{V} \cup \mathsf{T}) \cap \mathsf{R} = \emptyset$ and $(\mathsf{V} \cup \mathsf{T}) \cap \mathsf{C} = \emptyset$.

Definition 2.4.21. *The* 2-FOL formulae *over S are inductively defined: (i) For $t, t' \in \mathsf{T} \cup \mathsf{C}$, $(t \prec t')$ is a* 2-FOL *formula. (ii) For $x, x' \in \mathsf{V} \cup \mathsf{C}$, $(x \approx x')$ is a* 2-FOL *formula. (iii) For $r \in \mathsf{R}$, $x_1, \ldots, x_{a(r)-1} \in \mathsf{V} \cup \mathsf{C}$, and $t \in \mathsf{T} \cup \mathsf{C}$, $r(t, x_1, \ldots, x_{a(r)-1})$ is a* 2-FOL *formula. (iv) For $x \in \mathsf{V}$, if ϕ_1 and ϕ_2 are* 2-FOL *formulae then $(\neg\phi_1)$, $(\phi_1 \wedge \phi_2)$, and $(\exists x.\, \phi_1)$ are* 2-FOL *formulae. (v) For $t \in \mathsf{T}$, if ϕ_1 is a* 2-FOL *formula, then $(\exists t.\, \phi_1)$ is a* 2-FOL *formula.*

Semantics. We now give the semantics of 2-FOL with respect to temporal structures. Note that because the first position in each predicate symbol is used for explicitly referring time points, the semantics is given with respect to a temporal structure with an adapted signature. As an alternative, the semantics of 2-FOL can also be given with respect to first-order structures with temporal relations over the same signature such as abstract timestamp temporal databases (see Section 2.3.3).

Definition 2.4.22. *Let $D = (D_0, D_1, \dots)$ be a temporal structure over $S = (\mathsf{C}, \mathsf{R}, a)$, ϕ a* 2-FOL *formula over $S' = (\mathsf{C}', \mathsf{R}', a')$ with $\mathsf{C} = \mathsf{C}'$, $\mathsf{R} = \mathsf{R}'$, and $a(r) = a'(r) - 1$ for each $r \in \mathsf{R}$, and v a valuation. We define the relation $(D, v) \models_{\text{2-FOL}} \theta$ as follows:*

$$
\begin{array}{ll}
(D,v) \models_{\text{2-FOL}} t \prec t' & \textit{iff } v(t) < v(t') \\
(D,v) \models_{\text{2-FOL}} x \approx x' & \textit{iff } v(x) = v(x') \\
(D,v) \models_{\text{2-FOL}} r(t, x_1, \dots, x_{a(r)-1}) & \textit{iff } (v(x_1), \dots, v(x_{a(r)-1})) \in r^{D_{v(t)}} \\
(D,v) \models_{\text{2-FOL}} (\neg \phi_1) & \textit{iff } (D,v) \not\models_{\text{2-FOL}} \phi_1 \\
(D,v) \models_{\text{2-FOL}} (\phi_1 \wedge \phi_2) & \textit{iff } (D,v) \models_{\text{2-FOL}} \phi_1 \textit{ and } (D,v) \models_{\text{2-FOL}} \phi_2 \\
(D,v) \models_{\text{2-FOL}} (\exists x.\, \phi_1) & \textit{iff } (D, v[x/d]) \models_{\text{2-FOL}} \phi_1, \textit{ for some } d \in |D| \\
(D,v) \models_{\text{2-FOL}} (\exists t.\, \phi_1) & \textit{iff } (D, v[t/s]) \models_{\text{2-FOL}} \phi_1, \textit{ for some } s \in \mathbb{N}
\end{array}
$$

Notation and Terminology. Syntactic sugar and operator precedence rules can be defined in the standard way. For example, we can define $(t \preceq t') := (t \prec t' \vee t \approx t')$.

Let us now also consider how some example properties can be specified in 2-FOL.

Example 2.4.23. *Let $S = (\mathsf{C}, \mathsf{R}, a)$ with $\mathsf{R} = \{open, close\}$ and $a(open) = 2$ and $a(close) = 2$. As an alternative to the* FOTL *formula of Example 2.4.19, the property "it is always the case that every element a satisfying predicate $open$ eventually satisfies predicate $close$" can also be expressed by the following* 2-FOL *formula over S:*

$$\forall i. \forall x. open(i, x) \rightarrow (\exists j. i \preceq j \wedge close(j, x)).$$

While the above is only an example for how an MFOTL formula can be equivalently expressed in 2-FOL, note that all non-metric temporal operators of FOTL are definable by first-order formulae over a discrete and linearly ordered (time) domain. It is thus not difficult to see that all FOTL formulae can be equivalently expressed in 2-FOL [CT05]. As also the metric temporal operators of MFOTL are first-order definable, for each MFOTL formula there is an equivalent first-order formula, too. To represent MFOTL formulae, however, three-sorted first-order logic (3-FOL) is required. 3-FOL canonically extends 2-FOL but uses an additional sort for representing the time stamps. For example, for a given predicate $p(x)$, the MFOTL formula $\Diamond_{[c,d)}\, p(x)$ can be expressed by the equivalent 3-FOL formula $\exists i. \exists t. 0 \preceq i \wedge (\tau_0 + c \preceq t) \wedge (t \prec \tau_0 + d) \wedge p(i, t, x)$. Observe that each temporal relation now has two dedicated attributes, one storing the time point i and the other storing the time stamp τ (both ranging over the natural numbers).

As a result, the property of our running example can also be expressed using the following 3-FOL formula.

Example 2.4.24 (Running example)**.** *Let $S'_{\text{TPS}} = (\mathsf{C}, \mathsf{R}, a)$ be a (modified) signature for* TPS*, where $\mathsf{R} = \{trans, report\}$ and $a(trans) = a(report) = 4$. We can then express our example*

property as follows:

$$\forall i.\forall \tau_1.\forall t.\forall c.\mathit{trans}(i,\tau_1,t,c) \wedge \big(\exists j.\exists \tau_2.\, j \preceq i \wedge \tau_1 - \tau_2 \in [0,31) \wedge \exists t'.\, \mathit{trans}(j,\tau_2,t',c) \wedge \exists k.\exists \tau_3.\, j \preceq k \wedge \tau_3 - \tau_1 \in [0,3) \wedge \mathit{report}(k,\tau_3,t',c)\big) \rightarrow \exists l.\exists \tau_4.\, i \preceq l \wedge \tau_4 - \tau_1 \in [0,3) \wedge \mathit{report}(l,\tau_4,t,c),$$

where we write $\tau' - \tau \in [c,d)$ *for* $(\tau + c) \preceq \tau' \wedge (\tau' \prec \tau + d)$.

2.4.5 Non-logical Specification Languages

Apart from specification languages based on logic such as those presented above, there are also other types of specification languages suitable for the specification of temporal properties. In this section, we give a brief overview of some of them.

Automata

With properties being sets of finite and infinite sequences of elements (e.g., sets of words over some alphabet), automata are an alternative to logics for the specification of properties. In particular, if the set of possible system executions is definable over a finite alphabet, a property directly corresponds to the (ω-)regular language recognized by some finite state automaton. A (ω-)word, then, is an element of the property if and only if it is accepted by the respective automaton. For example, in the case of finite words, each regular language is recognized by some DFA. Similarly, for the case of infinite sequences, each ω-regular language is recognized by some Büchi automaton. Moreover, considering timed ω-words as the execution model, each timed ω-regular language is recognized by some timed automaton over the same alphabet.

An automata-based property specification approach that goes beyond finite state automata is offered by the theory of *security automata* [AS87, Sch00]. Security automata are a special class of Büchi automata that can handle infinite sets of states and input symbols. Instead of the classical Büchi acceptance condition, security automata use their own notion of acceptance. This allows for accepting both finite and infinite sequences of symbols and makes security automata recognize safety properties.

A general drawback of automata-based property specification is the following: If the set of automaton states is large or the transition function is complex, the specification of an automaton can be complex or even impractical. For these situations, [Sch00] proposes to encode the current state of a security automaton in multiple variables and to describe the transition function for the security automaton by guarded commands. Other authors proposed to use algebraic specification languages for the same purpose [BOS07]. Another alternative in the context of finite state automata is offered by regular expressions, which we describe next.

Regular Expressions

A further formalism for specifying properties of finite state systems, especially popular among programmers for its simplicity, is provided by regular expressions. Directly related to automata theory, regular expressions are built from a finite alphabet Σ and the operations union, concatenation, and Kleene star as follows: Every $a \in \Sigma$, the empty word ϵ, or the empty set $\emptyset$ is a regular expression. If E_1 and E_2 are regular expressions, then $(E_1 \cup E_2)$, $(E_1 \circ E_2)$, and (E_1^*) are also regular expressions. Any regular expression E induces a property by the language $L(E)$ that it describes. In fact, a language is regular if and only if some regular expression describes it [Sip96]. For example, for a given alphabet $\Sigma = \{a, b\}$, the regular expression a^*b defines the regular language $L(a^*b) = \{ab, aab, aaab, aaaab, \dots\}$ that contains all strings described by it.

Several extensions of regular expressions, resulting in either more compact representations or increased expressiveness, have been proposed. For example, extended regular expressions (i.e., regular expressions enriched with complementation), while equally expressive, can yield non-elementarily more compact expressions [SM73]. Furthermore, timed regular and timed ω-regular expressions, i.e., regular expressions with the ability to express quantitative timing constraints, express exactly timed ω-regular languages, thus being equivalent to timed automata in their expressive power [ACM02].

Regular expressions are often used to describe patterns of characters in text files. However, by interpreting the the symbols in Σ as observable events or states of some finite state system, regular expressions can be used to state arbitrary regular properties of such a systems. Moreover, being equivalent in descriptive power to finite state automata, specifying properties with regular expressions is often simpler than by defining a finite state automaton that recognizes the same language.

Process Algebras

Process algebras such as Communicating Sequential Processes (CSP) [Hoa85] or the Calculus of Communicating Systems [Mil82] are another class of popular formalisms for specifying properties of distributed and reactive systems. In a process algebra, a system is conceived as a set of concurrently executing processes. A process is built from other processes and symbols from some alphabet. Processes can be composed (sequentially, non-deterministically, recursively, or in parallel) and synchronized according to a set of algebraic operations. Let us consider a simple example using CSP notation (see [Ros97]). Let $\Sigma = \{a, b, c\}$ be an event alphabet. The process $P = a \rightarrow c \rightarrow STOP$ denotes the sequential composition of the events a and c. In other words, P first engages in the event a, then it engages in the event c, and stops execution afterwards. Similarly, the process $Q = b \rightarrow c \rightarrow STOP$ denotes the sequential composition of the events b and c. The process $R = P \,||\, Q$ then denotes the parallel composition of P and Q that behaves as the interleaving of P and Q and synchronizes them on event c.

The process R thus admits two traces over Σ, namely, bac and abc. In other words, the process R induces the property $L(R) = \{bac, abc\}$.

The process algebras mentioned above have been extended along many dimensions. While a detailed discussion is out of scope, we refer to [Bae05] for a general overview, to [MPW92a, MPW92b] for an introduction to the π-Calculus, to [Bae01] for timed process algebras in general, and to [RR88, SDJ+92] for Timed CSP in particular.

2.4.6 Discussion

Having presented a set of alternative formalisms for the specification of temporal property, we now briefly assess their relative strengths and weaknesses.

Let us first consider the presented logics-based means of specification. To start with, it is instructive to observe that LTL, MTL, and FOTL are essentially fragments of MFOTL. In particular, FOTL represents the MFOTL fragment with non-metric temporal operators only, whereas MTL represents the MFOTL fragment induced by propositional signatures, i.e., those signatures $S = (\mathsf{C}, \mathsf{R}, a)$, where $a(r) = 0$ for all $r \in \mathsf{R}$. Last and least expressive, LTL represents the MFOTL fragment induced by propositional signatures that only provides non-metric temporal operators. As a consequence, the semantics of these fragments are normally given with respect to ω-words, timed ω-words, or temporal structures, i.e., special cases of timed temporal structures as explained in Section 2.3.3. With the concept of freeze quantification as a means to specify quantitative time constraints, TPTL exists in its own right and cannot be treated as a fragment of MFOTL.

Considering their relative expressive power, the presented logics and some of their important fragments compare as follows. First of all, it is well-known that full LTL with both past and future operators and future-only LTL are equally expressive [Kam68, GPSS80]. In spite of this result, the use of past operators is advantageous because the use of full LTL can result in exponentially more succinct formulae than the use of future-only LTL [LMS02]. In contrast to the propositional case, the full FOTL fragment is more expressive than the future-only FOTL fragment and more expressive than past-only FOTL [AHdB95, AH99].

As LTL, MTL, and FOTL are essentially fragments of MFOTL, the differences in their relative expressive power are easy to see. While LTL is unable to express quantitative timing requirements, both LTL and MTL cannot deal with variables over infinite domains. Moreover, while FOTL can handle quantification, as opposed to MFOTL, it cannot cope with quantitative timing requirements either. Regarding the relative expressive power of TPTL and MTL, it is known that TPTL is strictly more expressive than MTL defined with both point-based and interval-based semantics [BCM05]. Moreover, because 2-FOL can express arbitrary temporal logical operators, 2-FOL can specify at least as many properties in temporal structures as FOTL. In fact, it can be shown that

2-FOL is, in general, strictly more expressive than FOTL [AHdB96, TN96]. Intuitively, the result is due to the fact that 2-FOL can reference multiple temporal contexts in a formula, whereas FOTL formulae always reference a single temporal context at a time. As we shall see in Chapters 4 and 5, however, in the context of monitoring this seeming limitation of temporal logic offers an advantage. Specifically, it makes possible the polynomially bounded space consumption of the presented monitoring algorithm.

Regarding the presented non-logical specification languages, we remark that the expressive power of finite state automata has been summarized in Section 2.2. Moreover, the relation between finite state automata and various types of regular expressions has already been pointed out. The following result characterizes the relationship between ω-regular languages recognized by Büchi automata and the properties definable in LTL: All properties definable in LTL over some alphabet Σ can be recognized by some Büchi automaton over Σ. The contrary does not hold though. The set of properties definable in LTL is a strict subset of the ω-regular languages. It is, however, well-known how LTL can be extended such that the properties definable in the resulting extended temporal logic are exactly the ω-regular languages [VW94]. While security automata specify safety properties, they do not directly support the expression of quantitative timing requirements. Instead, time must be tracked in a dedicated variable. Moreover, it remains unclear whether complex properties can be practically specified. Automata-based property specification of complex properties is not always intuitive, even if guarded command are used [BOS07].

Let us finally review the presented specification languages against the requirements imposed by the example property of our running example. As pointed out in Remark 2.4.15, specification languages based on finite alphabets are not expressive enough to sensibly handle our example property. Instead, we need support for variables ranging over infinite domains and their quantification. This disqualifies LTL, MTL, TPTL, regular expressions, finite state automata, and common process algebras from further consideration. Moreover, to formally capture our example property, a specification language must support quantitative time constraints and allow referring to both the past and the bounded future. This essentially also rules out security automata. As a result, MFOTL and 3-FOL finally remain as adequate formalisms to capture our example property. In this dissertation, we focus on MFOTL.

2.5 Finite Representations of Infinite Structures

In this section, we review infinite structures that admit finite representations. Our main focus lies on automatic structures, which we characterize in Section 2.5.2. Alternative approaches are briefly discussed in Section 2.5.3.

2.5.1 Requirements

For our purposes, a class $\mathcal{C}$ of finitely presentable infinite structures should at a minimum satisfy the following requirements:

1. *Finite representations:* Every structure $A \in \mathcal{C}$ is representable in a finite way.
2. *Effective semantics (for a relevant (first-order) logic $\mathcal{L}$):* Given any formula $\phi(\bar{x}) \in \mathcal{L}$ and a finite representation of a structure $A \in \mathcal{C}$, we can effectively construct a finite presentation of the set ϕ^A.

2.5.2 Automatic Structures

Automatic structures are possibly infinite relational structures whose domain and relations can be finitely represented by a collection of finite state automata over finite words of some alphabet [KN95, BG04]. Moreover, automatic structures admit effective semantics, which follows directly from the closure properties of regular languages.

In the following, we present essential definitions and results on automatic structures. For an overview of finite state automata and regular languages, see Section 2.2.1.

Definitions and Basic Results

To understand how k-ary relations can be represented using finite state automata, we first explain the notion of k-ary relations of words, for $k \geq 1$. To use ordinary finite state automata to represent such relations, we introduce the concept of a *convolution* of words. Intuitively, a convolution is an encoding of a tuple of words $\bar{w} \in \left(\Sigma^*\right)^k$ by a single word $w_1 \otimes \cdots \otimes w_k$ over the alphabet $\left(\left(\Sigma \cup \{\#\}\right)^k\right)^*$, where $\# \notin \Sigma$ and $\otimes$ is defined as follows.

Definition 2.5.1. *Let Σ be an alphabet and $\#$ a symbol not in Σ. The* convolution *of the words $w_1, \ldots, w_k \in \Sigma^*$ with $w_i = w_{i1} \cdots w_{i\ell_i}$ is the word*

$$w_1 \otimes \cdots \otimes w_k := \begin{bmatrix} w'_{11} \\ \vdots \\ w'_{k1} \end{bmatrix} \cdots \begin{bmatrix} w'_{1\ell} \\ \vdots \\ w'_{k\ell} \end{bmatrix} \in \left(\left(\Sigma \cup \{\#\}\right)^k\right)^*,$$

where $\ell = \max\{\ell_1, \ldots, \ell_k\}$ and $w'_{ij} = w_{ij}$, for $j \leq \ell_i$ and $w'_{ij} = \#$ otherwise.

The padding symbol $\#$ is added to the words w_i to ensure that all of them have the same length.

With the concept of a convolution at hand, we define a *regular relation* as follows.

Definition 2.5.2. *Let Σ be an alphabet and $\#$ a symbol not in Σ. A relation $\mathsf{R} \subseteq \left(\left(\Sigma \cup \{\#\}\right)^k\right)^*$ is* regular *if the language $\{w_1 \otimes \cdots \otimes w_k \mid (w_1, \ldots, w_k) \in \mathsf{R}\}$ is regular.*

We can now formally define automatic structures.

Definition 2.5.3. *A first-order structure A over a signature $S = (\mathsf{C}, \mathsf{R}, a)$ is* automatic *if there is a regular language $\mathcal{L}_{|A|} \subseteq \Sigma^*$ and a surjective function $\nu : \mathcal{L}_{|A|} \to |A|$ such that the language $\mathcal{L}_{\approx} := \{u \otimes v \mid u, v \in \mathcal{L}_{|A|} \text{ with } \nu(u) = \nu(v)\}$ is regular and, for each relation $r^A \subseteq |A|^{a(r)}$ with $r \in \mathsf{R}$, the language $\mathcal{L}_r := \{w_1 \otimes \cdots \otimes w_{a(r)} \mid w_1, \ldots, w_{a(r)} \in \mathcal{L}_{|D|} \text{ with } (\nu(w_1), \ldots, \nu(w_{a(r)})) \in r^A\}$ is regular.*

In other words, a first-order structure A over a signature $S = (\mathsf{C}, \mathsf{R}, a)$ and a domain $|A|$ is automatic if the domain $|A|$ and the relations r^A can be encoded as regular languages.

Each automatic structure admits a finite representation by means of a collection of finite state automata.

Definition 2.5.4. *An* automatic representation *of the automatic structure A consists of*

(i) the function $\nu : \mathcal{L}_{|A|} \to |A|$,

(ii) a family of words $(w_c)_{c \in \mathsf{C}}$ with $w_c \in \mathcal{L}_{|A|}$ and $\nu(w_c) = c^A$, for all $c \in \mathsf{C}$, and

(iii) a collection $(\mathcal{A}_{|A|}, \mathcal{A}_{\approx}, (\mathcal{A}_r)_{r \in \mathsf{R}})$ of automata that recognize the languages $\mathcal{L}_{|A|}$, $\mathcal{L}_{\approx}$, and $\mathcal{L}_r$, for all $r \in \mathsf{R}$.

Note that every automatic structure admits an automatic representation over the binary alphabet $\Sigma = \{0, 1\}$ [Blu99]. Moreover, every automatic structure has an automatic representation in which the function ν is injective [KN95].

The following theorem makes clear why automatic structures are important: the first-order theory in automatic structures is decidable.

Theorem 2.5.5 ([KN95])**.** *Let A be an automatic structure. There exists an algorithm that, given a first-order formula ϕ with free variables $(x_1, \ldots, x_n)$, builds an automaton that recognizes the relation ϕ^A.*

The correctness of this theorem follows immediately from the closure properties of regular languages.

Example 2.5.6 lists some examples of automatic structures.

Example 2.5.6. *The following are prominent examples of automatic structures. All finite structures such as, e.g., relational databases, are automatic. Morover, Presburger arithmetic, i.e., the first-order theory with addition and the ordering relation over the natural numbers, is automatic.*

Throughout this dissertation, we assume that for an automatic structure, we always have an automatic representation for it at hand.

Basic Arithmetic in Automatic Structures

We now show that we can define different basic arithmetic relations in the first-order logic over an automatic structure D. In the following, we assume that $|D| = \mathbb{N}$ and

that $<$ is the standard ordering on $\mathbb{N}$. This is without loss of generality whenever the function ν is injective, i.e., every element in $|D|$ has only one representative in $\mathcal{L}_{|D|}$.

We start by showing that the elements in $|D|$ can be linearly ordered by a regular relation. Let $L \subseteq \Sigma^*$ be the regular language that represents the domain $|D|$, where Σ is some finite alphabet. Without loss of generality, we assume a linear order $\prec_{\text{alph}}$ on Σ. We lift $\prec_{\text{alph}}$ to linearly order the elements in Σ^*. For $w, w' \in \Sigma^*$, we define $w \prec_* w'$ iff $|w| < |w'|$, or $|w| = |w'|$ and $w \prec_{\text{lex}} w'$, where $|u|$ denotes the length of a word $u \in \Sigma^*$ and $\prec_{\text{lex}}$ is the lexicographical ordering on Σ^* with respect to the ordering $\prec_{\text{alph}}$ on the alphabet Σ.

It is easy to see that $\prec_*$ can be recognized by an automaton reading the convolution of the letters of w and w', i.e., the language $O := \{w \otimes w' \mid w \prec_* w'\}$ is regular. $O \cap L$ is a regular language, which we can use to order the elements in $|D|$. For $a, b \in |D|$, we define $a <_* b$ iff $u \prec_* v$, where u and v represent a and b, respectively. Recall that we assume that the domain representation is injective, i.e., every element in $|D|$ has a unique representative in L.

Obviously, the ordering $<_*$ is regular and $(|D|, <_*)$ is isomorphic to $(\mathbb{N}, <)$. In the following, we assume that $|D| = \mathbb{N}$ and $<_* = <$.

Let us now consider some concrete relations. To begin with, note that the successor relation $\text{succ}(x, y) := \{(x, y) \in \mathbb{N}^2 \mid y = x + 1\}$ is regular, since the formula $x \prec y \wedge \neg\exists z.\, x \prec z \wedge z \prec y$ defines it. From this, it is easy to see that also $\{(x, y) \in \mathbb{N}^2 \mid x + d \leq y\}$ is regular, for any $d \in \mathbb{N}$. Moreover, for $d \in \mathbb{N}$, the formula

$$\exists z_0. \ldots \exists z_d.\, x \approx z_0 \wedge y \approx z_d \wedge \bigwedge_{0 \leq i < d} \text{succ}(z_i, z_{i+1})$$

defines the relation $\{(x, y) \in \mathbb{N}^2 \mid x + d = y\}$. From these relations, it follows that the relations $\{(x, y) \in \mathbb{N}^2 \mid x - y = d\}$, $\{(x, y) \in \mathbb{N}^2 \mid x - y \leq d\}$, and $\{(x, y) \in \mathbb{N}^2 \mid |x - y| \leq d\}$ are all regular.

2.5.3 Other Finitely Representable Structures

While automatic structures represent an important class of finitely representable structures, other classes of structures admit finite representations, too. In this section, we briefly review some of these alternatives in the light of the requirements postulated in Section 2.5.1. In doing so, we also mention alternatives to the use of automata as finite representations of structures. See [BG04] for more details.

Generalizations of automatic structures. Generalizing the concept of automatic structures, ω-automatic structures or (ω-)tree-automatic structures use ω-regular languages and ω-automata or finite automata on finite or infinite trees, respectively, as finite representations of possibly infinite structures. In contrast to automatic structures, the use of

ω-automatic structures allows for finitely representing structures that have an uncountable cardinality such as $\mathbb{R}$. In spite of their increased expressive power, ω-automatic structures still admit effective evaluation of all first-order formulae. Moreover, first-order formulae can be effectively evaluated also over (ω-)tree-automatic structures.

Constraint databases. Constraint databases extend relational databases by infinite relations that are finitely represented by quantifier-free formulae (i.e., constraints). In order to enable effective evaluation of first-order formulae, the constraints must be expressed over a fixed background structure that admits effective quantifier elimination. The evaluation can be implemented through standard algorithms from real algebraic geometry. See, for example, [KKR95, KPL00] for overviews.

Recursive structures. Recursive structures are countable first-order structures whose functions and relations are computable and can therefore be represented in a finite manner. Recursive structures have been studied intensively, in particular, also with respect to issues of finite model theory on recursive databases. Because recursive structures generally only admit effective evaluation algorithms for quantifier-free formulae, however, they do not provide effective semantics for first-order logic. See, for example, [EGNR98] for an overview.

"You can observe a lot by just watching."

Lawrence Peter "Yogi" Berra

Chapter 3

Runtime Monitoring

THE need to monitor reactive systems at execution time occurs in many domains and for many different purposes. For example, monitoring capabilities have been implemented and used in program debuggers, profilers, test tools, or databases for purposes such as software fault detection, diagnosis, or dynamic integrity checking [PN81, DGa04, CR06, Cho95]. Moreover, the ability to check whether an observed system behavior violates a given property is used for compliance or business activity monitoring [GMP06, DJLS08], the enforcement of history-based access or usage control policies [AF03, HPB+07], or the detection of dynamic separation of duty violations [San88].

In principle, monitoring functionality may be implemented in two different ways. In the first approach, a software developer manually translates requirements into procedural program code. Both the correctness and the efficiency of the implemented monitor then largely depend on the sophistication of the developer. In the second approach, known as *runtime monitoring*[1], the properties to be monitored are formally expressed using a declarative specification language such as LTL. A correct and efficient runtime monitor is then automatically synthesized from the formal property specification. Finally, the generated monitor is executed in parallel with the target system and checks whether the observed system behavior satisfies or violates the property. With its focus on the automated generation of monitoring functionality from a declarative specification, runtime monitoring offers a systematic and increasingly popular alternative to the manual and ad-hoc implementation of monitoring capability [Vis00, DGa04, Run08, LS08].

The purpose of this chapter is threefold. First, we give a gentle introduction to the field of runtime monitoring and sketch some of the core challenges involved. Second, we provide a brief but systematic overview of existing work, which integrates several fields of hitherto separate research. While a comprehensive survey is out of scope, we position existing work in the context of a simple typology. Third, we set the context and provide a reference point for our original contributions presented in later chapters.

[1] Synonymous names are runtime verification, execution monitoring, or dynamic analysis.

The structure of the chapter is as follows. In Section 3.1, we first define the concept of a runtime monitor and review some basic aspects of runtime monitoring. In Section 3.2, we then provide an overview and a classification of different runtime monitoring approaches. Finally, in Section 3.3, we discuss different types of applications of runtime monitoring. The chapter concludes with a summary in Section 3.4.

3.1 Definitions and Basic Concepts

As sketched in Section 1.1, runtime monitoring is the problem of checking whether or not a given property is satisfied by an observed system execution. In the following, we make this more precise and summarize the main challenges of runtime monitoring.

For the rest of this section, let S be a set of states or events.

3.1.1 Runtime Monitor

Before giving a more formal definition of a runtime monitor, recall from Section 2.4.1 that a property is a set of infinite executions. Properties are defined by means of a property specification language such as LTL. The behavior of a monitored system is represented using an appropriate model of system execution, for example, an ω-word. The model of the system that generates the observed behavior is not known, that is, the monitored system is viewed as a black box. Moreover, note that at each time point a runtime monitor can only observe a finite prefix of a possibly infinite execution of the monitored system. As a result, the monitor must yield its verdict based on only a finite prefix instead of the entire execution.

Definition 3.1.1. *Let $\sigma \in \mathsf{S}^\omega$ be an execution and $\Gamma \subseteq \mathsf{S}^\omega$ be a property. A (runtime) monitor is an algorithm that processes a finite prefix of σ and checks whether $\sigma \in \Gamma$ or $\sigma \notin \Gamma$.*

Observe that Definition 3.1.1 does not constrain how a runtime monitor processes an observed prefix. Specifically, it does not preclude algorithms that process an entire prefix at once to decide whether it belongs to an execution that satisfies or violates a given property. More often than not, however, runtime monitors are online algorithms that work in an incremental fashion. This means that they process a finite but continuously growing prefix of some infinite execution. Results computed at previous time points are reused and updated with newly observed information. Ideally, at each time point only information that is relevant for the current and future evaluations of the property under consideration is preserved.

3.1.2 Evaluating Finite Prefixes of Executions

While a runtime monitor only observes finite prefixes of executions, the semantics of property specification languages is usually defined with respect to infinite sequences

(see Section 2.4). A runtime monitor thus often cannot decide whether the observed prefix belongs to a system execution that satisfies the property according to the usual infinite semantics. More specifically, let $\Gamma \subseteq \mathsf{S}^\omega$ be a property and let $i \in \mathbb{N}$ denote a time point. Having observed the prefix $\sigma[0,i]$ of an execution $\sigma \in \mathsf{S}^\omega$, a runtime monitor is in one of the following situations [PZ06, BLS06, MNP08]:

1. *Satisfaction:* for all $\sigma' \in \mathsf{S}^\omega$, we have that $\sigma[0,i]\sigma' \in \Gamma$,
2. *Violation:* for all $\sigma' \in \mathsf{S}^\omega$, we have that $\sigma[0,i]\sigma' \notin \Gamma$, or
3. *Inconclusive:* for some $\sigma', \sigma'' \in \mathsf{S}^\omega$, we have that $\sigma[0,i]\sigma' \in \Gamma$ and $\sigma[0,i]\sigma'' \notin \Gamma$.

For an observed prefix $\sigma[0,i]$, a property is *satisfied* (or positively determined) if all possible extensions of the prefix satisfy the property (i.e., the observed prefix is good). Likewise, a property is *violated* (or negatively determined) if all possible extensions of $\sigma[0,i]$ violate the property (i.e., the observed prefix is bad). If the observed prefix can be extended such that some extensions satisfy and others violate the property, it is *inconclusive* whether the property is satisfied or violated.

Example 3.1.2. *We illustrate these situations by considering some* LTL *properties. Let w be an ω-word over $\Sigma = 2^{\mathsf{AP}}$ with $\mathsf{AP} = \{p, q\}$ and $\bar{w} = w_0 w_1 \dots w_i$ be a finite prefix of w. Let us consider the following situations and formulae:*

1. *Let $\phi = \Diamond\, p$ and assume that $p \in w_j$ for some $j \leq i$. Then, the property defined by ϕ is positively determined by $\bar{w}$.*
2. *Let $\phi = \Box\, p$ and assume that $p \notin w_j$ for some $j \leq i$. Then, the property induced by ϕ is negatively determined by $\bar{w}$.*
3. *Let $\phi = \Box\, p$ and assume that $p \in w_j$ for all j with $0 \leq j \leq i$. In this situation, the prefix $\bar{w}$ can be extended such that ϕ is either satisfied or violated over w. As a result, for the observed prefix $\bar{w}$ it is inconclusive whether or not ϕ is satisfied by w.*

Several solutions have been proposed to resolve the mismatch between the finite observed prefixes and the infinite semantics of property specification languages. The proposed solutions roughly fall into two categories: (1) the definition of finite semantics to prevent inconclusive situations from occurring or (2) the introduction of additional truth values to explicitly handle these situations. We briefly discuss them below.

Finite Semantics

The first approach is to equip property specification languages with a semantics on finite prefixes. This ensures that an unambiguous Boolean verdict can be associated with each observed prefix. In the context of (future-only) LTL, for example, finite semantics have been proposed by [LPZ85, GH01, EFH$^+$03, RH05]. Under the so-called weak view semantics, inconclusive prefixes are evaluated to $true$, while under the strong view semantics, they are interpreted as $false$ [EFH$^+$03]. The weak view reflects the intuition

that an observed prefix satisfies a property Π as long as it provides no evidence against Π. In contrast, the strong view formalizes the intuition that a prefix satisfies a property Π if it provides all the required evidence for Π. For the sake of illustration, consider again the third formula in Example 3.1.2. Because p holds at each time point in the observed prefix, the formula $\Box p$ is true under the weak view semantics and false under the strong view semantics.

A similar solution is adopted by Chomicki in the context of MFOTL and (finite) database histories [Cho92a, Cho92b, Cho95]. Concretely, only formulae of the form $\bar{\Box}\,\phi$ are used for expressing properties, where $\bar{\Box}$ is a finite version of the usual always operator ($\Box$) and ϕ is a past-only MFOTL formula. Intuitively, the operator $\bar{\Box}$ only requires that ϕ holds with respect to all time points in the observed prefix. Because ϕ can only include temporal past operators, the resulting monitor can always determine whether an observed prefix satisfies or violates the expressed property. Note that all properties expressible are sets of finite database histories. Moreover, observe that the use of $\bar{\Box}$ essentially corresponds to a weak view semantics as described above.

Additional Truth Values

The second approach is to introduce additional "truth" values to explicitly represent inconclusive situations. This is the solution proposed by [RHKR01, ABLS05, dR05, BLS06, BLS07b, BLS07a] in the contexts of LTL and a real-time variant of LTL. In the simplest case, a dedicated value $\mathit{inconclusive}$ is added to the truth domain. In addition to the usual Boolean verdicts true and false, a verdict may then also be $\mathit{inconclusive}$. An advantage of using such an additional truth value is that, in contrast to the strong or weak view semantics mentioned above, a violation or satisfaction must not be reported prematurely. Let us consider again Example 3.1.2. Having observed the prefix $\bar{w}$, a runtime monitor with a three-valued semantics would explicitly flag the formula $\Box p$ as $\mathit{inconclusive}$. This is in contrast to using the weak or strong view semantics that interpret this formula as true or false, respectively.

3.1.3 On (Non-)Monitorable Properties

Let $\Pi \subseteq S^\omega$ be a property induced by an (effectively computable) predicate $\hat{P}$ defined over individual executions. It is well-known that Π can be verified by some runtime monitor if it is a safety or a co-safety property [Vis00]. The reason why safety and co-safety properties are natural candidates for verification using runtime monitoring is that they have (finite) bad and good prefixes, respectively. This ensures that violations or satisfactions can always be determined by observing a finite prefix of an execution. As shown in [BLS07b] for three-valued LTL, however, the class of monitorable properties is strictly larger than the union of safety and co-safety properties. For example, consider the future-only LTL formula $\phi = ((p \vee q)\,\mathcal{U}\, r) \vee (\Box p)$ over the

alphabet $2^{\{p,q,r\}}$. The property induced by ϕ is monitorable but it is neither a safety nor a co-safety property. This can be seen as follows: The ω-word $w_p = \{p\}\{p\}\{p\}\cdots$ satisfies ϕ. Because none of its prefixes are good, however, ϕ does not describe a co-safety property. Furthermore, the ω-word $w_q = \{q\}\{q\}\{q\}\cdots$ does not satisfy ϕ. However, none of its prefixes are bad and thus ϕ does not describe a safety property either. In spite of this, any finite prefix that is neither bad or good can be extended to a good or bad prefix, respectively. This follows from the following observations: The word $w_{good} = w_0 w_1 \cdots w_i\{r\}$ is a good prefix for ϕ if $p \in w_j$ or $q \in w_j$ for all $j \leq i$. Moreover, the word $w_{bad} = w'_0 w'_1 \cdots w'_i$ with $w'_i = \emptyset$ is a bad prefix for ϕ. Consequently, any symbol containing r results in a good prefix, while any finite suffix that includes the symbol $\emptyset$ makes the prefix bad.

Not all types of properties can be verified by runtime monitoring though. As an example, consider the liveness property defined by the LTL formula $\phi = \Box \Diamond p$. Intuitively, this formula states that there should be infinitely many time points that satisfy p. By observing only a finite prefix of an infinite system execution, however, a runtime monitor can never determine whether the target system satisfies or violates this property. The reason is that every observed prefix can be extended to an infinite execution that satisfies ϕ as well as an infinite execution that violates ϕ. Hence, no observed prefix can be finitely extended such that ϕ is positively or negatively determined [PZ06].

More formally, the class of non-monitorable properties can be characterized using the notion of an *ugly* prefix. Let $\Pi \subseteq \mathsf{S}^\omega$ be a property induced by a formula ψ. A prefix $u \in \mathsf{S}^*$ is called ugly for Π if there is no $v \in \mathsf{S}^*$ such that uv is either a good or a bad prefix for Π. For an observed prefix $u \in \mathsf{S}^*$, the property Π (or the formula ψ that defines Π) is called *non-monitorable* after u if u is ugly. Correspondingly, the property Π is monitorable if it has no ugly prefix and is effectively computable [Vis00, PZ06, BLS07a].

3.2 An Overview and Typology of Runtime Monitoring

We now present an overview and classification of existing runtime monitoring approaches by using a simple typology. Our typology is based on three main dimensions, depicted in Figure 3.1. The first dimension concerns the *execution model* upon which a runtime monitoring approach is based. The second dimension captures the *specification language* used for defining the properties that shall be monitored. The third dimension reflects the actual mathematical technique used to realize the respective monitor from a given property specification.

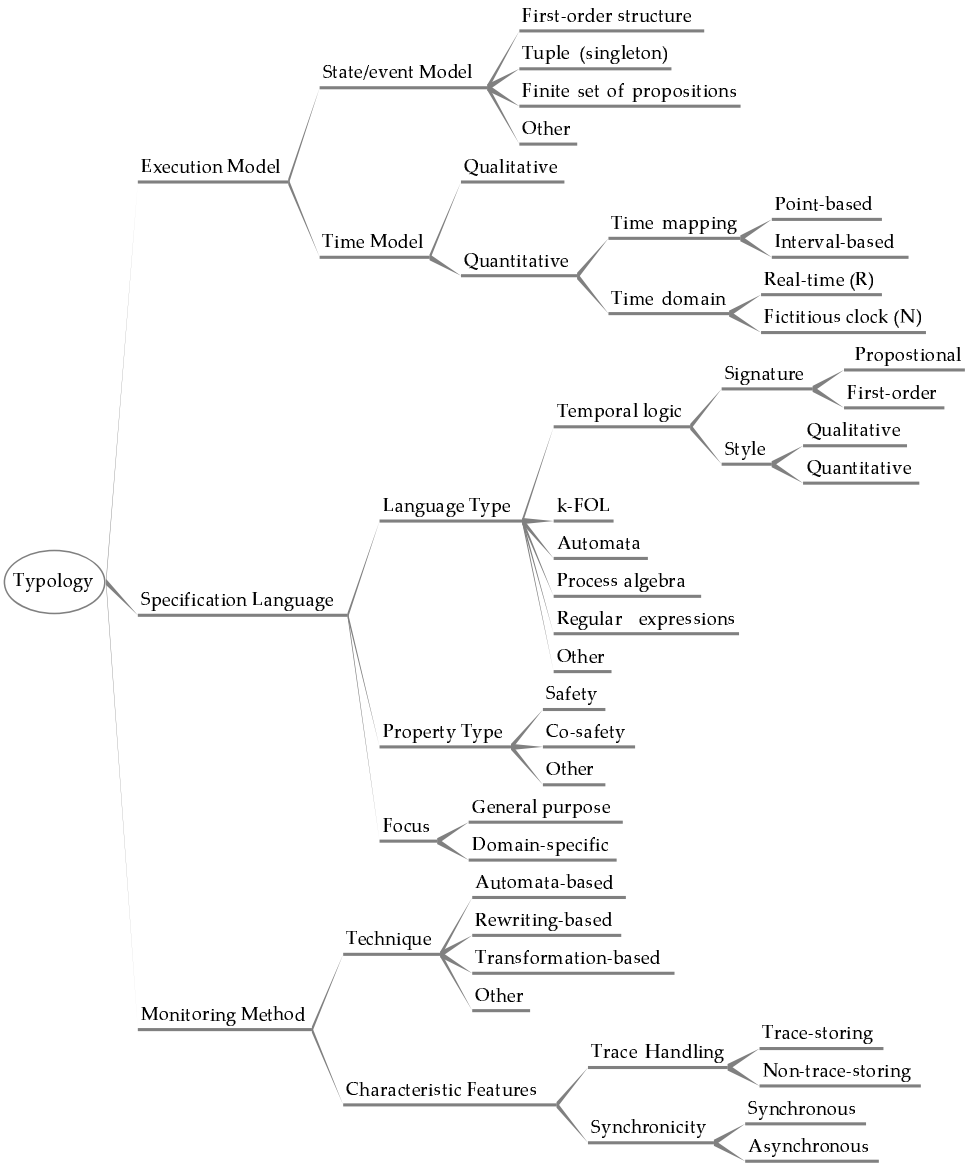


Figure 3.1: A runtime monitoring typology.

3.2.1 Execution Model

Existing runtime monitoring approaches can be distinguished according to the formal model used to represent system executions. As explained in Section 2.3, executions of reactive systems are typically represented as timed or untimed sequences of states or events. These sequences consist of a specific state/event model and an associated time model.

State/event Model

The following models are commonly used to represent individual states or events in system executions: *first-order structures* (e.g., [Kun84, LS87, Cho95, SW95, BKMP08a]), *singleton relations* (e.g., [RGL01, GOA05, DSS+05, SSLK06, ABW06, GLdB07]), or *finite*

sets of (interpretations of) propositions (e.g., [GD00, Vis00, GH01, KPA03, HR04a, TR05, dR05, BLS06, MNP06]). Some runtime monitoring approaches also rely on models *other* than the above (e.g., [BGHS04b, BRH07, Sto07, HV08]). Recall from Section 2.3.3 that singleton tuples and finite sets of propositions are special cases of first-order structures.

Time Model

We distinguish between approaches that use a *qualitative* (e.g., [Kun84, LS87, RGL01, HR01b, GH01, HR04a, Sto07, HV08]) and those using a *quantitative* time model. Among those that rely on quantitative time models, we further distinguish between those that are *point-based*, i.e., a time stamp is associated with each state or event, (e.g., [Cho95, TR05, BLS06, Dru06, AK07]) and those that are *interval-based*, i.e., each state is associated with an interval from the respective time domain, (e.g., [GD00, HJL03, MNP06]). Typically, $\mathbb{R}$ *(real-time)* (e.g., [GD00, MNP06]) or $\mathbb{N}$ *(fictitious clock)* (e.g., [Cho95, TR05, BLS06, BKMP08a]) are considered as the time domain.

Note that a choice of a particular state/event model and a time model gives rise to a unique execution model. For example, a tuple-based state/event model and a quantitative time model with fictitious clock semantics gives rise to a data stream as defined in Section 2.3.3.

In the database community, monitoring approaches naturally use database histories or (timed) temporal databases as their execution models (e.g., [Kun84, LS87, Cho95, SW95]). Similarly, the canonical execution model of the data stream processing community are data streams (e.g., [ABW06, GLdB07]). Surprisingly, most work in the area of software program monitoring focussed on the use of propositional executions models (e.g., [GD00, GH01, KPA03, HR04a, TR05, dR05, BLS06, MNP06]). Recent exceptions are [DSS+05, BGHS04b, SSLK06, Sto07].

3.2.2 Specification Language

Naturally, existing runtime monitoring approaches can also be classified in terms of their *specification language*. Recall that the semantics of the specification language must be defined with respect to the selected execution model. See Section 2.4 for formal definitions.

Language Type

In existing runtime monitoring approaches, properties are specified using fragments of *temporal logics*, *(k-sorted) first-order logic* (e.g, [Tom03, AK07]), *automata* (e.g., [Sch00, dR05, LBW05, HLM+08]), *regular expressions* (e.g., [SR03, CR03, BGHS04b, vL06]), or *process algebra* (e.g, [BFMW01]). Temporal logic-based specification languages are further differentiated as being either *propositional* (e.g., [GD00, GH01, KPA03, HR04a,

BGHS04b, TR05, dR05, BLS06, MNP06]) or *first-order* (e.g., [Kun84, LS87, Cho95, SW95, BKMP08b] and as either *qualitative* (e.g, [Kun84, LS87, RGL01, HR01b, GH01, HR04a, Sto07, HV08]) or *quantitative* (e.g, [Cho95, GD00, HJL03, BGHS04b, BGHS04b, TR05, MNP06, BLS06, Dru06, AK07]), respectively. Moreover, some approaches to runtime monitoring use *other* means of specification such as functional, object-oriented, or other imperative programming languages (e.g, [BGHS04b, DSS+05, GLdB07]). Some monitoring approaches also support several specification languages (e.g., [BGHS04b]).

Property Type

We can further classify runtime monitoring approaches with respect to the properties that can be expressed and monitored. The runtime monitors mentioned so far can express and detect violations of *safety properties*. Normally, this at least includes *invariants* and, if a quantitative specification language is used, possibly also *time-bounded liveness properties* (recall that these are safety properties, too). In case the specification language supports negation, often the same approaches can also express and detect satisfaction of *co-safety properties*. Moreover, some approaches can even handle *other temporal properties* that are neither safety nor co-safety properties (e.g., [BLS07b]).

Focus

The focus of a specification language is said to be *domain-specific* if it provides specific features or operations for a particular domain (e.g., [GOA05]). Otherwise, the approach is *general purpose* (e.g., [BGHS04a, BRH07, BKMP08a]. Because of their declarative nature logic-based specification languages are typically considered general purpose.

Not surprisingly, the consideration of more expressive property specification languages or executions models usually results in more complex runtime monitoring problems. Moreover, the choice of a property specification language also has an immediate impact on the properties that can be specified. For example, time-bounded liveness properties obviously can only be expressed if a quantitative specification language and a respective execution model has been chosen. Furthermore, while every safety property expressible by an LTL formula ϕ can be equivalently expressed by an LTL formula of the form $\Box\,\psi$, where ψ is a past-only LTL formula [LPZ85], this does not hold in the case of FOTL. As shown by [CN95], there are safety properties that cannot be expressed by formulae of this form.

3.2.3 Monitoring Method

Finally, we can classify existing runtime monitoring approaches by their specific monitoring method, particularly, by the actual technique used to realize the required mon-

itoring functionality. In addition, they can be also be distinguished in terms of specific features that the monitoring method provides.

Technique

The majority of runtime monitoring methods rely on *automata-based*, *rewriting-based*, *transformation-based*, or *other* techniques for automatically realizing the required monitoring functionality. Let us briefly discuss each of them.

Automata-based Techniques. As the name suggests, automata-based runtime monitors involve the construction of one or more finite state automata from a given property specification. The usual approach leverages the close connection between languages recognized by finite state automata and properties defined by formulae in propositional temporal logics. For example, LTL formulae can be translated into runtime monitors based on Büchi automata by adapting standard LTL-to-Büchi automata construction techniques [VW86, VW94, GH01]. Many other monitoring methods for properties expressed in qualitative or quantitative propositional temporal logics, regular expressions, or process algebras use similar techniques (e.g., [TOOD96, GPVW96, Gei01, SR03, FS04, dR05, BLS06, MNP06]).

While the automata-based techniques work well for propositional temporal logics, they do not easily carry over to first-order temporal logic. The problem is mainly due to the undecidability of the potential satisfiability problem for FOTL over infinite temporal databases [Cho95]. A further difficulty is caused by first-order quantification of variables over infinite domains, resulting in a possibly infinite number of automaton states. To address the first problem, a weaker notion of satisfiability based on finite database histories can be used [LZ91]. A solution that addresses the second problem is to only admit property specifications expressed in the so-called biquantified (future-only) FOTL fragment [LS87, CN95], where temporal operators must not occur in the scope of data quantifiers (with the exception of the implicit universal quantification of the free variables outside temporal operators). For such formulae, the standard propositional construction of a finite state automaton can be used [Wol83, LS87, LF89].

Rewriting-based Techniques. In contrast to automata-based approaches, where the runtime monitor is usually synthesized from some property specification before the actual monitoring starts, *rewriting-based* techniques directly operate on sequences of system states or events (e.g., [HR01b, HR01c, RV03, BGHS04b, RH05]). Specifically, they process each new state or event as soon as it becomes available and update the monitored property specification according to predefined rules until its truth value can be decided. For example, the rewriting-based monitoring algorithm given in [HR01b] transforms an LTL formula ϕ upon arrival of each new event e to a formula ϕ_e. The transformed formula ϕ_e then represents a new formula that must be satisfied by the

remaining suffix of the execution trace. As the monitoring algorithm consumes new events, the formula ϕ evolves into other LTL formulae until a final truth value can be determined.

Transformation-based Techniques. Another approach to realizing a runtime monitor is based on a *transformation* of the original property specification into an equivalent representation, which is better suited for incremental and space-efficient monitoring than the original representation (e.g., [Cho95, AK07, BKMP08a]). For example, in [AK07] property specifications expressed in bounded temporal logic are translated into monadic first-order logic over difference inequalities. At each step of the monitored timed trace, the monadic first-order logical formula is modified by taking into account new state and time information. The resulting formula is then checked for being a tautology or unsatisfiable by a decision procedure for monadic difference logic. In Chapter 4, we present a monitoring approach that is based on a similar idea. In particular, we transform bounded MFOTL formulae into first-order logical formulae over an extended signature and incrementally construct an extended first-order structure such that the transformed formulae can be evaluated at each time point.

Other Techniques. Several other techniques for the construction of runtime monitors have been proposed in the literature. In the stream-based approach, data streams are semi-imperatively 'programmed' by the iterative application of so-called data stream operators to form new data streams (e.g., [BW01, DSS+05, ABW06, GLdB07]). In the query-based approach (e.g., [KMSF01, GOA05]), system states or events are recorded in a database and analyzed, for example, using Structured Query Language (SQL) queries, for each observed prefix or after the monitored system terminates. In pattern-based approaches, typical properties are formalized by using parameterizable specification patterns (e.g., [GMP06]). Similarly, algorithm-based techniques, for example, for the detection of data races (e.g., [SBN+97, AHB03]) or deadlock detection (e.g., [BH06, AS06]) realize required monitoring functionality by dedicated algorithms.

Characteristic Features

Runtime monitoring methods can be further distinguished by their characteristic features [Cho95, RH05]. In the following, we focus on two specific features, namely how a runtime monitor handles the observed trace as well as its ability to recognize the violation or satisfaction of a property as quickly as possible.

Trace Handling. Runtime monitoring methods can be distinguished in terms of their requirements with respect to the observed prefix of some system execution. A *trace-storing* monitor requires that the entire observed prefix be preserved (e.g., [KMSF01, HR01c, GOA05]). On the positive side, this allows for analyzing the observed prefix

also with respect to previously unknown properties. On the negative side, the space consumption of trace-storing monitors is linear in the length of the observed prefix. Trace-storing monitors are suitable for *offline* monitoring scenarios, where they process observed prefixes of executions that were generated and recorded at some earlier time point. For example, algorithms for the analysis of log files usually belong to this category. In contrast, a *non-trace-storing* monitor does not need to store the entire observed prefix of a system execution. Instead, it only stores an encoding of the processed prefix, namely the information that allows to evaluate the property with respect to the processed prefix at the current and future time points (e.g., [LS87, Cho95, BGHS04b, RH05]). Ideally, the size of this encoding only depends on the monitored property and not on the size of the observed prefix. Non-trace-storing runtime monitors are suitable for *online* scenarios, where new state or event information is processed in an incremental manner as soon as it becomes available. Automata-based and rewriting-based methods are usually non-trace-storing. In the context of first-order query languages and databases, non-trace-storing algorithms are often based on some form of logical data expiration [Tom03].

Synchronicity. A second important feature concerns the synchronicity of a runtime monitor. A *synchronous* monitor provides the ability to detect property violations as soon as possible. As noted in [RH05], in general, synchronous monitoring algorithms require the ability to decide satisfiability of a given property specification, which, depending on the specification language, may be computationally expensive or even undecidable. For example, for biquantified FOTL formulae with no quantifier in the scope of a temporal operator, the satisfiability problem with respect to (infinite) temporal databases is decidable in exponential time. In contrast, for biquantified FOTL formulae with a single quantifier in the scope of a temporal operator, the same problem is undecidable [CN95]. Similarly, the satisfiability problem for past-only FOTL formulae over (infinite) temporal databases is undecidable [Cho95]. Runtime monitors that cannot guarantee to detect property violations always at the earliest possible time point are called *asynchronous*.

3.3 Applications of Runtime Monitoring

Runtime monitoring has many useful applications. Depending on the needs of the application, however, the runtime monitor can be integrated with differing degrees of coupling with the monitored system. For most applications, the runtime monitoring functionality must also be complemented by additional technology, for example, the ability to automatically react to violations of a monitored property.

In this section, we give a brief overview of possible types of applications of runtime monitoring as described in the previous sections. We first discuss a set of aspects

that require consideration when implementing runtime monitoring functionality for a specific application. Based on two of these aspects, we then provide a application classification of runtime monitoring technology.

3.3.1 Implementation Aspects

When applying runtime monitoring technology in the context of a specific application, a number of implementation aspects must usually be considered. We discuss three of them in the following.

Usage

Runtime monitors can be used in either of two modes. *Online* monitors are executed simultaneously with the monitored target system. As the target system changes its state or generates new events, relevant information is made available to the online monitor. The monitor thus observes an execution prefix of increasing size. Typically, online monitors process the new information in an incremental fashion by using non-trace-storing algorithms. In contrast, *offline* monitors are executed after a prefix of some system execution has been generated and recorded, for example, in a log file. Thus, offline monitors are often based on trace-storing algorithms.

Placement

The next aspect concerns the question of how a runtime monitor is integrated with the monitored target system. We distinguish between an *inline* and an *outline* placement of the monitor. In the case of inline monitoring, the monitoring module is integrated in the control flow of the monitored system such that each relevant state change or event is followed by a check of the monitor. Typically, the inline monitor also uses the same resource space as the monitored system. For software program monitoring, for example, an inline monitor is usually embedded directly into the code of the monitored software program. However, note that inline monitors can also run in a separate module or even on separate hardware. Observe that inline monitors are necessarily online and require synchronous algorithms.

In contrast, an outline monitor executes possibly simultaneously but otherwise independently of the target system. Usually, it is implemented in a separate module or as a separate processes. Outline monitors can be used in online or offline mode and may use of both synchronous or asynchronous algorithms.

Instrumentation

Instrumentation concerns the questions of how and when relevant states or events of the monitored target system are identified and communicated to the runtime monitor.

In principle, the instrumentation of the target system can be done in a *manual* or an *automated* fashion.

In the former case, for example, specific monitoring code (for inline monitoring), logging statements (for offline monitoring), or event emitting code (for online/outline monitoring) are manually added to the program code of the monitored system. In addition, the use of configurable logging or messaging frameworks such as Apache log4j [log] or JMS [jms] allows for flexible control over the amount and timing of communicated information.

As an alternative to manual instrumentation, toolkits that support various types of automated instrumentation have been proposed. A prominent example of such a toolkit is AspectJ [KHH+01], which is based on the aspect-oriented programming paradigm [KLM+97]. In AspectJ, a dedicated instrumentation specification (called an AspectJ program) defines when information about new states or events must be output and how the respective information is made available to the runtime monitor. A dedicated compiler then weaves the instrumentation code into the source or the byte code of the target system. Further examples of tools or frameworks supporting automated instrumentation are JPaX [HR04b], Java-MoP [CR05], Valgrind [NS07], or JavaMaC [KVK+04]. More detailed accounts on how software systems can be instrumented is provided in [PN81] or [Art05].

3.3.2 Application Classification

Applications of runtime monitoring technology can be classified in terms of the placement and the usage of the monitoring component. In the following, we briefly explain the resulting classification (depicted in Figure 3.2) and mention some concrete applications.

Inline Analysis and Runtime Enforcement

A first class of applications comprises those that use inlined runtime monitors (also known as reference monitors) to enable so-called runtime enforcement [Sch00, LBW05]. In these applications, each relevant state change is followed by an immediate check through the inlined runtime monitor. If an observed state change would result in a violation of the monitored property, a dedicated enforcement component either suppresses or modifies the attempted state change and thus prevents the monitored system from violating a given property.

Two example applications that are based on inlined runtime monitors are (history-based) access control and the checking of dynamic integrity constraints in databases. In the first case, a reference monitor evaluates each access request of the monitored (i.e., access-controlled) system against an access control property. A typical history-based access control property might read as follows: "a user u is only granted access

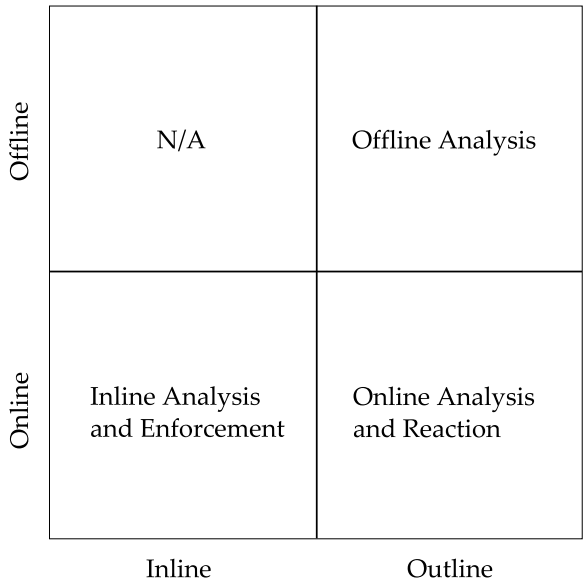


Figure 3.2: Classification of runtime monitoring applications.

to a resource r if u has the permission to access r and u has not accessed r within the previous hour". An access control system based on an inlined runtime monitor then enforces this property by only admitting those access requests that satisfy the policy. Access requests that would result in a violation of the property are denied by the access control system.

In a similar fashion, the maintenance of dynamic integrity constraints in databases can be implemented by using an inlined runtime monitor [Kun84, LS87, Pac97, Cho95, CT98]. Similar to a history-based access control property, dynamic integrity constraints can be formalized using an appropriate specification language such as past-only FOTL. As the monitored database is being updated, a runtime monitor analyzes every new database history against the property specification. A database update is only committed if the resulting database history satisfies the property specification. Otherwise, the update transaction is aborted or the database is rolled-back to the previous database state.

Online Analysis and Reaction

A second class of applications of runtime monitoring includes those that rely on online monitors which are placed outline of the monitored target system. While applications based on outlined runtime monitors cannot prevent the monitored system from violating a monitored property, they can still execute predefined operations as soon as the satisfaction or violation of a property is detected. Such operations can, for example, influence the monitored system, manipulate or notify other systems, trigger a compensating action, or simply record the violation.

Many applications are implemented based on this paradigm. For example, in oracle-

based testing, runtime monitors, so-called test oracles, are automatically derived from a formal test case specification. The test oracles are then executed in parallel with the system under test and violations of specified assertions are reported to the test engineer [TOOD96, HJL03, DtSD05]. If the test oracle is only used before a system enters the production stage, also the question of test coverage must be addressed to obtain sufficient confidence into the correctness of the tested system [ABG+05]. A concrete example of an oracle-based testing framework for real-time systems that builds on timed automata is T-UPPAAL [MLN04, LMN04, HLM+08].

A second example based on this application paradigm are temporal database triggers [SW95]. Intuitively, a temporal database trigger is an expression of the form **if** ϕ **then** a, where ϕ is a temporal logical formula interpreted over finite database histories and a is an *action* defined in a specific action language. Whenever the current database history is updated, the formula ϕ is evaluated. If the updated database history satisfies ϕ, the action a is executed.

Further applications based on the same principles are specification-based intrusion detection [NST04, vL06], complex event processing [GJS92, CL01, SSS+03, Owe07], and monitoring-oriented programming [CR03, CdR04].

Offline Analysis

The third class of applications is characterized by an offline usage and an outline placement of the monitoring component. The classical example for this class are applications are methods for the analysis of log files (e.g., [RGL01, FS04, CS06]). This class also includes those applications where relevant log data is gathered in a database during program execution and later analyzed by means of SQL or similar queries (e.g., [KMSF01, GOA05]).

3.4 Summary

In this chapter, we have presented a systematic overview of current runtime monitoring approaches. We first defined a runtime monitor as an algorithm that processes a finite prefix of a possibly infinite system execution and checks whether the observed prefix satisfies or violates a given property. We then sketched the issues that arise when evaluating infinite properties over finite prefixes of executions and presented a rough characterization of those properties that can be monitored.

We then surveyed different approaches to runtime monitoring and classified existing work in terms of the formal model used to represent system executions, the specification language, and the monitoring method. The classification of existing work differed depending on the research field. Temporal databases and restricted first-order temporal logics were typically used for monitoring in the context of dynamic integrity

checking for databases. This contrasts with research in the software program monitoring community. Here most work has focussed on the use of propositional executions models and propositional temporal logics so far. Moreover, while a large number of runtime monitoring approaches for propositional temporal logics relies on automata-based techniques, runtime monitoring methods for first-order temporal logics either use alternative techniques or syntactically restrict the use of quantification.

In the last section, we gave a brief overview of three classes of applications. The first class relies on inlined reference monitors and aims to prevent property violations altogether. The second class uses outlined runtime monitors that observe the behavior of reactive system in an online setting and may react to property satisfactions or violations by means of executing specific operations. The third class of applications makes use of offline monitors and, for example, is used to analyze log files.

Naturally, the practical usefulness of any runtime monitoring approach is largely determined by the interplay of two factors. On the one hand, we must ask whether the expressiveness provided by the specification language is sufficient to express a set of properties deemed relevant for a particular application context. On the other hand, the space consumed by a generated runtime monitor should be feasible in the application context under consideration. Typically, the expressiveness of a specification language is positively correlated with the complexity of the respective runtime monitoring problem. This simple relationship naturally stipulates the guiding principle to always select that combination of execution model, specification language, and respective monitoring method that provides enough flexibility in terms of the expressiveness of the specification language with the least space or time complexity.

"The infinite is in the finite of every instant."

Zen Proverb

Chapter 4

Monitoring with Automatic Structures

In this chapter, we present a general approach to the runtime monitoring of properties specified using MFOTL. In a nutshell, our monitor works as follows: Given an MFOTL formula $\Box\phi$ over a signature S, where ϕ is bounded, we first transform ϕ into a first-order formula $\hat{\phi}$ over an extended signature $\hat{S}$, obtained by augmenting S with auxiliary predicates for every temporal subformula in ϕ. Our monitor then incrementally processes a timed temporal structure (D, τ) over S and determines for each time point i those elements in (D, τ) that violate ϕ. This is achieved by incrementally constructing a collection of finite state automata that finitely represent the (possibly infinite) interpretations of the auxiliary predicates and by evaluating the transformed first-order formula $\neg\hat{\phi}$ over the extended $\hat{S}$-structure at every time point.

This chapter is structured as follows. In Section 4.1, we first explain the restrictions that are necessary to apply our monitoring approach. Before we present our approach in detail, a high-level overview is presented in Section 4.2. In Section 4.3, we then explain how to extend the signature of the monitored structure and how to transform the monitored formula. Subsequently, Section 4.4 presents how we incrementally construct an extended structure over the transformed signature and thus reduce monitoring to first-order query evaluation. In Section 4.5 we then illustrate our constructions by a simple example before we present our monitoring algorithm in Section 4.6. We conclude the chapter with a summary in Section 4.7.

4.1 Restrictions

Since first-order logic is undecidable and not all temporal properties can be monitored, we restrict both the formulae and the temporal structures under consideration. We now discuss these restrictions.

Throughout this chapter, let (D, τ) be a timed temporal structure over the signature $S = (\mathsf{C}, \mathsf{R}, a)$ and ψ the formula to be monitored. Without loss of generality, we assume that each subformula of ϕ has the vector of free variables $\bar{x} = (x_1, \ldots, x_n)$. We make

the following restrictions on ψ and D. First, we require ψ to be of the form $\Box\,\phi$, where ϕ is bounded. It follows that ψ describes a safety property. Recall that not all safety properties can be expressed by formulae of this form [CN95]. This is in contrast to propositional linear temporal logic, where every safety property can be expressed as $\Box\,\beta$, where β contains only past-time operators [LPZ85].

Example 4.1.1. *Let us give two examples of system properties expressed in the* MFOTL *fragment that our monitor can handle. First, the property "whenever the program variable in stores the input x, within 5 time units x must be stored in the program variable out" can be expressed by $\Box\,\forall x.\, in(x) \rightarrow \Diamond_{[0,6)}\, out(x)$. Second, the property "the value of the program variable v increases by 1 in each step from an initial value 0 until it becomes 5, then it stays constant" can be formalized as $\Box(\neg(\bullet\, true) \rightarrow v(0)) \wedge (\exists i.\, v(i) \wedge i \prec 5 \rightarrow \bigcirc\, v(i+1)) \wedge (v(5) \rightarrow \bigcirc\, v(5))$. Note that we use relations that are singletons to model program variables.*

Example 4.1.2 (Running example)**.** *Let S_{TPS} be the signature of the* TPS *as defined in Example 2.3.2. Recall that our example property from Section 1.3 can be expressed in* MFOTL *as*

$$\Box\,\forall t.\forall c.\, trans(t,c) \wedge \left(\blacklozenge_{[0,31)}\, \exists t'.\, trans(t',c) \wedge \Diamond_{[0,3)}\, report(t',c)\right) \rightarrow \Diamond_{[0,3)}\, report(t,c).$$

Note that the formula belongs to the restricted MFOTL *fragment.*

Second, we require that each structure in D is automatic. Recall that each automatic structure can be finitely represented by a collection of automata over finite words. Remember that we assume that for an automatic structure, we always have an automatic representation for it at hand.

In addition to the requirement that each structure in D is automatic, we require that D has a constant domain representation. This means that the domain of each D_i is represented by the same regular language $\mathcal{L}_{|D|}$ and each word in $\mathcal{L}_{|D|}$ represents the same element in $|D|$, i.e., each automatic representation has the same function $\nu : \mathcal{L}_{|D|} \rightarrow |D|$. Finally, we assume that $|D| = \mathbb{N}$ and that $<$ is the standard ordering on $\mathbb{N}$. This is without loss of generality whenever the function ν is injective, i.e., every element in $|D|$ has only one representative in $\mathcal{L}_{|D|}$. Recall that every automatic structure has an automatic representation in which the function ν is injective [KN95] (see Section 2.5.2).

Remark 4.1.3. *Let us recall some properties of automatic structures that we will use later. First, for a first-order formula θ, we can effectively construct an automaton that represents the set θ^{D_i}. This follows from the closure properties of regular languages. Second, some basic arithmetical relations are first-order definable in the structure $(\mathbb{N}, <)$ and thus regular. In particular, the successor relation $\{(x,y) \in \mathbb{N}^2 \mid y = x+1\}$ is regular, since the formula $x \prec y \wedge \neg\exists z.\, x \prec z \wedge z \prec y$ defines it. It is also easy to see that $\{(x,y) \in \mathbb{N}^2 \mid x + d \leq y\}$ is regular, for any $d \in \mathbb{N}$.*

4.2 Overview of the Monitoring Method

Our monitoring method for the formula $\Box\,\phi$ identifies at each time point the set of elements that violate ϕ. Formally, for the given timed temporal structure (D, τ), our runtime monitoring method determines at each time point $i \in \mathbb{N}$ the set

$$\{\bar{a} \mid (D, \tau, v[\bar{x}/\bar{a}], i) \not\models_{\text{MFOTL}} \phi, \text{ for some valuation } v\}.$$

To monitor $\Box\,\phi$ with respect to the given timed temporal structure (D, τ), we incrementally build a sequence of structures $\hat{D}_0, \hat{D}_1, \ldots$ over an extended signature $\hat{S}$. The extension depends on the temporal subformulae of ϕ. Moreover, we transform ϕ by replacing each temporal subformula α by an auxiliary predicate p_α. For each time point i, we then determine the elements that violate ϕ by evaluating a transformed formula $\neg\hat{\phi} \in \mathsf{FOL}$ over $\hat{D}_i$. Formally, at each time point $i \in \mathbb{N}$, we thus determine the set

$$\{\bar{a} \mid (\hat{D}_i, v[\bar{x}/\bar{a}]) \models_{\text{FOL}} \neg\hat{\phi}, \text{ for some valuation } v\},$$

where we require that $p_\alpha^{\hat{D}_i} = \{\bar{a} \mid (\hat{D}_i, v[\bar{x}/\bar{a}]) \models_{\text{FOL}} p_\alpha, \text{ for some valuation } v\}$, for each temporal subformula α.

Observe that with future operators, we usually cannot determine this set yet when time point i occurs. Our monitor, which we present in Section 4.6, therefore maintains a list of unevaluated subformulae for past time points.

In the following, we first describe how we extend S and transform ϕ. Afterwards, we explain how we effectively and incrementally build $\hat{D}_i$. Finally, we present our monitor and prove its correctness.

4.3 Signature Extension and Formula Transformation

In addition to the predicates in R, the extended signature $\hat{S}$ contains an auxiliary predicate p_α for each temporal subformula α of ϕ. For subformulae of the form $\beta\,\mathcal{S}_I\,\gamma$ and $\beta\,\mathcal{U}_I\,\gamma$, we introduce further predicates, which store information that allow us to incrementally update the auxiliary relations.

Definition 4.3.1. *Let $\hat{S} := (\hat{\mathsf{C}}, \hat{\mathsf{R}}, \hat{a})$ be the signature with $\hat{\mathsf{C}} := \mathsf{C}$ and*

$$\begin{aligned}\hat{\mathsf{R}} := \mathsf{R} \cup \{&p_\alpha \mid \alpha \textit{ temporal subformula of } \phi\} \cup \\ &\{r_\alpha \mid \alpha \textit{ subformula of } \phi \textit{ of the form } \beta\,\mathcal{S}_I\,\gamma \textit{ or } \beta\,\mathcal{U}_I\,\gamma\} \cup \\ &\{s_\alpha \mid \alpha \textit{ subformula of } \phi \textit{ of the form } \beta\,\mathcal{U}_I\,\gamma\}\,.\end{aligned}$$

For $r \in \mathsf{R}$, let $\hat{a}(r) := a(r)$. If α is a temporal subformula with n free variables, then $\hat{a}(p_\alpha) := n$, and $\hat{a}(r_\alpha) := n{+}1$ and $\hat{a}(s_\alpha) := n{+}2$, if r_α and s_α exist. We assume that $p_\alpha, r_\alpha, s_\alpha \notin \mathsf{C} \cup \mathsf{R} \cup \mathsf{V}$.

We transform MFOTL formulae over the signature S into first-order formulae over the extended signature $\hat{S}$ as follows.

Definition 4.3.2. *For $\theta \in$ MFOTL with the vector of free variables $\bar{x}$, we define*

$$\hat{\theta} := \begin{cases} \neg\hat{\beta} & \text{if } \theta = \neg\beta, \\ \hat{\beta} \wedge \hat{\gamma} & \text{if } \theta = \beta \wedge \gamma, \\ \exists y.\, \hat{\beta} & \text{if } \theta = \exists y.\, \beta, \\ p_\theta(\bar{x}) & \text{if } \theta \text{ is a temporal formula,} \\ \theta & \text{is an atomic formula.} \end{cases}$$

The formula transformation has the following properties.

Lemma 4.3.3. *Let θ be a subformula of ϕ. For all $i \in \mathbb{N}$, the following properties hold:*
(i) If $p_\alpha^{\hat{D}_i} = \alpha^{(D,\tau,i)}$ for all $\alpha \in tsub(\theta)$, then $\hat{\theta}^{\hat{D}_i} = \theta^{(D,\tau,i)}$.
(ii) If $p_\alpha^{\hat{D}_i}$ is regular for all $\alpha \in tsub(\theta)$, then $\hat{\theta}^{\hat{D}_i}$ is regular.

Proof. A simple induction over the formula structure of $\hat{\theta}$ establishes (i) and (ii). ⊣

Example 4.3.4 (Running example)**.** *Let S_{TPS} be the signature of the* TPS *as defined in Example 2.3.2. Recall that our example property can be formally expressed in* MFOTL *as*

$$\Box\, \forall t. \forall c. trans(t,c) \wedge \big(\blacklozenge_{[0,31)}\, \exists t'. trans(t',c) \wedge \Diamond_{[0,3)}\, report(t',c)\big) \rightarrow \Diamond_{[0,3)}\, report(t,c).$$

To determine the elements that violate this property, we first drop the outermost temporal operator $\Box$. We then negate the remaining formula and remove the outermost quantifiers. The resulting formula

$$trans(t,c) \wedge \big(\blacklozenge_{[0,31)}\, \exists t'. trans(t',c) \wedge \Diamond_{[0,3)}\, report(t',c)\big) \wedge \neg\big(\Diamond_{[0,3)}\, report(t,c)\big)$$

identifies at each time point those tuples that violate the example property. Replacing the derived temporal operators $\blacklozenge_{[0,366)}$ and $\Diamond_{[0,21)}$ with their respective definitions, we obtain

$$trans(t,c) \wedge \underbrace{\big(true\, \mathcal{S}_{[0,31)}\, \exists t'. trans(t',c) \wedge \underbrace{\big(true\, \mathcal{U}_{[0,3)}\, report(t',c)\big)}_{\theta_1}\big)}_{\theta_3} \wedge \neg \underbrace{\big(true\, \mathcal{U}_{[0,3)}\, report(t,c)\big)}_{\theta_2}.$$

We then extend the signature S_{TPS} by introducing additional predicates for the temporal subformulae θ_1, θ_2, and θ_3. According to Definition 4.3.1, in addition to the predicates of S_{TPS}, the extended signature $\hat{S}_{\text{TPS}}$ thus includes the 4-ary auxiliary predicates s_{θ_1} and s_{θ_2}, the ternary auxiliary predicates r_{θ_1} and r_{θ_2}, the binary predicates r_{θ_3}, p_{θ_1}, and p_{θ_2}, and the unary predicate p_{θ_3}, respectively. From the extended signature and Definition 4.3.2, we obtain the transformed formula

$$trans(t,c) \wedge p_{\theta_3}(c) \wedge \neg p_{\theta_2}(t,c).$$

4.4 Incremental Extended Structure Construction

We now show how the auxiliary relations in the $\hat{D}_i$s are incrementally constructed. Their instantiations are computed recursively both over time and over the formula structure, where evaluations of subformulae may also be needed from future time points. We later show that this is well-defined and can be evaluated incrementally.

For $c \in \mathsf{C}$ and $r \in \mathsf{R}$, we define $c^{\hat{D}_i} := c^{D_i}$ and $r^{\hat{D}_i} := r^{D_i}$. We address the construction of the auxiliary relations for each type of temporal operator separately.

4.4.1 Previous

We first address the past operator $\bullet_I$.

Definition 4.4.1. *For $\alpha = \bullet_I \beta$ with $I \in \mathbb{I}$, we define*

$$p_\alpha^{\hat{D}_i} := \begin{cases} \hat{\beta}^{\hat{D}_{i-1}} & \text{if } i > 0 \text{ and } \tau_i - \tau_{i-1} \in I, \\ \emptyset & \text{otherwise.} \end{cases}$$

Intuitively, a tuple $\bar{a}$ is in $p_\alpha^{\hat{D}_i}$ if $\bar{a}$ satisfies β at the previous time point $i-1$ and the difference of the two successive time stamps is in the interval I given by the metric temporal operator $\bullet_I$.

Lemma 4.4.2. *Let $\alpha = \bullet_I \beta$. For $i > 0$, if $p_\delta^{\hat{D}_{i-1}}$ is regular and $p_\delta^{\hat{D}_{i-1}} = \delta^{(D,\tau,i-1)}$ for all $\delta \in tsub(\beta)$, then $p_\alpha^{\hat{D}_i}$ is regular and $p_\alpha^{\hat{D}_i} = \alpha^{(D,\tau,i)}$. Moreover, $p_\alpha^{\hat{D}_0}$ is regular and $p_\alpha^{\hat{D}_0} = \alpha^{(D,\tau,0)}$.*

Proof. For $i = 0$, the lemma obviously holds. For $i > 0$, the regularity of $p_\alpha^{\hat{D}_i}$ follows from the assumption that the relations $p_\delta^{\hat{D}_{i-1}}$ are regular and Lemma 4.3.3(ii). The equality of the two sets follows from Lemma 4.3.3(i) and the semantics of the temporal operator $\bullet_I$. ⊣

4.4.2 Next

We next address the future operator $\bigcirc_I$.

Definition 4.4.3. *For $\alpha = \bigcirc_I \beta$ with $I \in \mathbb{I}$, we define*

$$p_\alpha^{\hat{D}_i} := \begin{cases} \hat{\beta}^{\hat{D}_{i+1}} & \text{if } \tau_{i+1} - \tau_i \in I, \\ \emptyset & \text{otherwise.} \end{cases}$$

Note that the definition of $p_\alpha^{\hat{D}_i}$ depends on the relations of the next structure D_{i+1} and on the auxiliary relations for $\delta \in tsub(\beta)$ of the next extended structure $\hat{D}_{i+1}$. Hence, the monitor instantiates $p_\alpha^{\hat{D}_i}$ with a delay of at least one time step.

Lemma 4.4.4. *Let $\alpha = \bigcirc_I \beta$. If $p_\delta^{\hat{D}_{i+1}}$ is regular and $p_\delta^{\hat{D}_{i+1}} = \delta^{(D,\tau,i+1)}$ for all $\delta \in tsub(\beta)$, then $p_\alpha^{\hat{D}_i}$ is regular and $p_\alpha^{\hat{D}_i} = \alpha^{(D,\tau,i)}$.*

Proof. For $i \geq 0$, the regularity of $p_\alpha^{\hat{D}_i}$ follows from the assumption that the relations $p_\delta^{\hat{D}_{i+1}}$ are regular and Lemma 4.3.3(ii). The equality of the two sets follows from Lemma 4.3.3(i) and the semantics of the temporal operator $\bigcirc_I$. ⊣

4.4.3 Since

We now address the past operator $\mathcal{S}_I$ with $I = [c, d) \in \mathbb{I}$. Assume $\alpha = \beta \, \mathcal{S}_I \, \gamma$. We start with the initialization and update of the auxiliary relations for r_α.

Definition 4.4.5. *For $i = 0$, we define*

$$r_\alpha^{\hat{D}_0} := \hat{\gamma}^{\hat{D}_0} \times \{0\},$$

and for $i > 0$, we define

$$\begin{aligned} r_\alpha^{\hat{D}_i} := \; & \big(\hat{\gamma}^{\hat{D}_i} \times \{0\}\big) \cup \\ & \big\{(\bar{a}, y) \in \mathbb{N}^{n+1} \,\big|\, \bar{a} \in \hat{\beta}^{\hat{D}_i},\ y < d,\ \textit{and}\ (\bar{a}, y') \in r_\alpha^{\hat{D}_{i-1}},\ \textit{for}\ y' = y - \tau_i + \tau_{i-1}\big\}. \end{aligned}$$

Intuitively, a pair $(\bar{a}, y)$ is in $r_\alpha^{\hat{D}_i}$ if $\bar{a}$ satisfies α at the time point i independent of the lower bound c, where the "age" y indicates how long ago the formula γ was satisfied by $\bar{a}$. If $\bar{a}$ satisfies γ at the time point i, it is added to $r_\alpha^{\hat{D}_i}$ with the age 0. For $i > 0$, we additionally update the tuples $(\bar{a}, y) \in r_\alpha^{\hat{D}_{i-1}}$. First, $\bar{a}$ must satisfy β at the time point i. Second, the age is adjusted by the difference of the time stamps τ_{i-1} and τ_i. Third, the new age must be less than d, otherwise it is too old to satisfy α.

The arithmetic constraint $y' = y - \tau_i + \tau_{i-1}$ in the definition of $r_\alpha^{\hat{D}_i}$ for $i > 0$ is first-order definable in D, see Remark 4.1.3. Note that $\tau_i + \tau_{i-1}$ is a constant value. Now it is not hard to see that $r_\alpha^{\hat{D}_i}$ is regular if all its components are regular.

With the relation $r_\alpha^{\hat{D}_i}$, we can determine the elements that satisfy α at the time point i.

Definition 4.4.6. *For $i \in \mathbb{N}$, we define*

$$p_\alpha^{\hat{D}_i} := \big\{\bar{a} \in \mathbb{N}^n \,\big|\, (\bar{a}, y) \in r_\alpha^{\hat{D}_i}, \textit{for some}\ y \geq c\big\}.$$

Lemma 4.4.7. *Let $\alpha = \beta \, \mathcal{S}_{[c,d)} \, \gamma$. Assume that $p_\delta^{\hat{D}_j}$ is regular and $p_\delta^{\hat{D}_j} = \delta^{(D,\tau,j)}$, for all $j \leq i$ and $\delta \in \mathit{tsub}(\beta) \cup \mathit{tsub}(\gamma)$. Then the following properties hold:*

(i) The relation $r_\alpha^{\hat{D}_i}$ is regular and for all $\bar{a} \in \mathbb{N}^n$ and $y \in \mathbb{N}$,

$$(\bar{a}, y) \in r_\alpha^{\hat{D}_i} \quad \textit{iff} \quad \begin{array}{c} \textit{there is a } j \in [0, i+1) \textit{ such that } y = \tau_i - \tau_j < d\,, \bar{a} \in \gamma^{(D,\tau,j)}\,, \\ \textit{and } \bar{a} \in \beta^{(D,\tau,k)}, \textit{ for all } k \in [j+1, i+1)\,. \end{array}$$

(ii) The relation $p_\alpha^{\hat{D}_i}$ is regular and $p_\alpha^{\hat{D}_i} = \alpha^{(D,\tau,i)}$.

Proof. Property (ii) follows immediately from (i) and the definition of $p_\alpha^{\hat{D}_i}$. Note that we use Lemma 4.3.3 without explicitly referring to it. Let v_0 denote any valuation.

We prove (i) by induction over i.

Base case $i = 0$: The set $r_\alpha^{\hat{D}_0}$ is regular, since it can be defined by the formula

$$\psi(\bar{x}, y) := \hat{\gamma}(\bar{x}) \wedge \neg\exists z.\ \mathrm{succ}(z, y)\,.$$

Note that, by assumption, the relations for the predicates in $\hat{\gamma}$ are regular.

The equivalence for $i = 0$ follows directly from the definition of $r_\alpha^{\hat{D}_0}$. Note that $\tau_i - \tau_i < d$, since in the definition of the syntax of MFOTL, we require that $I \neq \emptyset$. Hence, $d > 0$.

Step case $i > 0$: We first show that $r_\alpha^{\hat{D}_i}$ is regular. Similar to the base case, it follows that the set $S := \hat{\gamma}^{\hat{D}_i} \times \{0\}$ is regular. The set $T := \{(\bar{a}, y) \in \mathbb{N}^{n+1} \mid y < d,\ \bar{a} \in \hat{\beta}^{\hat{D}_i}$, and $(\bar{a}, y') \in r_\alpha^{\hat{D}_{i-1}},$ for $y' = y - \tau_i + \tau_{i-1}\}$ is also regular. It can be expressed by the first-order formula

$$\psi(\bar{x}, y) := y \prec d \wedge \hat{\beta}(\bar{x}) \wedge \exists y'.\ \psi'(\bar{x}, y') \wedge y' + (\tau_i - \tau_{i-1}) \approx y\,,$$

where ψ' is the formula that defines $r_\alpha^{\hat{D}_{i-1}}$, which is regular by the induction hypothesis. Note that d and $\tau_i - \tau_{i-1}$ are constant values and not variables. Since $r_\alpha^{\hat{D}_i}$ is defined as the union of S and T, we conclude that $r_\alpha^{\hat{D}_i}$ is regular.

Now, we show the step case for the other claim.

($\Rightarrow$) If the tuple $(\bar{a}, y)$ is in S, then the claim is obviously true. Assume that $(\bar{a}, y) \in T$. By definition, there is a tuple $(\bar{a}, y')$ in $r_\alpha^{\hat{D}_{i-1}}$ such that $y' = y - \tau_i + \tau_{i-1}$. By the induction hypothesis, we have that

$$\begin{aligned}\exists j \in [0, i) : y' = \tau_{i-1} - \tau_j < d,\ (D, \tau, v_0[\bar{x}/\bar{a}], j) \models \gamma, \text{ and}\\ \forall k \in [j+1, i) : (D, \tau, v_0[\bar{x}/\bar{a}], k) \models \beta\,.\end{aligned}$$

It follows that $y = y' + \tau_i - \tau_{i-1} = \tau_i - \tau_j$. From the assumption, we conclude that $(D, \tau, v_0[\bar{x}/\bar{a}], k) \models \beta$, for all k with $j < k \leq i$.

($\Leftarrow$) If $j = i$, it follows that $y = 0$. From the assumption and the definition of $r_\alpha^{\hat{D}_i}$, it follows that $(\bar{a}, 0) \in r_\alpha^{\hat{D}_i}$. Assume that $j < i$. By the induction hypothesis, $(\bar{a}, y') \in r_\alpha^{\hat{D}_{i-1}}$ with $y' = y - (\tau_i - \tau_{i-1})$. From the definition of $r_\alpha^{\hat{D}_i}$ and the assumption, we conclude that $(\bar{a}, y) \in r_\alpha^{\hat{D}_i}$. ⊣

Note that the definition of $r_\alpha^{\hat{D}_i}$ only depends on the relation $r_\alpha^{\hat{D}_{i-1}}$, if $i > 0$, and on the relations in $\hat{D}_i$ for which the corresponding predicates occur in the subformulae of $\hat{\beta}$ or $\hat{\gamma}$. Furthermore, the definition of $p_\alpha^{\hat{D}_i}$ only depends on $r_\alpha^{\hat{D}_i}$.

4.4.4 Until

We now address the bounded future operator $\mathcal{U}_I$ with $I = [c, d) \in \mathbb{I}$ and $d \in \mathbb{N}$. Assume that $\alpha = \beta \mathcal{U}_I \gamma$. For all $i \in \mathbb{N}$, let $\ell_i := \max\{j \in \mathbb{N} \mid \tau_{i+j} - \tau_i < d\}$. We call ℓ_i the lookahead offset at time point i. For convenience, let $\ell_{-1} := 0$. To instantiate the relation $p_\alpha^{\hat{D}_i}$, only

the relations $p_\delta^{\hat{D}_i}, \ldots, p_\delta^{\hat{D}_{i+\ell_i}}$ are relevant, where $\delta \in tsub(\beta) \cup tsub(\gamma)$. The definition of $p_\alpha^{\hat{D}_i}$ is based on the auxiliary relations $r_\alpha^{\hat{D}_i}$ and $s_\alpha^{\hat{D}_i}$, which we first show how to initialize and update.

We define $r_\alpha^{\hat{D}_i}$ as the union of the sets N_r and U_r. N_r contains the tuples that are new in the sense that they are obtained from data at the time points $i + \ell_{i-1}, \ldots, i + \ell_i$; U_r contains the updated data from the time points $i, \ldots, i + \ell_{i-1} - 1$.

Definition 4.4.8. *For $i \in \mathbb{N}$, we define $r_\alpha^{\hat{D}_i} := N_r \cup U_r$, where*

$$N_r \;:=\; \left\{(\bar{a}, j) \in \mathbb{N}^{n+1} \,\middle|\, \ell_{i-1} \leq j \leq \ell_i,\ \bar{a} \in \hat{\gamma}^{\hat{D}_{i+j}},\ \textit{and}\ \tau_{i+j} - \tau_i \geq c\right\}$$

and

$$U_r \;:=\; \begin{cases} \left\{(\bar{a}, j) \in \mathbb{N}^{n+1} \,\middle|\, (\bar{a}, j+1) \in r_\alpha^{\hat{D}_{i-1}}\ \textit{and}\ \tau_{i+j} - \tau_i \geq c\right\} & \textit{if } i > 0, \\ \emptyset & \textit{otherwise.} \end{cases}$$

Intuitively, $r_\alpha^{\hat{D}_i}$ stores the tuples satisfying the formula $\Diamond_I \gamma$ at the time point i, where each tuple in $r_\alpha^{\hat{D}_i}$ is augmented by the index relative to i where the tuple satisfies γ.

Similarly to $r_\alpha^{\hat{D}_i}$, the relation $s_\alpha^{\hat{D}_i}$ is the union of a set N_s for the new elements and a set U_s for the updates.

Definition 4.4.9. *For $i \in \mathbb{N}$, we define $s_\alpha^{\hat{D}_i} := N_s \cup U_s$, where*

$$N_s \;:=\; \left\{(\bar{a}, j, j') \in \mathbb{N}^{n+2} \,\middle|\, \ell_{i-1} \leq j \leq j' \leq \ell_i\ \textit{and}\ \bar{a} \in \hat{\beta}^{\hat{D}_{i+k}},\ \textit{for all}\ k \in [j, j'+1)\right\}$$

and

$$U_s \;:=\; \begin{cases} \left\{(\bar{a}, j, j') \in \mathbb{N}^{n+2} \,\middle|\, (\bar{a}, j+1, j'+1) \in s_\alpha^{\hat{D}_{i-1}}\right\} \cup \\ \left\{(\bar{a}, j, j') \in \mathbb{N}^{n+2} \,\middle|\, (\bar{a}, j+1, \ell_{i-1}) \in s_\alpha^{\hat{D}_{i-1}}\ \textit{and}\ (\bar{a}, \ell_{i-1}, j') \in N_s\right\} & \textit{if } i > 0 \\ \emptyset & \textit{otherwise.} \end{cases}$$

Intuitively, $s_\alpha^{\hat{D}_i}$ stores the tuples and the bounds of the interval (relative to i) in which β is satisfied.

With the relations $r_\alpha^{\hat{D}_i}$ and $s_\alpha^{\hat{D}_i}$ at hand, we define the relation $p_\alpha^{\hat{D}_i}$ as follows.

Definition 4.4.10. *For $i \in \mathbb{N}$, we define*

$$p_\alpha^{\hat{D}_i} := \left\{\bar{a} \in \mathbb{N}^n \,\middle|\, (\bar{a}, j) \in r_\alpha^{\hat{D}_i}\ \textit{and}\ (\bar{a}, 0, j') \in s_\alpha^{\hat{D}_i}, \textit{for some}\ j \leq j' + 1\right\}.$$

Lemma 4.4.11. *Let $\alpha = \beta \mathcal{U}_I \gamma$. Assume that $p_\delta^{\hat{D}_k}$ is regular and $p_\delta^{\hat{D}_k} = \delta^{(D,\tau,k)}$, for all $k \leq i + \ell_i$ and $\delta \in tsub(\beta) \cup tsub(\gamma)$. Then the following properties hold:*

(i) The relation $r_\alpha^{\hat{D}_i}$ is regular and for all $\bar{a} \in \mathbb{N}$ and $j \in \mathbb{N}$,

$$(\bar{a}, j) \in r_\alpha^{\hat{D}_i} \qquad \textit{iff} \qquad \bar{a} \in \gamma^{(D,\tau,i+j)}\ \textit{and}\ \tau_{i+j} - \tau_i \in I\,.$$

(ii) The relation $s_\alpha^{\hat{D}_i}$ is regular and for all $\bar{a} \in \mathbb{N}^n$ and $j, j' \in \mathbb{N}$,

$$(\bar{a}, j, j') \in s_\alpha^{\hat{D}_i} \quad \textit{iff} \quad j \leq j',\ \tau_{i+j'} - \tau_i < d,\ \textit{and}\ \bar{a} \in \beta^{(D,\tau,i+k)},\ \textit{for all}\ k \in [j, j'+1)\,.$$

(iii) The relation $p_\alpha^{\hat{D}_i}$ is regular and $p_\alpha^{\hat{D}_i} = \alpha^{(D,\tau,i)}$.

Proof. Property (iii) follows immediately from (i), (ii), and the definition of $p_\alpha^{\hat{D}_i}$. Again we use Lemma 4.3.3 without explicitly referring to it. Let v_0 denote any valuation.

Let us first prove (i), which we do by induction over i.

Base case $i = 0$: We first show that $r_\alpha^{\hat{D}_i}$ is regular. It suffices to show that the set N_r is regular. The regularity of N_r can be seen as follows. For $j \in \mathbb{N}$ with $\ell_{i-1} \leq j \leq \ell_i$, we define $N_r^j := \emptyset$ if $\tau_{i+j} - \tau_i < c$, and $N_r^j := \hat{\gamma}^{\hat{D}_{i+j}} \times \{j\}$ otherwise. Obviously, N_r^j is regular. Since $N_r = \bigcup_{\ell_{i-1} \leq j \leq \ell_i} N_r^j$, we conclude that N_r is regular.

Note that by definition of ℓ_i we have $\tau_{\ell_i} - \tau_j < d$, for every $j \leq \ell_i$. The equivalence follows directly from the definition of N_r.

Step case $i > 0$: We first show that $r_\alpha^{\hat{D}_i}$ is regular. It suffices to show that the sets N_r and U_r are regular. The regularity of N_r can be shown as in the base case. The regularity of U_r can be seen as follows. The set $H := \{j \in \mathbb{N} \mid \ell_{i-1} \leq j \leq \ell_i \text{ and } \tau_{i+j} - \tau_j \geq c\}$ is regular, since it is finite. The set U_r can be defined by the formula $h(z) \wedge \exists z'.\ \mathrm{succ}(z, z') \wedge r(\bar{x}, z')$, where h denotes the formula that defines H and r denotes the formula that defines the set $r_\alpha^{\hat{D}_{i-1}}$, which is regular by induction hypothesis.

If $j \geq \ell_{i-1}$, the equivalence follows, similar to the base case, directly from the definition of ℓ_i. In the following, assume $j < \ell_{i-1}$.

($\Rightarrow$) We have that $(\bar{a}, j) \in U_r$. By definition, $(\bar{a}, j+1) \in r_\alpha^{\hat{D}_{i-1}}$ and $\tau_{i+j} - \tau_i \geq c$. By the induction hypothesis, $\bar{a} \in \hat{\gamma}^{\hat{D}_{i-1+j+1}}$ and $\tau_{i-1+j+1} - \tau_{i-1} \in I$. Since the difference of the time stamps from $i-1$ to $i+j$ and from i to $i+j$ decreases, we have that $\tau_{i+j} - \tau_i < d$. We are done since $\tau_{i+j} - \tau_i \geq c$ by the definition of U_r.

($\Leftarrow$) From the definition of ℓ_i it follows that $\tau_{i-1+j+1} - \tau_{i-1} < d$. Hence, $\bar{a} \in \hat{\gamma}^{\hat{D}_{i-1+j+1}}$ and $\tau_{i-1+j+1} - \tau_{i-1} \in I$. By the induction hypothesis, $(\bar{a}, j+1) \in r_\alpha^{\hat{D}_{i-1}}$. From the definition of U_r, we conclude that $(\bar{a}, j) \in r_\alpha^{\hat{D}_i}$.

Let us now prove (ii), which we do again by induction over i.

Base case $i = 0$: We first show that $s_\alpha^{\hat{D}_i}$ is regular. It suffices to show that the set N_s is regular. To see that N_s is regular, let $N_s^{j,j'} := \bigcap_{j \leq k \leq j'} \hat{\beta}^{\hat{D}_{i+k}}$, for $j, j' \in \mathbb{N}$ with $\ell_{i-1} \leq j \leq j' \leq \ell_i$. Obviously, $N_s^{j,j'}$ is regular and $N_s = \bigcup_{\ell_{i-1} \leq j \leq j' \leq \ell_i} (N_s^{j,j'} \times \{(j, j')\})$. We conclude that N_s is regular.

Note that by the definition of ℓ_i we have $\tau_{\ell_i} - \tau_{j'} < d$, for every $j' \leq \ell_i$. The equivalence follows directly from the definition of N_s.

Step case $i > 0$: We first show that $r_\alpha^{\hat{D}_i}$ is regular. It suffices to show that the sets N_s and U_s are regular. The regularity of N_s can be shown as in the base case. The regularity of

U_s can be seen as follows. The formula

$$\big(\exists y'.\, \exists z'.\, \mathrm{succ}(y, y') \wedge \mathrm{succ}(z, z') \wedge s(\bar{x}, y', z')\big) \vee$$
$$\big(\exists y'.\, \exists z'.\, \mathrm{succ}(y, y') \wedge z' \approx \ell' \wedge s(\bar{x}, y', z') \wedge n(\bar{x}, z', z)\big)$$

defines U_r, where n is the formula that defines the set N_s and s is the formula that defines $s_\alpha^{\hat{D}_i - 1}$, which is regular by the induction hypothesis. Note that ℓ_{i-1} is a constant.

If $j' \geq \ell_{i-1}$, the equivalence follows, similar to the base case, directly from the definition of ℓ_i. For $j < \ell_{i-1}$, it suffices to prove that $(\bar{a}, j, j') \in U_s$ iff $j \leq j'$, $\tau_{i+j'} - \tau_i \leq d$, and $\bar{a} \in \hat{\gamma}^{\hat{D}_{i+k}}$, for all $k \in [j, j' + 1)$ The proof is similar as for (i). We omit it. $\dashv$

4.5 Example

For the sake of simplicity, instead of using our running example, we now illustrate the described transformations and constructions by using the formula

$$\Box\, \forall x.in(x) \rightarrow \Diamond_{[0,6)}\, out(x)$$

for Example 4.1.1. We observe that the formula is defined over a signature $S = (\mathsf{C}, \mathsf{R}, a)$, where R contains the unary predicates in and out.

As a first step, we remove syntactic sugar. We obtain the formula

$$\Box\, \neg\exists x.in(x) \wedge \neg\big(\exists y.y \approx y\, \mathcal{U}_{[0,6)}\, out(x)\big).$$

In order to detect violations of this formula, we negate it to obtain $\Diamond\, \theta$ with

$$\theta := \exists x.in(x) \wedge \neg\big(\exists y.y \approx y\, \mathcal{U}_{[0,6)}\, out(x)\big).$$

We extend the signature S according to Definition 4.3.1. Note that θ contains one temporal subformula, namely $\alpha := \exists y.y \approx y \mathcal{U}_{[0,6)}\, out(x)$. Hence, the extended signature $\hat{S}$ is obtained from S by adding the auxiliary predicates p_α, r_α, and s_α to R. According to Definition 4.3.2, we transform θ into the first-order formula

$$\hat{\theta} := \exists x.in(x) \wedge \neg p_\alpha(x).$$

For each time point i, we incrementally build the auxiliary relations $p_\alpha^{\hat{D}_i}$, $r_\alpha^{\hat{D}_i}$, and $s_\alpha^{\hat{D}_i}$ such that the auxiliary predicate p_α is satisfied by exactly those elements that satisfy α at i.

To illustrate how the auxiliary relations are built, let us consider the timed temporal structure given in Figure 4.1. Observe that to build the relations $r_\alpha^{\hat{D}_i}$, for $i \geq 0$, we require the relations out^{D_j} with $i \leq j \leq \ell_i$. Recall that ℓ_i is the lookahead offset at time point i. For example, at $i = 0$ we have that $\ell_0 = 3$ because $\tau_3 - \tau_0 < 6$ and $\tau_4 - \tau_0 = 6$. Hence, to build $r_\alpha^{\hat{D}_0}$, we need to take into account the relations out^{D_0}, out^{D_1}, out^{D_2}, and out^{D_3}. Moreover, as the subformula $\exists y.y \approx y$ is always true we do

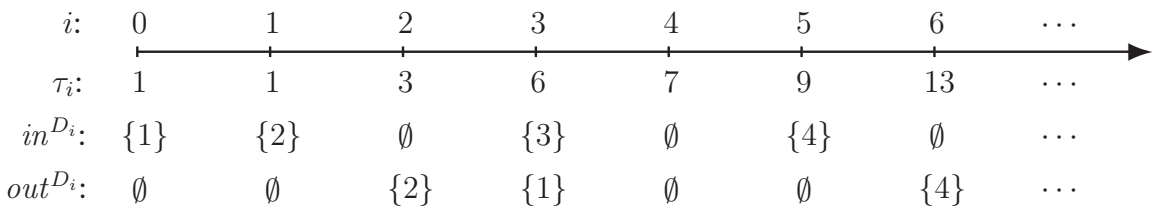


Figure 4.1: Example timed temporal structure.

not depend on any relations to build the relations $s_\alpha^{\hat{D}_i}$, for all $i \geq 0$. For $i = 0$, we thus have $r_\alpha^{\hat{D}_0} := \{(1,3),(2,2)\}$ and $s_\alpha^{\hat{D}_0} = \big(\mathbb{N}\times\{0\}\times\{0\}\big) \cup \big(\mathbb{N}\times\{0,1\}\times\{1\}\big) \cup \big(\mathbb{N}\times\{0,1,2\}\times\{2\}\big) \cup \big(\mathbb{N}\times\{0,1,2,3\}\times\{3\}\big)$. The first component of a pair in $r_\alpha^{\hat{D}_i}$ denotes an element occurring in a relation out^{D_j}, with $i \leq j \leq \ell_i$. The second component stores the difference $j-i$ between the respective indices. For example, the pair $(1,3)$ in $r_\alpha^{\hat{D}_0}$ means that the element 1 occurs in out^{D_3}. Similarly, while the first component of a triple in $s_\alpha^{\hat{D}_i}$ denotes the elements for which the subformula $\exists y.y \approx y$ is satisfied, the second and the third components denote the bounds of the interval relative to i for which that element satisfies the formula. Because the subformula $\exists y.y \approx y$ is satisfied by any $a \in \mathbb{N}$, the relation $s_\alpha^{\hat{D}_0}$ contains all tuples (a,j,j') with $a \in \mathbb{N}$ as their first as well as all $j, j' \in \mathbb{N}$ with $0 \leq j \leq j' \leq \ell_0$ as their second and third components. We obtain $p_\alpha^{\hat{D}_0} := \{1,2\}$ by projecting out the first component of the tuples $(a,j) \in r_\alpha^{\hat{D}_0}$ and $(a,0,j') \in s_\alpha^{\hat{D}_0}$ for which the condition $j \leq j'+1$ is satisfied.

We obtain $r_\alpha^{\hat{D}_1}$ from $r_\alpha^{\hat{D}_0}$ by updating the tuples already contained in $r_\alpha^{\hat{D}_0}$ and by possibly adding new tuples to $r_\alpha^{\hat{D}_1}$. Because $\ell_1 = 2$, we must not take any further relations into account and simply update the index component of the tuples that are already contained in $r_\alpha^{\hat{D}_0}$. We thus get $r_\alpha^{\hat{D}_1} := \{(1,2),(2,1)\}$. Similarly, we obtain $s_\alpha^{\hat{D}_1}$ by decreasing the relative indices of the tuples already contained in $s_\alpha^{\hat{D}_0}$ by one. Hence, we have that $s_\alpha^{\hat{D}_1} := \big(\mathbb{N}\times\{0\}\times\{0\}\big) \cup \big(\mathbb{N}\times\{0,1\}\times\{1\}\big) \cup \big(\mathbb{N}\times\{0,1,2\}\times\{2\}\big)$ and $p_\alpha^{\hat{D}_1} := \{1,2\}$.

In addition to updating those tuples already contained in $r_\alpha^{\hat{D}_1}$, to obtain $r_\alpha^{\hat{D}_2}$ we must also take tuples from additional relations into account. In particular, because $\ell_2 = 2$ also the tuples in out^{D_4} must be considered. As $out^{D_4} = \emptyset$, no new elements are added though. As a result, we get $r_\alpha^{\hat{D}_2} := \{(1,1),(2,0)\}$ by decrementing the indices of the tuples stored in $r_\alpha^{\hat{D}_1}$. Moreover, we have $s_\alpha^{\hat{D}_2} := s_\alpha^{\hat{D}_1}$ and thus obtain $p_\alpha^{\hat{D}_2} := \{1,2\}$.

For $i = 3$, we decrease the second component of the tuples in $r_\alpha^{\hat{D}_2}$ to obtain $r_\alpha^{\hat{D}_3} := \{(1,0)\}$. Note that the tuple $(2,0)$ contained in $r_\alpha^{\hat{D}_2}$ is not carried over to $r_\alpha^{\hat{D}_3}$ because from the viewpoint of $i = 3$ the element 2 occurs in the past. Moreover, we have that $s_\alpha^{\hat{D}_3} := s_\alpha^{\hat{D}_2}$ and thus $p_\alpha^{\hat{D}_3} := \{1\}$. Because $3 \in in^{D_3}$ but $3 \notin p_\alpha^{\hat{D}_3}$, the formula $\forall x.in(x) \rightarrow \Diamond_{[0,6)}\, out(x)$ is violated at $i = 3$.

1: $i \leftarrow 0$ *% current index in input sequence* $(D_0, \tau_0), (D_1, \tau_1), \ldots$
2: $q \leftarrow 0$ *% index of next query evaluation in sequence* $(D_0, \tau_0), (D_1, \tau_1), \ldots$
3: $Q \leftarrow \big\{ \big(\alpha, 0, \mathit{waitfor}(\alpha)\big) \,\big|\, \alpha \text{ temporal subformula of } \phi \big\}$
4: **loop**
5: Carry over constants and relations of D_i to $\hat{D}_i$.
6: **for all** $(\alpha, j, \emptyset) \in Q$ **do** *% respect ordering of subformulae*
7: Build relations for α in $\hat{D}_j$ (e.g., build $r_\alpha^{\hat{D}_j}$ and $p_\alpha^{\hat{D}_j}$ if $\alpha = \beta \,\mathcal{S}_I\, \gamma$).
8: Discard auxiliary relations for α in $\hat{D}_{j-1}$ if $j - 1 \geq 0$
(e.g., discard $r_\alpha^{\hat{D}_{j-1}}$ if $\alpha = \beta \,\mathcal{S}_I\, \gamma$).
9: Discard relations $p_\delta^{\hat{D}_j}$, where δ is a temporal subformula of α.
10: **while** all relations $p_\alpha^{\hat{D}_q}$ are built for $\alpha \in \mathit{tsub}(\phi)$ **do**
11: Output valuations violating ϕ at time point q, i.e., output $(\neg\hat{\phi})^{\hat{D}_q}$ and q.
12: Discard structure $\hat{D}_{q-1}$ if $q - 1 \geq 0$.
13: $q \leftarrow q + 1$
14: $Q \leftarrow \big\{ \big(\alpha, i+1, \mathit{waitfor}(\alpha)\big) \,\big|\, \alpha \text{ temporal subformula of } \phi \big\} \cup$
$\big\{ \big(\alpha, j, \bigcup_{\theta \in \mathit{update}(S, \tau_{i+1} - \tau_i)} \mathit{waitfor}(\theta)\big) \,\big|\, (\alpha, j, S) \in Q \text{ and } S \neq \emptyset \big\}$
15: $i \leftarrow i + 1$ *% process next element in input sequence* (D_{i+1}, τ_{i+1})
16: **end loop**

Figure 4.2: Monitor $\mathcal{M}(\phi)$

4.6 Monitoring Algorithm and Correctness

Figure 4.2 presents the monitor $\mathcal{M}(\phi)$. Without loss of generality, it assumes that each temporal subformula occurs only once in ϕ. In the following, we outline its operation.

The monitor uses two counters i and q. The counter i is the index of the current element (D_i, τ_i) in the input sequence $(D_0, \tau_0), (D_1, \tau_1), \ldots$, which is processed sequentially. Initially, i is 0 and it is incremented at the end of each loop iteration (lines 4–16). The counter $q \leq i$ is the index of the next time point q (possibly in the past, from the point of view of i) for which we evaluate $\neg\hat{\phi}$ over the structure $\hat{D}_q$. The evaluation is delayed until the relations $p_\alpha^{\hat{D}_q}$ for $\alpha \in \mathit{tsub}(\phi)$ are all instantiated (lines 10–13). Furthermore, the monitor uses the list[1] Q to ensure that the auxiliary relations of $\hat{D}_0, \hat{D}_1, \ldots$ are built at the right time: if $(\alpha, j, \emptyset)$ is an element of Q at the beginning of a loop iteration, enough time has elapsed to build the relations for the temporal subformula α of the structure $\hat{D}_j$. The monitor initializes Q in line 3. The function $\mathit{waitfor}$, defined below,

[1]We abuse notation by using set notation for lists. Moreover, we assume that Q is ordered in that (α, j, S) occurs before (α', j', S'), whenever α is a proper subformula of α', or $\alpha = \alpha'$ and $j < j'$.

extracts the subformulae that cause a delay of the formula evaluation.

$$waitfor(\theta) := \begin{cases} waitfor(\beta) & \text{if } \theta = \neg\beta,\ \theta = \exists x.\, \beta, \text{ or } \theta = \bullet_I\, \beta, \\ waitfor(\beta) \cup waitfor(\gamma) & \text{if } \theta = \beta \wedge \gamma \text{ or } \theta = \beta\, \mathcal{S}_I\, \gamma, \\ \{\theta\} & \text{if } \theta = \bigcirc_I\, \beta \text{ or } \theta = \beta\, \mathcal{U}_I\, \gamma, \\ \emptyset & \text{otherwise.} \end{cases}$$

The list Q is updated in line 14 before we increment i and start a new loop iteration. For the update, we use the function $update$ that is defined as follows for a formula set U and $\Delta \in \mathbb{N}$:

$$\begin{aligned} update(U, \Delta) := \ & \{\beta \mid \bigcirc_I\, \beta \in U\} \cup \{\beta\, \mathcal{U}_{[\max\{0, c-\Delta\}, d-\Delta)}\, \gamma \mid \beta\, \mathcal{U}_{[c,d)}\, \gamma \in U, \text{ with } d - \Delta > 0\} \\ & \cup \{\beta \mid \beta\, \mathcal{U}_{[c,d)}\, \gamma \in U \text{ or } \gamma\, \mathcal{U}_{[c,d)}\, \beta \in U, \text{ with } d - \Delta \leq 0\} \end{aligned}$$

The update adds a new tuple $(\alpha, i+1, waitfor(\alpha))$ to Q, for each temporal subformula α of ϕ, and it removes the tuples of the form $(\alpha, j, \emptyset)$ from Q. Moreover, for tuples (α, j, S) with $S \neq \emptyset$, the set S is updated using the functions $waitfor$ and $update$ by taking into account the elapsed time to the next time point, i.e. $\tau_{i+1} - \tau_i$.

In lines 6–9, we build the relations for which enough time has elapsed, i.e., the auxiliary relations for α in $\hat{D}_j$ with $(\alpha, j, \emptyset) \in Q$. Since a tuple $(\alpha', j, \emptyset)$ does not occur before a tuple $(\alpha, j, \emptyset)$ in Q, where α is a subformula of α', the relations in $\hat{D}_j$ for α are built before those for α'. To build the relations, we use the incremental constructions described earlier in this section. We thus discard certain relations after we have built the relations for α in $\hat{D}_j$ to reduce space consumption. For instance, if $j > 0$ and $\alpha = \beta \mathcal{S}_I \gamma$, we discard the relation $r_\alpha^{\hat{D}_{j-1}}$, and we discard $r_\alpha^{\hat{D}_{j-1}}$ and $s_\alpha^{\hat{D}_{j-1}}$ when $\alpha = \beta \mathcal{U}_I \gamma$.

In lines 10–13, the valuations violating ϕ at time point q are output together with q, for all q where the relations $p_\alpha^{\hat{D}_q}$ of all immediate temporal subformulae α of ϕ have been built. After an output, the remainder of the extended structure $\hat{D}_{q-1}$ is discarded and q is incremented by 1.

Theorem 4.6.1. *The monitor $\mathcal{M}(\phi)$ from Figure 4.2 has the following properties:*

(i) Whenever $\mathcal{M}(\phi)$ outputs $(\neg\hat{\phi})^{\hat{D}_q}$, then $(\neg\hat{\phi})^{\hat{D}_q} = (\neg\phi)^{(D,\tau,q)}$. Furthermore, the set $(\neg\hat{\phi})^{\hat{D}_q}$ is effectively constructable and finitely representable.

(ii) For every $n \in \mathbb{N}$, $\mathcal{M}(\phi)$ eventually sets the counter q to n in some loop iteration.

Proof. For the proof of Theorem 4.6.1, we index the program variable Q of the monitor $\mathcal{M}(\phi)$ from Figure 4.2 by the counter i. That means, Q_i denotes the list when we enter (line 4) the $(i+1)$st loop iteration. Analogously, we index the program variable q with i. Thus, q_i is the value when the enter the $(i+1)$st loop iteration.

In the following, assume that $(D_0, \tau_0), (D_1, \tau_1), \ldots$ is the input sequence of the monitor $\mathcal{M}(\phi)$ and $\phi \in$ MFOTL is a bounded input formula. Moreover, let T be the set of temporal subformulae of ϕ.

We start with some observations about the monitoring algorithm $\mathcal{M}(\phi)$. Let $\alpha \in T$ and $i, j \in \mathbb{N}$.

(1) If $(\alpha, j, S), (\alpha, j, S') \in Q_i$, then we have that $S = S'$. This follows immediately from the initialization (line 3) and the update (line 14) of the list Q.
(2) We have that $(\alpha, i, \mathit{waitfor}(\alpha)) \in Q_i$. This follows directly from the initialization of Q (line 3) and the update of Q (line 14).
(3) If $(\alpha, j, S) \in Q_i$, then there is an integer $i' \geq i$ such that $(\alpha, j, \emptyset) \in Q_{i'}$. This follows from the update of Q (line 14) (in particular, from the application of the functions $\mathit{waitfor}$ and update), and because the sequence of time stamps τ is monotonically increasing and makes progress. Note that we only remove a tuple (α, j, S) from Q_i if $S = \emptyset$ and after the relations for α in $\hat{D}_j$ have been built (lines 7 and 14).

From (2) and (3), it follows that for every $\alpha \in T$ and $j \in \mathbb{N}$, we eventually execute line 7, where we build the relations for α in the structure $\hat{D}_j$. From (1), it follows that line 7 is executed at most once for $\alpha \in T$ and $j \in \mathbb{N}$. It follows that for each $q \in \mathbb{N}$, we execute line 9 exactly once in a run of $\mathcal{M}(\phi)$.

Furthermore, observe that the relations $p_\alpha^{\hat{D}_j}$, for $\alpha \in \mathit{tsub}(\phi)$ are only discarded at line 12 of the monitoring algorithm. We conclude that for every value of the counter q, the condition of the while loop (line 10) will eventually become true in some loop iteration. Hence, the counter q will always eventually be increased by 1. We conclude that the second property (ii) of the theorem holds.

We now turn to the property (i) of the theorem. We need the following definition of the *temporal rank* of a formula θ:

$$\mathit{rank}(\theta) := \begin{cases} \mathit{rank}(\theta') & \text{if } \theta = \neg\theta' \text{ or } \theta = \exists x.\, \theta', \\ \max\{\mathit{rank}(\theta'), \mathit{rank}(\theta'')\} & \text{if } \theta = \theta' \wedge \theta'', \\ 1 + \mathit{rank}(\theta') & \text{if } \theta = \bullet_I\, \theta' \text{ or } \theta = \circ_I\, \theta' \\ 1 + \max\{\mathit{rank}(\theta'), \mathit{rank}(\theta'')\} & \text{if } \theta = \theta'\, \mathcal{S}_I\, \theta'' \text{ or } \theta = \theta'\, \mathcal{U}_I\, \theta'', \\ 0 & \text{otherwise.} \end{cases}$$

For the remainder of the proof, let us assume that $(\alpha, j, \emptyset) \in Q_i$. The fact that $q_i \leq j$ holds at the beginning of the $(i+1)$st loop iteration is easily established by an induction over i. In the following, we prove by induction over $\mathit{rank}(\alpha)$ that for the construction of the relations of α in $\hat{D}_j$, the necessary relations (according to the incremental constructions given in section 4.4) have been built earlier and have not yet been discarded. From the lemmas in section 4.4 about these constructions, it follows that $p_\alpha^{\hat{D}_j} = p_\alpha^{(D,\tau,j)}$ and $p_\alpha^{\hat{D}_j}$ is regular. From this, we then conclude that the monitor $\mathcal{M}(\phi)$ has the property (i) of the theorem.

Base Case: $\mathit{rank}(\alpha) = 1$. We make a case split on α's main connective.

- $\alpha = \bullet_I\, \beta$: We have that $\mathit{tsub}(\beta) = \emptyset$. Hence, the construction of the relation $p_\alpha^{\hat{D}_j}$ requires no auxiliary relations. Moreover, if $j > 0$, the atomic relations $r^{\hat{D}_{j-1}}$ with $r \in \mathsf{R}$ have not been discarded. This follows from the fact that $q_i \leq j$ and that a relation $r^{\hat{D}_{j-1}}$ is only discarded in line 12. Observe that in line 12, we discard $\hat{D}_{q-1}$

and not $\hat{D}_q$.

- $\alpha = \beta\, \mathcal{S}_I\, \gamma$: We have that $tsub(\beta) \cup tsub(\gamma) = \emptyset$. Similarly to the above case, the atomic relations $r^{\hat{D}_j}$ for $r \in \mathsf{R}$ have not been discarded. Moreover, for $j > 0$ the auxiliary relation $r_\alpha^{\hat{D}_{j-1}}$ has been built earlier and has not been discarded. The fact that it has been built earlier follows from the ordering of the tuples in the list Q and the fact that there is an $i' \leq i$ such that $(\alpha, j-1, \emptyset) \in Q_{i'}$. The latter fact easily follows from an induction over i by using the initialization and the updates of the list Q.
- $\alpha = \bigcirc_I\, \beta$: Since $tsub(\beta) = \emptyset$, we only have to check that a relation $r^{\hat{D}_{j+1}}$ with $r \in \mathsf{R}$ is available. Because of the initialization and the update of the list Q, we have that $j+1 = i$ and $(\alpha, j, \emptyset) \in Q_{j+1}$. Thus, the relation $r^{\hat{D}_{j+1}}$ is available.
- $\alpha = \beta\, \mathcal{U}_I\, \gamma$: Let I be the interval $[c, d)$. Since $tsub(\beta) \cup tsub(\gamma) = \emptyset$, it suffices to check that for all $r \in \mathsf{R}$ and $k < \max\{k' \in \mathbb{N} \mid \tau_{j+k'} - \tau_j < d\}$, the relation $r^{\hat{D}_{j+k}}$ is available. This follows from the initialization and the updates of the list Q. As in the case for the since operator, we conclude that the relations $r_\alpha^{\hat{D}_{j-1}}$ and $s_\alpha^{\hat{D}_{j-1}}$ are available.

Step Case: $rank(\alpha) > 1$. We make again a case split on the main connective of α.

- $\alpha = \bullet_I\, \beta$: For $j = 0$, there is nothing to prove since $p_\alpha^{\hat{D}_j} = \emptyset$. Let $j > 0$. As in the corresponding case of the base case, we have that the relations $r^{\hat{D}_{j-1}}$ with $r \in \mathsf{R}$ are available.
  Let $\delta \in tsub(\beta)$. There is an $i' \leq i$ such that $(\delta, j-1, \emptyset) \in Q_{i'}$. This fact easily follows from an induction over i by using the initialization and the updates of the list Q. Due to the ordering of the list Q and the induction hypothesis, we have that $p_\delta^{\hat{D}_{j-1}}$ has been built earlier. It has not yet been discarded because this only happens in line 9 or line 12, i.e., after the relation $p_\alpha^{D_j}$ has been built. Note that $q_i \leq j$.
- $\alpha = \beta\, \mathcal{S}_I\, \gamma$: This case uses a similar argumentation as the corresponding case of the base case for the relations $r^{\hat{D}_j}$ with $r \in \mathsf{R}$ and $r_\alpha^{\hat{D}_{j-1}}$. For the relations $p_\delta^{\hat{D}_j}$ with $\delta \in tsub(\beta) \cup tsub(\gamma)$, we use a similar argumentation as in the case for the previous operator.
- $\alpha = \bigcirc_I\, \beta$: As in the corresponding case of the base case, the relations $r^{\hat{D}_{j+1}}$ with $r \in \mathsf{R}$ are available. Let $\delta \in tsub(\beta)$. There is an $i' \leq i$ such that $(\delta, j+1, \emptyset) \in Q_{i'}$. This fact easily follows from an induction over i by using the induction hypothesis on δ and by using the initialization and the updates of the list Q.
- $\alpha = \beta\, \mathcal{U}_I\, \gamma$: Let I be the interval $[c, d)$ and let $\ell_j := \max\{k' \in \mathbb{N} \mid \tau_{j+k'} - \tau_j < d\}$. Recall that ℓ_j is the lookahead offset at time point j. As in the corresponding case of the base case, the relations $r^{\hat{D}_{j+k}}$ with $r \in \mathsf{R}$ and $k \leq \ell_j$ are available. Moreover, when $j > 0$ we conclude as in the base case that the relations $r_\alpha^{\hat{D}_{j-1}}$ and $s_\alpha^{\hat{D}_{j-1}}$ are available.
  Let $\delta \in tsub(\beta) \cup tsub(\gamma)$. By the induction hypothesis, we have that the relations

when building the relations for δ of $\hat{D}_{j+k}$ with $k \leq \ell_j$ are available. It suffices to show that for all $k \leq \ell_j$, there is an $i' \leq i$ such that $(\delta, j+k, \emptyset) \in Q_{i'}$. This follows easily from the initialization and the updates of the list Q. ⊣

4.7 Summary

In this chapter, we have presented an automata-based monitoring approach for an expressive safety fragment of metric first-order temporal logic. The use of automatic structures substantially generalizes both the kinds of structures and the class of formulae that can be monitored. Specifically, by representing possibly infinite relations as finite state automata it allows for the arbitrary use of negation and quantification in property specifications. This eliminates the limitations that arise in databases, where relations must be finite and thus negation and quantification can only be used in a restricted manner (see the next chapter for more details). Moreover, independent of the use of automatic structures, our approach allows the arbitrary nesting of past and bounded future operators. The availability of both past and bounded future operators results in compact and natural property specifications. Note that it is unknown whether the past-only fragment of MFOTL is as expressive as the fragment with both past and bounded future operators and whether formulae in the past-only fragment can be expressed as succinctly as those in the future-bounded fragment.

In contrast to runtime monitoring approaches that provide only temporal past operators (e.g., [Cho95]), bounded formulae cannot always be evaluated instantly when new state or event information becomes available. To guarantee a correct semantics, our approach ensures algorithmically that formulae are only evaluated when they are well-defined.

"Our knowledge can only be finite, while our ignorance must necessarily be infinite"

Karl Popper

Chapter 5

Monitoring with Finite Relations

In this chapter, we consider an important special case of our runtime monitoring approach, namely the case, where all relations are finite. In this setting, we can represent the relations directly as tables and build on work from the area of relational databases [Cod70, AHV95] for implementing the monitor $\mathcal{M}(\phi)$. Specifically, we can leverage efficient database operations to implement the incremental constructions described in Section 4.4. The direct representation of relations as tables also makes possible several types of optimizations. Furthermore, if all relations are finite, we can show that the space consumed by our monitor is polynomially bounded by the cardinality of the set of data constants appearing in the processed prefix.

When representing relations as finite tables, however, we inherit standard problems from database theory. Because negation and quantification can result in infinite relations, their use in monitored formulae must be restricted. As a result, not all formulae from our fragment can be handled by our runtime monitor for finite relations. To ensure that only formulae which always result in finite relations are admitted for monitoring, we thus provide solutions that build upon and extend similar work from the area of temporal databases [CTB01, Cho95, CT95].

This chapter is structured as follows. In Section 5.1, we explain the difficulties associated with the requirement that relations be always finite and sketch our solution. In Section 5.2, we formally define the notion of temporal subformula domain independence as a semantic characterization of those bounded MFOTL formulae that can be handled by our monitor for finite relations. In Section 5.3, we then describe our solution to ensure that only bounded temporal subformula domain independent MFOTL formulae are admitted for monitoring. Several types of space optimizations are described in Section 5.4. Finally, we analyze the space requirements of our runtime monitor in Section 5.5 and conclude the chapter with a discussion in Section 5.6.

5.1 Working with Finite Relations

Observe that our constructions from Section 4.4 do not work when the auxiliary relations are required to be finite. In particular, Lemma 4.4.2, Lemma 4.4.4, Lemma 4.4.7, and Lemma 4.4.11 become invalid when replacing "regular" by "finite." The constructed relations are still regular but possibly infinite. The problem of infinite relations manifests itself in two related cases where finite relations are complemented. In the following, we illustrate these cases and sketch solutions for our monitoring method. Technical details are presented in Sections 5.2 and 5.3.

5.1.1 Derived Temporal Operators

To illustrate the first problem case, consider the formula $\blacksquare_I\, r(x)$, which abbreviates $\neg(\mathit{true}\ \mathcal{S}_I\ \neg r(x))$. Recall from Chapter 4 that at each time point we build a relation containing the tuples that satisfy $\mathit{true}\ \mathcal{S}_I\ \neg r(x)$. But when r corresponds to a finite relation at each time point, its complement will be infinite. We can solve this problem by using the dual temporal operators $\mathcal{R}_I$ (release) and $\mathcal{T}_I$ (trigger), which are defined as $\beta\ \mathcal{R}_I\ \gamma\ :=\ \neg(\neg\beta\ \mathcal{U}_I\ \neg\gamma)$ and $\beta\ \mathcal{T}_I\ \gamma\ :=\ \neg(\neg\beta\ \mathcal{S}_I\ \neg\gamma)$. For these two operators, we can define incremental update constructions similar to those for $\mathcal{S}_I$ and $\mathcal{U}_I$ given in Section 4.4. Instead of $\mathcal{S}_I$, we can then use $\mathcal{T}_I$ to handle $\blacksquare_I\, r(x)$ without negation, since $\blacksquare_I\, r(x)$ and $\mathit{false}\ \mathcal{T}_I\ r(x)$ are logically equivalent.

5.1.2 Negation

To illustrate the second problem case, consider the formula $p(x) \wedge \blacklozenge_I\, \neg q(x)$. The subformula $\blacklozenge_I\, \neg q(x)$ is problematic because at each time point we store the elements that satisfy $\blacklozenge_I\, \neg q(x)$ in an auxiliary relation. This relation is infinite when the relations for q are finite. However, the formula in its entirety is unproblematic since at each time point an element that satisfies $\blacklozenge_I\, \neg q(x)$ must also be in the relation for the predicate p, which is finite. To handle negation here, we build on work from database theory on domain independence (e.g., [Fag82]), where a similar problem arises with queries containing negation and quantification (see, e.g., [Ull88] or [GT91]). A standard solution tries to rewrite queries so that the quantified variables range only over finitely many elements (see [AHV95]). We generalize this solution for first-order queries to bounded MFOTL formulae in that we try to rewrite a given MFOTL formula ϕ such that the subformulae of all temporal operators have only finitely many satisfying valuations. For instance, we rewrite $p(x) \wedge \blacklozenge_I\, \neg q(x)$ to the logically equivalent formula $p(x) \wedge \blacklozenge_I \big(\neg q(x) \wedge \Diamond_I\, p(x)\big)$. Note that each temporal subformula in this transformed formula has only finitely many satisfying valuations. Hence, the transformed formula can be handled by our monitoring algorithm if the interval I is finite. Recall the requirement that temporal future operators be bounded.

After rewriting the formula ϕ, we check, based on the syntax of the result ψ, if each $\theta \in \{\alpha \,|\, \alpha = \psi, \alpha$ is a temporal subformula of ψ, or α is a direct subformula of a temporal subformula of $\psi\}$ has only finitely many satisfying valuations. If ψ passes this check, we know that ψ can be handled by our monitor for finite relations. Otherwise, no conclusions can be drawn.

5.2 Characterization of Monitorable Formulae

In a setting where all relations are finite, system executions are represented as timed temporal databases, i.e., timed temporal structures over some signature $S = (\mathsf{C}, \mathsf{R}, a)$, where all relations r^{D_i} are finite, for each $r \in \mathsf{R}$ and $i \in \mathbb{N}$. Note that the infinite relations for $\approx$ and $\prec$ do not change over time. As usual, they are considered built-in and do not belong to the processed structures.

We now formally define the class of MFOTL formulae that can be handled by our monitoring approach in this setting. Throughout this section, let $S = (\mathsf{C}, \mathsf{R}, a)$ be a signature and v_0 a valuation.

5.2.1 Auxiliary Definitions

For technical reasons to become clear later, we first also provide strict versions of the binary temporal operators $\mathcal{S}_I$ and $\mathcal{U}_I$. Namely, we define the MFOTL$^+$ formulae by extending Definition 2.4.16 to also include the strict operators $\dot{\mathcal{U}}_I$ and $\dot{\mathcal{S}}_I$. Note that in the metric case, the strict and the non-strict versions of the operators since and until cannot be derived from each other.

The semantics of $\dot{\mathcal{U}}_I$ and $\dot{\mathcal{S}}_I$ is defined as follows.

Definition 5.2.1. *Let (D, τ) be a timed temporal structure over a signature S, with $D = (D_0, D_1, \dots)$ and $\tau = (\tau_0, \tau_1, \dots)$, β and γ MFOTL$^+$ formulae over S, v a valuation, and $i \in \mathbb{N}$. In addition to Definition 2.4.17, we define*

$$(D, \tau, v, i) \models (\beta \,\dot{\mathcal{S}}_I\, \gamma) \;\; \textit{iff for some } j < i,\ \tau_i - \tau_j \in I,\ (D, \tau, v, j) \models \gamma, \textit{ and } (D, \tau, v, k) \models \beta, \textit{ for all } k \in [j+1, i+1)$$

and

$$(D, \tau, v, i) \models (\beta \,\dot{\mathcal{U}}_I\, \gamma) \;\; \textit{iff for some } j > i,\ \tau_j - \tau_i \in I,\ (D, \tau, v, j) \models \gamma, \textit{ and } (D, \tau, v, k) \models \beta, \textit{ for all } k \in [i, j)\,.$$

We also define the strict versions of the derived operators $\Box_I$, $\Diamond_I$, $\blacksquare_I$, $\blacklozenge_I$, $\mathcal{R}_I$, and $\mathcal{T}_I$, which are derived from $\dot{\mathcal{S}}_I$ and $\dot{\mathcal{U}}_I$ analogously to their non-strict counterparts. We decorate the strict operators with a dot to distinguish them from the non-strict versions. For example, we write $\dot{\blacksquare}_I$ for the strict version of $\blacksquare_I$. Moreover, we denote as MFOTL$^+$ the set of formulae obtained from Definition 2.4.17 and the operators $\dot{\mathcal{S}}_I$

and $\dot{\mathcal{U}}_I$, where the (strict) derived temporal operators $\dot{\Diamond}_I, \dot{\blacklozenge}_I, \dot{\Box}_I, \dot{\blacksquare}_I, \dot{\mathcal{R}}_I$, and $\dot{\mathcal{T}}_I$, the (non-strict) derived temporal operators $\Diamond_I, \blacklozenge_I, \Box_I, \blacksquare_I, \mathcal{R}_I$ and $\mathcal{T}_I$, the derived Boolean connectives $\vee, \rightarrow$, and $\leftrightarrow$, and the universal quantifier $\forall$ are all treated as primitives.

The following function determines the set of direct subformulae of a temporal formula. For a formula $\alpha \in \mathsf{MFOTL}^+$, we define

$$dstf(\alpha) := \begin{cases} \{\beta\} & \text{if } \alpha = \otimes\beta, \text{ where } \otimes \text{ is an unary temporal operator,} \\ \{\beta, \gamma\} & \text{if } \alpha = \beta \oplus \gamma, \text{ where } \oplus \text{ is a binary temporal operator,} \\ \emptyset & \text{otherwise.} \end{cases}$$

We say that $i \in \mathbb{N}$ is *minimal* for $q \in \mathbb{N}$, the timed temporal database (D, τ) with $D = (D_0, D_1, \dots)$, and the bounded MFOTL$^+$ formula θ if for all valuations v, we have that

$$(D, \tau, v, q) \models \theta \qquad \text{iff} \qquad (D', \tau, v, q) \models \theta\,,$$

for all $D' = (D'_0, D'_1, \dots)$ with $D'_j = D_j$, for all $j \leq i$. Since, at any time point, θ can only refer to time points finitely far into the future and τ makes progress, there is always a minimal i, for every given q, (D, τ), and θ.

The *active domain* of a finite prefix $\bar{D} = (D_0, D_1, \dots, D_i)$ of D is

$$adom(\bar{D}) := \{c^{D_0} \mid c \in \mathsf{C}\} \cup \bigcup_{0 \leq k \leq i} \bigcup_{r \in \mathsf{R}} \{d_j \mid (d_1, \dots, d_{a(r)}) \in r^{D_k} \text{ and } 1 \leq j \leq a(r)\}\,.$$

5.2.2 Temporal Domain Independence

The following definition characterizes those MFOTL$^+$ formulae that can be monitored by our monitor for finite relations. We write $\models_{\mathsf{U}}$ to denote the relation $\models$, as defined in Definition 2.4.17, but quantification is relativized to the set $\mathsf{U} \subseteq |D|$. In the following, let v_0 be an arbitrary valuation.

Definition 5.2.2. *Let θ be a bounded* MFOTL$^+$ *formula with the free variables given by the vector $\bar{x} = (x_1, \dots, x_n)$. Moreover, let T be the set of temporal subformulae of θ.*

(i) The formula θ is temporal domain independent *if for all timed temporal databases (D, τ), all $i, q \in \mathbb{N}$ with $q \leq i$, and all $\mathsf{U}, \mathsf{U}' \subseteq |D|$, we have that*

$$\left\{\bar{d} \in \mathsf{U}^n \,\middle|\, (D, \tau, v_0[\bar{x}/\bar{d}], q) \models_{\mathsf{U}} \theta\right\} = \left\{\bar{d} \in \mathsf{U}'^n \,\middle|\, (D, \tau, v_0[\bar{x}/\bar{d}], q) \models_{\mathsf{U}'} \theta\right\},$$

whenever $adom(\bar{D}) \subseteq \mathsf{U}, \mathsf{U}'$ with $\bar{D} = (D_1, \dots, D_q, \dots, D_i)$ and i is minimal for q, (D, τ), and θ.

(ii) The formula θ is temporal subformula domain independent *(*TSF *domain independent, for short) if (1) is θ temporal domain independent, (2) each $\alpha \in \mathsf{T}$ is temporal domain independent, and (3) each $\delta \in dstf(\alpha)$ is temporal subformula independent, for all $\alpha \in \mathsf{T} \setminus \{\neg\beta(\bar{x})\, \mathcal{S}_I\, \gamma(\bar{x}), \neg\beta(\bar{x})\dot{\mathcal{S}}_I\gamma(\bar{x})\}$.*

The notion of temporal domain independence is a natural generalization of the standard notion of domain independence (see [AHV95]). Bounded temporal domain independent formulae have properties similar to first-order domain independent formulae. For instance, the set $\theta^{(D,\tau,i)}$ is finite, for a bounded temporal domain independent formula θ, a temporal database (D,τ), and $i \in \mathbb{N}$. Moreover, the set $\theta^{(D,\tau,i)}$ contains only data tuples whose elements are constants or which appear in a finite prefix of (D,τ). The length of the prefix depends on i, τ, and θ.

Lemma 5.2.3. *Let θ be a bounded* MFOTL$^+$ *formula, (D,τ) a timed temporal database, and $q \in \mathbb{N}$. If θ is temporal domain independent, then $\theta^{(D,\tau,q)}$ is finite.*

Proof. Let $i \in \mathbb{N}$ be minimal for q, (D,τ), and θ. Let $\bar{D} = (D_0, \ldots, D_q, \ldots, D_i)$. We have that

$$\left\{\bar{d} \in |D|^n \,\middle|\, (D,\tau,v_0[\bar{x}/\bar{d}],q) \models \theta\right\} = \left\{\bar{d} \in adom(\bar{D})^n \,\middle|\, (D,\tau,v_0[\bar{x}/\bar{d}],q) \models_{adom(\bar{D})} \theta\right\}.$$

Since $adom(\bar{D})$ is finite, we are done. ⊣

5.2.3 Discussion

Recall from Chapter 4 that our monitor construction is based on the incremental updating of auxiliary relations. For a bounded MFOTL$^+$ formula ϕ to be monitorable by our runtime monitor in the setting with only finite relations, ϕ must be TSF domain independent. By Lemma 5.2.3, this ensures that at each time point the sets of satisfying valuations of ϕ and all the auxiliary relations introduced for its temporal subformulae are finite. Observe that it is not enough for ϕ to be temporal domain independent because the auxiliary relation(s) for each temporal subformula are updated individually.

Example 5.2.4 (Running example)**.** *Consider again our running example. As shown in Example 2.4.20, our example property can be defined by the* MFOTL *formula*

$$\Box\, \forall t. \forall c. trans(t,c) \wedge \left(\blacklozenge_{[0,31)}\, \exists t'. trans(t',c) \wedge \Diamond_{[0,3)}\, report(t',c)\right) \rightarrow \Diamond_{[0,3)}\, report(t,c).$$

Removing syntactic sugar yields the MFOTL *formula*

$$\Box\, \neg\big(\exists t. \exists c. trans(t,c) \wedge \left(\blacklozenge_{[0,31)}\, \exists t'. trans(t',c) \wedge \Diamond_{[0,3)}\, report(t',c)\right) \wedge \neg(\Diamond_{[0,3)}\, report(t,c))\big)$$

from which it is not difficult to see that our example property is TSF *domain independent. Hence, it can be handled by our monitor for finite relations.*

5.3 Syntactical Approximation based on Rewriting

For an arbitrary MFOTL formula ϕ, it undecidable whether is it is temporal domain independent [Pao69, Cho95]. In the following, we thus present a procedure based on

rewriting that identifies a subset of the class of TSF domain independent formulae, which we call the class of TSF *safe-range formulae*. Our procedure extends a similar procedure for non-metric first-order temporal logic [CTB01] in two respects. First, we extend the procedure to also handle metric temporal operators. Second, we improve the procedure to recognize a larger set of formulae by defining additional rewrite rules for the strict dual temporal operators $\dot{\mathcal{R}}_I$ and $\dot{\mathcal{T}}_I$, with $I \in \mathbb{I}$.

For the remainder of this chapter, let $S = (\mathsf{C}, \mathsf{R}, a)$ be a signature and V denote a countably infinite set of variables, where we assume $\mathsf{V} \cap \mathsf{R} = \emptyset$ and $\mathsf{V} \cap \mathsf{C} = \emptyset$. Moreover, let (D, τ) be a temporal database over S, where we assume that $|D| = \mathbb{N}$ and $<$ is the standard ordering on $\mathbb{N}$.

5.3.1 Overview

In the following, we outline the individual steps to check whether a formula $\phi \in \mathsf{MFOTL}^+$ can be monitored with our approach.

1. *Normalization.* In the first step, we transform ϕ into a logically equivalent formula $\phi' \in \mathsf{MFOTL}^+$, where all quantified variables have unique names, syntactic sugar is unfolded, double negations are removed, and negations are pushed towards the leaves of the subformulae by applying rewrite rules.
2. *Safe-range check.* We then check if the sets of free and range-restricted variables in ϕ' coincide and if for all subformulae of the form $\exists x.\, \psi$, where the variable x occurs free in ψ, it holds that x is range-restricted in ψ. If this is the case, ϕ' is safe-range and we proceed to step 3. Otherwise, ϕ' is rejected.
3. *Propagation of range restrictions.* Next, we transform ϕ' into a logically equivalent formula $\phi'' \in \mathsf{MFOTL}^+$ by applying rewrite rules such that all range restrictions are propagated towards the subformulae of ϕ'.
4. TSF *safe-range check.* In the final step, we check if all temporal subformulae of ϕ'', and all their direct subformulae are safe-range. If this is the case, ϕ'' is TSF *safe-range*. Otherwise, ϕ'' is not TSF safe-range and is rejected. Finally, we check that all temporal future operators in ϕ'' are bounded. If this is also the case, the formula ϕ'' can be monitored.

The subsequent sections describe these steps in detail.

5.3.2 Normalization

In the first step, the input formula $\phi \in \mathsf{MFOTL}^+$ is transformed into the logically equivalent formula $sr(\phi)$ as defined in Definition 5.3.1. In the following, let $\mathsf{FV}(\theta)$ denote the set of free variables of a formula $\theta \in \mathsf{MFOTL}^+$.

Definition 5.3.1. *We denote as $sr(\phi)$ the formula obtained from $\phi \in \mathsf{MFOTL}^+$*
(1) by renaming the quantified variables using unique names;

(2) by replacing the occurrences of subformulae of the form $\forall x.\,\beta$, $\beta \rightarrow \gamma$, *and* $\beta \leftrightarrow \gamma$ *by* $\neg\exists x.\,\neg\alpha$, $\neg\alpha \vee \beta$, *and* $(\neg\alpha \vee \beta) \wedge (\neg\beta \vee \alpha)$, *respectively;*

(3) by pushing down negations and removing double negations by applying the following rules as long as possible:

(a) $\neg\neg\alpha \mapsto \alpha$

(b) $\exists x.\alpha \mapsto \alpha$, *if* $x \notin \mathsf{FV}(\alpha)$

(c) $\neg(\alpha \vee \beta) \mapsto \neg\alpha \wedge \neg\beta$ *and* $\neg(\alpha \wedge \beta) \mapsto \neg\alpha \vee \neg\beta$

(d) $\neg(\bigcirc_I \alpha) \mapsto \bigcirc_I \neg\alpha$

(e) $\neg(\dot{\Diamond}_I\alpha) \mapsto \dot{\Box}_I\neg\alpha$, $\neg(\dot{\Box}_I\alpha) \mapsto \dot{\Diamond}_I\neg\alpha$, $\neg(\dot{\blacklozenge}_I\alpha) \mapsto \dot{\blacksquare}_I\neg\alpha$, *and* $\neg(\dot{\blacksquare}_I\alpha) \mapsto \dot{\blacklozenge}_I\neg\alpha$

(f) $\neg(\Diamond_I\alpha) \mapsto \Box_I\neg\alpha$, $\neg(\Box_I\alpha) \mapsto \Diamond_I\neg\alpha$, $\neg(\blacklozenge_I\alpha) \mapsto \blacksquare_I\neg\alpha$, *and* $\neg(\blacksquare_I\alpha) \mapsto \blacklozenge_I\neg\alpha$

(g) $\neg(\beta\dot{\mathcal{S}}_I\gamma) \mapsto \neg\beta\dot{\mathcal{T}}_I\neg\gamma$ *and* $\neg(\beta\dot{\mathcal{U}}_I\gamma) \mapsto \neg\beta\dot{\mathcal{R}}_I\neg\gamma$

(h) $\neg(\beta\dot{\mathcal{T}}_I\gamma) \mapsto \neg\beta\dot{\mathcal{S}}_I\neg\gamma$ *and* $\neg(\beta\dot{\mathcal{R}}_I\gamma) \mapsto \neg\beta\dot{\mathcal{U}}_I\neg\gamma$

(i) $\neg(\beta\,\mathcal{S}_I\,\gamma) \mapsto \neg\beta\,\mathcal{T}_I\,\neg\gamma$ *and* $\neg(\beta\,\mathcal{U}_I\,\gamma) \mapsto \neg\beta\,\mathcal{R}_I\,\neg\gamma$

(j) $\neg(\beta\,\mathcal{T}_I\,\gamma) \mapsto \neg\beta\,\mathcal{S}_I\,\neg\gamma$ *and* $\neg(\beta\,\mathcal{R}_I\,\gamma) \mapsto \neg\beta\,\mathcal{U}_I\,\neg\gamma$

Because at most one rule can be applied after each rewrite step, the rewrite rules are confluent. Moreover, It is easy to see that $sr(\phi)$ is logically equivalent to ϕ, since every step preserves logical equivalence. Observe that the rewrite rules are terminating because the rules (c)–(j) push negation inwards, (a) eliminates double negation, and (b) eliminates unnecessary quantifiers. Finally, note that the Boolean connective $\neg$ cannot be moved over an existential quantifier $\exists$.

Lemma 5.3.2. *Let* $\phi \in \mathsf{MFOTL}^+$ *be a formula,* v *a valuation, and* $i \in \mathbb{N}$. *It holds that* $(D, \tau, v, i) \models \phi$ *iff* $(D, \tau, v, i) \models sr(\phi)$.

5.3.3 Safe-range Check

Definition 5.3.3. *For* $\alpha \in \mathsf{MFOTL}^+$, *we define* $\mathsf{RR}(\alpha)$ *as follows, where* x, x' *range over* V *and* c *over* C.

$$\mathsf{RR}(\alpha) := \begin{cases} \{x\} & \text{if } \alpha = x \approx c,\ \alpha = c \approx x, \text{or } \alpha = x \prec c, \\ \{t_i \mid t_i \in \mathsf{V} \wedge 1 \leq i \leq a(r)\} & \text{if } \alpha = r(t_1, \ldots, t_{a(r)}), \\ \emptyset & \text{if } \alpha = \neg\beta, \alpha = c \prec x, \alpha = x \approx x', \\ & \quad \alpha = x \prec x', \text{or } \alpha = x \prec c, \\ \mathsf{RR}(\beta) \setminus \{x\} & \text{if } \alpha = \exists x.\,\beta \text{ and } x \in \mathsf{RR}(\beta), \\ \mathsf{RR}(\beta) & \text{if } \alpha = \otimes\beta \text{ with } \otimes \in \{\bullet_I, \dot{\blacklozenge}_I, \blacklozenge_I, \dot{\blacksquare}_I, \blacksquare_I\} \\ & \qquad \cup \{\bigcirc_I, \dot{\Diamond}_I, \Diamond_I, \dot{\Box}_I, \Box_I\}, \\ \mathsf{RR}(\beta) \cup \mathsf{RR}(\gamma) & \text{if } \alpha = \beta \wedge \gamma,\ \alpha = \beta\dot{\mathcal{S}}_I\gamma,\ \text{or } \alpha = \beta\dot{\mathcal{U}}_I\gamma, \\ \mathsf{RR}(\beta) \cap \mathsf{RR}(\gamma) & \text{if } \alpha = \beta \vee \gamma, \alpha = \beta\dot{\mathcal{T}}_I\gamma, \text{or } \alpha = \beta\dot{\mathcal{R}}_I\gamma, \\ \mathsf{RR}(\gamma) & \text{if } \alpha = \beta\,\mathcal{S}_I\,\gamma, \alpha = \beta\,\mathcal{U}_I\,\gamma, \alpha = \beta\,\mathcal{T}_I\,\gamma, \\ & \quad \text{or } \alpha = \beta\,\mathcal{R}_I\,\gamma. \end{cases}$$

Note that RR is only applied to formulae $sr(\phi)$, where $\phi \in \mathsf{MFOTL}^+$. In particular, this means that universal quantifiers and Boolean connectives like $\rightarrow$ have been replaced according to Definition 5.3.1.

Definition 5.3.4. *The formula α is called* safe-range *if $\mathsf{RR}(\alpha) = \mathsf{FV}(\alpha)$ and for every subformula of α of the form $\exists x.\beta$, it holds that $x \in \mathsf{FV}(\beta)$ implies $x \in \mathsf{RR}(\beta)$.*

Because of Lemma 5.3.5 below, we do not check whether ϕ is safe-range but we check whether the normalized formula $sr(\phi)$ is safe-range.

Lemma 5.3.5. *Let $\phi \in \mathsf{MFOTL}^+$. If ϕ is safe-range, then $sr(\phi)$ is also safe-range.*

Proof. Follows directly from Definition 2.4.17 and Definition 5.3.1. Note that $\mathsf{RR}(\neg\beta) = \emptyset$ for any $\beta \in \mathsf{MFOTL}^+$. Hence, pulling the negation inwards increases the chances that a formula becomes safe-range. ⊣

Because the non-strict operators $\mathcal{S}_I$, $\mathcal{U}_I$, $\mathcal{T}_I$, and $\mathcal{R}_I$ have less favorable properties in terms of restricting the ranges of free variables, in the following we only present rewrite rules for the strict operators. An extension to the non-strict versions is straightforward.

5.3.4 Propagation of Range Restrictions

We now describe how range restrictions can be propagated towards the subformulae of a safe-range formula. The following logical equivalences allow us to move range-restricting subformulae between the left- and right-hand sides of the $\dot{\mathcal{S}}_I$ and $\dot{\mathcal{U}}_I$ operators and to move a range-restricting formula into the scope of a temporal connective.

Lemma 5.3.6. *Let $\alpha, \beta, \gamma \in \mathsf{MFOTL}^+$. The following logical equivalences hold:*

1. $\alpha \dot{\mathcal{U}}_I \beta \equiv \alpha \dot{\mathcal{U}}_I (\dot{\blacklozenge}_I \alpha \wedge \beta)$
2. $\alpha \dot{\mathcal{U}}_I \beta \equiv (\alpha \wedge \dot{\Diamond}_I \beta) \dot{\mathcal{U}}_I \beta$
3. $\alpha \dot{\mathcal{S}}_I \beta \equiv \alpha \dot{\mathcal{S}}_I (\dot{\Diamond}_I \alpha \wedge \beta)$
4. $\alpha \dot{\mathcal{S}}_I \beta \equiv (\alpha \wedge \dot{\blacklozenge}_I \beta) \dot{\mathcal{S}}_I \beta$
5. $\alpha \wedge (\beta \dot{\mathcal{U}}_I \gamma) \equiv \alpha \wedge (\beta \dot{\mathcal{U}}_I (\dot{\blacklozenge}_I \alpha \wedge \gamma))$
6. $\alpha \wedge (\beta \dot{\mathcal{U}}_I \gamma) \equiv \alpha \wedge ((\beta \wedge \blacklozenge_I \alpha) \dot{\mathcal{U}}_I \gamma)$
7. $\alpha \wedge (\beta \dot{\mathcal{S}}_I \gamma) \equiv \alpha \wedge (\beta \dot{\mathcal{S}}_I (\dot{\Diamond}_I \alpha \wedge \gamma))$
8. $\alpha \wedge (\beta \dot{\mathcal{S}}_I \gamma) \equiv \alpha \wedge ((\beta \wedge \Diamond_I \alpha) \dot{\mathcal{S}}_I \gamma)$

Proof. Follows directly from Definitions 2.4.17 and 5.2.1. ⊣

Definition 5.3.7. *Let ϕ be a safe-range formula. We denote as $ra(\phi)$ the result of applying the rules stated in Figure 5.1 together with standard rules of commutativity and associativity for conjunction, starting from the outermost main connective.*

1.	$\alpha \wedge (\beta \vee \gamma)$	$\mapsto$	$(\alpha \wedge \beta) \vee (\alpha \wedge \gamma)$	(push range restriction into $\vee$)
2.	$\alpha \wedge (\beta \dot{\mathcal{R}}_I \gamma)$	$\mapsto$	$\alpha \wedge ((\beta \wedge \blacklozenge_I \alpha) \dot{\mathcal{R}}_I (\gamma \wedge \blacklozenge_I \alpha)$	(push range restriction into $\dot{\mathcal{R}}_I$)
3.	$\alpha \wedge (\beta \dot{\mathcal{T}}_I \gamma)$	$\mapsto$	$\alpha \wedge ((\beta \wedge \dot{\Diamond}_I \alpha) \dot{\mathcal{T}}_I (\gamma \wedge \dot{\Diamond}_I \alpha)$	(push range restriction into $\dot{\mathcal{T}}_I$)
4.	$\alpha \wedge \exists x.\beta$	$\mapsto$	$\alpha \wedge \exists x.(\alpha \wedge \beta)$	(push range restriction into $\exists$)
5.	$\alpha \wedge \neg\beta$	$\mapsto$	$\alpha \wedge \neg(\alpha \wedge \beta)$	(push range restriction into $\neg$)
6.	$(\alpha \wedge \gamma) \dot{\mathcal{S}}_I \beta$	$\mapsto$	$(\alpha \wedge \gamma) \dot{\mathcal{S}}_I (\dot{\Diamond}_I \alpha \wedge \beta)$	(distribute from left to right in $\dot{\mathcal{S}}_I$)
7.	$(\alpha \wedge \gamma) \dot{\mathcal{U}}_I \beta$	$\mapsto$	$(\alpha \wedge \gamma) \dot{\mathcal{U}}_I (\blacklozenge_I \alpha \wedge \beta)$	(distribute from left to right in $\dot{\mathcal{U}}_I$)
8.	$\beta \dot{\mathcal{S}}_I (\alpha \wedge \gamma)$	$\mapsto$	$(\blacklozenge_I \alpha \wedge \beta) \dot{\mathcal{S}}_I (\alpha \wedge \gamma)$	(distribute from right to left in $\dot{\mathcal{S}}_I$)
9.	$\beta \dot{\mathcal{U}}_I (\alpha \wedge \gamma)$	$\mapsto$	$(\dot{\Diamond}_I \alpha \wedge \beta) \dot{\mathcal{U}}_I (\alpha \wedge \gamma)$	(distribute from right to left in $\dot{\mathcal{U}}_I$)
10.	$\alpha \wedge (\beta \dot{\mathcal{S}}_I \gamma)$	$\mapsto$	$\alpha \wedge ((\Diamond_I \alpha \wedge \beta) \dot{\mathcal{S}}_I \gamma)$	(push into $\dot{\mathcal{S}}_I$, left side)
11.	$\alpha \wedge (\beta \dot{\mathcal{U}}_I \gamma)$	$\mapsto$	$\alpha \wedge ((\blacklozenge_I \alpha \wedge \beta) \dot{\mathcal{U}}_I \gamma)$	(push into $\dot{\mathcal{U}}_I$, left side)
12.	$\alpha \wedge (\gamma \dot{\mathcal{S}}_I \beta)$	$\mapsto$	$\alpha \wedge (\gamma \dot{\mathcal{S}}_I (\dot{\Diamond}_I \alpha \wedge \beta)$	(push into $\dot{\mathcal{S}}_I$, right side)
13.	$\alpha \wedge (\gamma \dot{\mathcal{U}}_I \beta)$	$\mapsto$	$\alpha \wedge (\gamma \dot{\mathcal{U}}_I (\blacklozenge_I \alpha \wedge \beta)$	(push into $\dot{\mathcal{U}}_I$, right side)
14.	$\alpha \wedge \dot{\blacklozenge}_I \beta$	$\mapsto$	$\alpha \wedge \dot{\blacklozenge}_I (\dot{\Diamond}_I \alpha \wedge \beta)$	(push into $\dot{\blacklozenge}_I$)
15.	$\alpha \wedge \dot{\Diamond}_I \beta$	$\mapsto$	$\alpha \wedge \dot{\Diamond}_I (\blacklozenge_I \alpha \wedge \beta)$	(push into $\dot{\Diamond}_I$)
16.	$\alpha \wedge \dot{\blacksquare}_I \beta$	$\mapsto$	$\alpha \wedge \dot{\blacksquare}_I (\dot{\Diamond}_I \alpha \wedge \beta)$	(push into $\dot{\blacksquare}_I$)
17.	$\alpha \wedge \dot{\Box}_I \beta$	$\mapsto$	$\alpha \wedge \dot{\Box}_I (\dot{\blacklozenge}_I \alpha \wedge \beta)$	(push into $\dot{\Box}_I$)
18.	$\alpha \wedge \blacklozenge_I \beta$	$\mapsto$	$\alpha \wedge \blacklozenge_I (\Diamond_I \alpha \wedge \beta)$	(push into $\blacklozenge_I$)
19.	$\alpha \wedge \Diamond_I \beta$	$\mapsto$	$\alpha \wedge \Diamond_I (\blacklozenge_I \alpha \wedge \beta)$	(push into $\Diamond_I$)
20.	$\alpha \wedge \blacksquare_I \beta$	$\mapsto$	$\alpha \wedge \blacksquare_I (\Diamond_I \alpha \wedge \beta)$	(push into $\blacksquare_I$)
21.	$\alpha \wedge \Box_I \beta$	$\mapsto$	$\alpha \wedge \Box_I (\blacklozenge_I \alpha \wedge \beta)$	(push into $\Box_I$)
22.	$\alpha \wedge \bullet_I \beta$	$\mapsto$	$\alpha \wedge \bullet_I (\bigcirc_I \alpha \wedge \beta)$	(push into $\bullet_I$)
23.	$\alpha \wedge \bigcirc_I \beta$	$\mapsto$	$\alpha \wedge \bigcirc_I (\bullet_I \alpha \wedge \beta)$	(push into $\bigcirc_I$)

Figure 5.1: Rewrite rules for ra. The above rules are used when x is a variable that is range-restricted in the subformula α (i.e., $x \in \mathsf{RR}(\alpha)$) and free but not range-restricted in the subformula β (i.e., $x \in \mathsf{FV}(\beta) \setminus \mathsf{RR}(\beta)$).

Note that some of the rewrite rules given in Figure 5.1 may lead to formulae that cannot be monitored. For example, consider rule 6, which describes how to rewrite a formula of the form $(\alpha \wedge \gamma) \dot{\mathcal{S}}_I \beta$ into the logically equivalent formula $(\alpha \wedge \gamma) \dot{\mathcal{S}}_I (\dot{\Diamond}_I \alpha \wedge \beta)$. If we have $I = [c, d)$, where $d \in \mathbb{N}$, there is no problem and the rewritten formula can be handled using our monitor. If, however, we have $I = [c, \infty)$, the rewritten formula, while preserving logical equivalence, is not bounded any more and thus cannot be monitored by our approach. Generally speaking, whenever a formula containing an unbounded past operator (i.e., $I = [0, \infty)$) is rewritten, the rewrite rules in Figure 5.1 may generate a logically equivalent formula that is no longer bounded. Because of this, the rules 6, 10, 12, 14, 16, 18, and 20 require that $I = [c, d)$, where $d \in \mathbb{N}$.

Lemma 5.3.8. *Let $\phi \in \mathsf{MFOTL}^+$ be a safe-range formula, v a valuation, and $i \in \mathbb{N}$. We have*

that

$$(D, \tau, v, i) \models \phi \qquad \textit{iff} \qquad (D, \tau, v, i) \models ra(sr(\phi))\,.$$

Proof. The equivalence follows from Definitions 2.4.17 and 5.2.1, Lemma 5.3.6, and standard equivalences for first-order logic. ⊣

5.3.5 TSF Safe-range Check

A safe-range formula $\theta \in \mathsf{MFOTL}^{+}$ is called TSF *safe-range* if each temporal subformula α of θ and each direct subformula $\delta \in \mathit{dstf}(\alpha)$ is safe-range.

For a safe-range formula $\phi \in \mathsf{MFOTL}^{+}$, we check if $ra(\phi)$ is TSF safe-range. This can be done by examining if each temporal subformula α of $ra(\phi)$ and each direct subformula $\delta \in \mathit{dstf}(\alpha)$ is safe-range and bounded.

Lemma 5.3.9. *Let ϕ be a bounded formula in MFOTL^{+}. If ϕ is* TSF *safe-range then it is* TSF *domain independent.*

Proof. Assume that $\delta \in \mathsf{MFOTL}^{+}$ is bounded. It is straightforward to see by induction over the formula structure that if δ is safe-range, then only finitely many valuations satisfy δ. Hence, δ is temporal domain independent. ⊣

5.4 Space Optimizations

By restricting our setting to finite relations several types of space optimizations can be applied to our incremental constructions. If finite relations are represented using tables, removing unnecessary tuples from a relation directly translates into a lower space requirement to physically store the resulting relation. Note that this is in contrast to the use of automatic structures and automatic representations, where removing a tuple from a relation may translate into a larger space requirement to store the finite state automaton that represents the resulting relation.

We now discuss four types of optimization techniques to reduce the space consumption of our monitoring algorithm. The optimizations described in Sections 5.4.1 and 5.4.2 are specific to our approach and were implemented in our prototypical runtime monitoring framework presented in Chapter 6. In contrast, the optimizations summarized in Sections 5.4.3 and 5.4.4 are straightforward extensions of previous work and are only mentioned for reasons of completeness.

5.4.1 Reducing Redundancy

To minimize the size of the relations, we can optimize our incremental structure construction by removing redundant data tuples from auxiliary relations. For example,

consider the formula $\alpha = \beta\, \mathcal{S}_{[0,\infty)}\, \gamma$ and assume that $\bar{a}$ satisfies γ and β at all time points. In this case, the relation for r_α in $\hat{D}_i$ stores the tuples $(\bar{a}, y)$, with $y = \tau_i - \tau_j$ for all $j \leq i$. However, it suffices to store only one of these tuples.

More generally, for $\alpha = \beta\, \mathcal{S}_I\, \gamma$, we can optimize the construction as follows. If $(\bar{a}, y), (\bar{a}, y') \in r_\alpha^{\hat{D}_i}$ with $y, y' \in I$ and $y > y'$, then we can remove $(\bar{a}, y)$ from $r_\alpha^{\hat{D}_i}$. This follows by an easy inductive argument. Since $y, y' \in I$, both tuples satisfy the condition of our construction so that $\bar{a}$ is put into the relation $p_\alpha^{\hat{D}_i}$. Moreover, if the updated version of $(\bar{a}, y)$ is in $r_\alpha^{\hat{D}_{i+1}}$, then also the updated version of $(\bar{a}, y')$ is in $r_\alpha^{\hat{D}_{i+1}}$, and we have that $y + \tau_{i+1} - \tau_i > y' + \tau_{i+1} - \tau_i$. Again, both updated tuples satisfy the condition such that $\bar{a}$ is put into the relation $p_\alpha^{\hat{D}_{i+1}}$.

Similar optimizations apply to the case where $\alpha = \beta\, \mathcal{U}_I\, \gamma$ with $I = [c, d)$. The relation $s_\alpha^{\hat{D}_i}$ may contain redundant elements. Namely, if $(\bar{a}, j_1, j_1'), (\bar{a}, j_2, j_2') \in s_\alpha^{\hat{D}_i}$ with $[j_1, j_1') \subsetneq [j_2, j_2')$ then we can remove $(\bar{a}, j_1, j_1')$ from $s_\alpha^{\hat{D}_i}$. After removing such elements, $s_\alpha^{\hat{D}_i}$ only contains tuples where the intervals given by the last coordinates of an element in $s_\alpha^{\hat{D}_i}$ is maximal. When filtering out these elements, we need to adjust the update of $s_\alpha^{\hat{D}_i}$ slightly because elements of the form $(\bar{a}, 0, j') \in s_\alpha^{\hat{D}_{i-1}}$ must no longer be ignored. Note that the optimization has removed the element $(\bar{a}, 1, j')$ from $s_\alpha^{\hat{D}_{i-1}}$.

Another optimization is to remove a tuple $(\bar{a}, j, j')$ in $s_\alpha^{\hat{D}_i}$ if $\tau_{i+j'+1} - \tau_i < c$ and $j' < \ell_i$. Recall that ℓ_i is the lookahead offset at the time point i, i.e., $\ell_i := \max\{j \in \mathbb{N} \mid \tau_{i+j} - \tau_i < d\}$. Note that the semantics of the $\mathcal{U}_I$ operator requires that $\bar{a}$ has to satisfy γ at the time point $i + j'$. Since this time point is too close to time instant i, the timing constraint given by the interval I is violated. We remark that we cannot remove the element $(\bar{a}, j, j')$ if $j' = \ell_i$ since we might need the information that $\bar{a}$ satisfies β in the time instants $i + j, \ldots, i + \ell_i$ when updating the relation for s_α.

Finally, we can use the relation $r_\alpha^{\hat{D}_i}$ to eliminate elements in $s_\alpha^{\hat{D}_i}$. Namely, we can eliminate $(\bar{a}, j, j')$ in $s_\alpha^{\hat{D}_i}$ if $j' < \ell_i$ and when there is no $(\bar{a}, k) \in r_\alpha^{\hat{D}_i}$ with $j \leq k$ and $k - 1 \leq j'$.

5.4.2 Dedicated Constructions for Derived Operators

Another kind of optimization is to tune the above definitions for the auxiliary predicates for certain kinds of formulae. For instance, if $\alpha = \Diamond_I\, \gamma$ (i.e., $\alpha = true\, \mathcal{U}_I\, \gamma$) then we do not need the auxiliary relations for s_α at all. Furthermore, some of the tuples can be removed from $r_\alpha^{\hat{D}_i}$. Namely, if $(\bar{a}, j)$ and $(\bar{a}, j')$ are in $r_\alpha^{\hat{D}_i}$, with $j < j'$, then $(\bar{a}, j)$ can be removed from $r_\alpha^{\hat{D}_i}$. This can be seen by an argument similar to the one we gave when optimizing relations that handle the operator $\mathcal{S}_I$.

5.4.3 Algebraic Transformations

The rewriting techniques given for past-only FOTL in [CT95] can be extended to the bounded MFOTL fragment. Thereby, we can reduce the number of auxiliary relations

created from an input formula and also minimize their arity. For example, by rewriting the formula $\exists x.\, \bigcirc_I \beta$ to $\bigcirc_I \exists x.\beta$ we reduce the arity of the auxiliary relation $p^{D_i}_{\bigcirc_I \exists x.\beta}$ by one. Similarly, by rewriting the formula $\bigcirc_I \bullet_I \beta$ to β we can minimize the number of auxiliary relations created. Under certain conditions, formulae containing nested metric operators with different intervals can also be rewritten. For example, if $c \geq d'$, the formula $\Diamond_{[c,d)} \blacklozenge_{[c',d')} p(x)$ can be rewritten to $\Diamond_{[c-d',d-c')} p(x)$. Similar optimizations apply for formulae of the kind $\bigcirc_I \beta \vee \bigcirc_I \gamma$, $\bigcirc_I \beta \wedge \bigcirc_I \gamma$, and $\Diamond_I \blacklozenge_I p(x)$.

5.4.4 Context-based Optimizations

We can also reduce the number of tuples stored in the auxiliary relations by analyzing the contexts in which the relations are used and by then restricting the definitions of the auxiliary relations with appropriate *magic conditions* as described in [CT95]. For example, for a formula $x \prec 10 \wedge (q(x) \vee \beta(x)\, \mathcal{U}_I\, \gamma(x))$, we can adapt the definitions of the auxiliary relations $r^{\hat{D}_i}_{\beta(x)\mathcal{U}_I\gamma(x)}$ and $s^{\hat{D}_i}_{\beta(x)\mathcal{U}_I\gamma(x)}$ such that only those elements that satisfy the condition $x \prec 10$ are stored. Note that the availability of both future and metric operators in MFOTL requires that the definitions of [CT95] be adapted accordingly.

5.5 Analysis of Space Consumption

In the following, we analyze the space requirements of our runtime monitor for the setting, where all relations are finite. For the remainder of this chapter, we assume that the given formula ϕ is bounded and TSF domain independent.

5.5.1 Modifications

Before we proceed with our analysis, we slightly modify our monitor from Chapter 4.

First, observe that the counters q and i of $\mathcal{M}(\phi)$ grow arbitrarily large when processing the sequence $(D_0, \tau_0), (D_1, \tau_1), \ldots$. This problem can be partly overcome by replacing these two counters with a single counter that stores $i - q$, i.e., the distance between the current time and the time when the last evaluation of $\neg\hat{\phi}$ took place. Still, $i - q$ can become arbitrarily large if ϕ contains a temporal subformula of the form $\beta \mathcal{U}_I \gamma$ and the number of time points with the same time stamp in (D, τ) is unbounded, i.e., $\{j' - j \mid \tau_j = \tau_{j+1} = \cdots = \tau_{j'} \text{ with } j \leq j'\}$ is infinite. Note that tuples stored in Q also contain indices of the sequence (D, τ). These indices must also be made relative to q.

A problem related to the above one is that the difference between time stamps can be arbitrarily large. If ϕ contains a subformula of the form $\alpha = \beta\, \mathcal{S}_{[c,\infty)}\, \gamma$, the auxiliary relations $r^{\hat{D}_0}_\alpha, r^{\hat{D}_1}_\alpha, \ldots$ may need to store tuples whose last component grows with each i. To overcome this problem, we slightly modify the incremental construction of $r^{\hat{D}_i}_\alpha$ for $i > 0$. Namely, the "age" of a tuple $(\bar{a}, y) \in r^{\hat{D}_{i-1}}_\alpha$ is only increased when it is less

than c. For this special case, we define

$$r_\alpha^{\hat{D}_i} := \left(\hat{\gamma}^{\hat{D}_i} \times \{0\}\right) \cup \left\{(\bar{a}, \min\{c, z + \tau_i - \tau_{i-1}\}) \in \mathbb{N}^{n+1} \,\middle|\, \bar{a} \in \hat{\beta}^{\hat{D}_i} \text{ and } (\bar{a}, z) \in r_\alpha^{\hat{D}_{i-1}}\right\}.$$

This new construction ensures that the size of the last component of the tuples in $r_\alpha^{\hat{D}_i}$ is bounded. Similar to Lemma 4.4.7, we can prove that this construction has the desired properties.

In the following, we assume that the monitor $\mathcal{M}(\phi)$ uses this modified construction and the relative indexing as explained above.

5.5.2 Analysis

We now analyze the sizes of the auxiliary relations stored by our monitor in each loop iteration. We first introduce the following abstract notion for analyzing the resources consumed by runtime monitors in general.

Analysis Method

Let C be a class of temporal structures over the signature $S = (\mathsf{C}, \mathsf{R}, a)$ and let $pre(C)$ denote the set of nonempty finite prefixes of the temporal structures in C.

Definition 5.5.1. *Let $f, g : pre(C) \to \mathbb{N}$ and $s : \mathbb{N} \to \mathbb{N}$ be functions. We write $f \lhd^s g$ if $f(\bar{D}, \bar{\tau}) < s(g(\bar{D}, \bar{\tau}))$, for all $(\bar{D}, \bar{\tau}) \in pre(C)$.*

In our context, the function $f : pre(C) \to \mathbb{N}$ measures the consumption of a particular resource (e.g., storage) of a monitor after it has processed the finite prefix $(\bar{D}, \bar{\tau})$. The function $g : pre(C) \to \mathbb{N}$ measures the size of the prefix $(\bar{D}, \bar{\tau})$. Intuitively, $f \lhd^s g$ means that, at any time point, the resource consumption (measured by f) of the monitor is bounded by the function $s : \mathbb{N} \to \mathbb{N}$ with respect to the size of the processed prefix (measured by g) of an input from C.

In our analysis of the monitor $\mathcal{M}(\phi)$, we use the following concrete functions f and g. Let $(\bar{D}, \bar{\tau}) \in pre(C)$ with $\bar{D} = (D_0, \ldots, D_i)$ and $\bar{\tau} = (\tau_0, \ldots, \tau_i)$.

- We define $g(\bar{D}, \bar{\tau}) := |adom(\bar{D})|$, where $adom(\bar{D})$ is the active domain of $(\bar{D}, \bar{\tau})$. Note that g only counts the number of elements of $\bar{D}$ that are constants or that occur in some of $\bar{D}$'s relations. It ignores the sizes of these elements, the number of times an element appears in $\bar{D}$, and where an element occurs. Moreover, it ignores the time stamps in $\bar{\tau}$.
- We define $f(\bar{D}, \bar{\tau})$ to be the sum of the cardinalities of the relations for $r \in \hat{\mathsf{R}}$ stored by $\mathcal{M}(\phi)$ after the $(i+1)$st loop iteration, having processed the input $(D_0, \tau_0), (D_1, \tau_1), \ldots, (D_i, \tau_i)$.

Observe that $f \lhd^s g$ is a desirable property of a monitor. Intuitively, it says that the amount of data stored by the monitor does not depend on how long the monitor has

been running but only on the number of domain elements that appeared so far. Furthermore, the stored data is bounded by the function s. We remark that the property of a (polynomially) bounded history encoding [Cho95] can be formalized as $f \lhd^s g$, for some (polynomial) $s : \mathbb{N} \to \mathbb{N}$.

Main Result

Before we prove our main result on the space consumption of our runtime monitors in the setting with finite relations, we first establish Lemma 5.5.2. Recall that ϕ is a bounded and TSF domain independent formula (see Definition 5.2.2) and (D, τ) a timed temporal database over the signature $S = (\mathsf{C}, \mathsf{R}, a)$, i.e., all relations for $r \in \mathsf{R}$ are finite.

Lemma 5.5.2. *Let α be a temporal subformula of ϕ or a direct subformula of a temporal subformula of ϕ. Assume that $\mathcal{M}(\phi)$ constructs the relation $p_\alpha^{\hat{D}_j}$ in the $(i+1)$st loop iteration on the input (D, τ). Then, $\alpha^{(\hat{D},\tau,j)} \subseteq adom(\bar{D})^n$, where $\bar{D} = (D_0, \ldots, D_i)$ and n is the number of free variables of α.*

Proof. Since ϕ is TSF domain independent, we have that α is temporal domain independent. From the correctness of the monitor $\mathcal{M}(\phi)$, it follows that i is minimal for j, (D, τ), and α. Similar as in Lemma 5.2.3, we show that $\alpha^{(D,\tau,j)} \subseteq adom(\bar{D})^n$. ⊣

With Lemma 5.5.2, we can now prove Theorem 5.5.3, which states that—under a technical restriction—the space consumption of our modified monitor is polynomially bounded by the cardinality of the data constants appearing in the processed prefix.

Theorem 5.5.3. *Let C be a class of temporal databases. Assume that there is some $\ell \in \mathbb{N}$ such that $\max\{j \mid \tau_i = \tau_{i+1} = \ldots = \tau_{i+j}\} < \ell$, for all $(D, \tau) \in C$ and all $i \in \mathbb{N}$. Then, we have that $f \lhd^s g$, where $s : \mathbb{N} \to \mathbb{N}$ is a polynomial of degree $\max\{a(r) \mid r \in \hat{\mathsf{R}}\}$.*

Proof. Throughout the proof, we assume that the monitor $\mathcal{M}(\phi)$ processes the temporal structure (D, τ) from the class C. Let us first make the following observation. If $q = 0$, the monitor stores in the $(i+1)$st iteration at most the relations $r^{\hat{D}_q}, r^{\hat{D}_{q+1}}, \ldots, r^{\hat{D}_i}$, for $r \in \hat{\mathsf{R}}$. If $q > 0$, the monitor might additionally store the relations $r^{\hat{D}_{q-1}}$, for $r \in \hat{\mathsf{R}}$. Without loss of generality, we assume that $q > 0$. We now proceed as follows. (1) We establish an upper bound on $i - q$. (2) We establish an upper bound on the cardinalities of the relations $r^{\hat{D}_j}$ with $q - 1 \leq j \leq i$.

Let us start with (1). Recall that ℓ is the bound on the number of equal time stamps in the sequence τ. Namely, we have that $\max\{j \mid \tau_k = \tau_{k+1} = \cdots = \tau_{k+j}\} < \ell$, for all $k \in \mathbb{N}$. The bound of $i - q$ depends on ℓ and the temporal future operators that occur in ϕ. First, observe that since there are at most ℓ equal time stamps in τ, the monitor postpones the evaluation of a formula $p(x)\, \mathcal{U}_{[c,d)}\, q(x)$ at the time point i by at most $\ell \cdot d$ time steps. Furthermore, note that a formula of the form $\bigcirc_I p(x)$ is postponed by one

time step. Taking the nesting of subformulae with temporal operators into account, we define

$$maxwaitfor(\theta) := \begin{cases} maxwaitfor(\beta) & \text{if } \theta = \neg\beta,\ \theta = \exists x.\, \beta, \\ & \quad \text{or } \theta = \bullet_I\, \beta, \\ \max\{maxwaitfor(\beta), maxwaitfor(\gamma)\} & \text{if } \theta = \beta \wedge \gamma \text{ or } \theta = \beta\, \mathcal{S}_I\, \gamma, \\ 1 + maxwaitfor(\beta) & \text{if } \theta = \bigcirc_I\, \beta, \\ \ell d + \max\{maxwaitfor(\beta), maxwaitfor(\gamma)\} & \text{if } \theta = \beta\, \mathcal{U}_{[c,d)}\, \gamma, \\ 0 & \text{otherwise.} \end{cases}$$

From the initialization of the list Q and the updates of Q in each loop iteration, it follows by induction that $i - q \leq maxwaitfor(\phi)$.

Let us now turn to (2). We define m as $\max\{a(r) \mid r \in \mathsf{R}\}$ and k as the maximum of $1 + maxwaitfor(\phi)$ and the maximal upper bound of an interval of a since operator in ϕ, where we set the maximum of the upper bound of an interval $[c, \infty)$ in a since operator to c.

Assume that the monitor constructs a relation $r^{\hat{D}_j}$ for $r \in \hat{\mathsf{R}}$ at time point i, i.e., in the $(i+1)$st loop iteration. In the following, we give an upper bound on the cardinality of $r^{\hat{D}_j}$. First note that the data elements $d \in |D|$ that occur in $r^{\hat{D}_j}$ also occur in $adom(\bar{D})$, where $\bar{D} = (D_0, \ldots, D_i)$ and $\bar{\tau} = (\tau_0, \ldots, \tau_i)$. This follows from Lemma 5.5.2.

- $r \in \mathsf{R}$ or $r = p_\alpha$, where α is a temporal formula: We have that $|r^{\hat{D}_j}| \leq |adom(\bar{D})|^m$.
- $r = r_\alpha$, where α has the form $\beta\, \mathcal{S}_I\, \gamma$: Recall that in this case $r^{\hat{D}_j}$ consists of tuples of the form $(\bar{a}, y)$ with $y \leq k$. Note that $y \leq k$ holds because our optimized construction for the temporal operator $\mathcal{S}_{[c,\infty)}$. We have that $|r^{\hat{D}_j}| \leq |adom(\bar{D})|^m \cdot k$.
- $r = r_\alpha$, where α has the form $\beta\, \mathcal{U}_I\, \gamma$: Recall that in this case $r^{\hat{D}_j}$ consists of tuples of the form $(\bar{a}, j)$ with $j \leq k$. We have that $|r^{\hat{D}_j}| \leq |adom(\bar{D})|^m \cdot k$.
- $r = s_\alpha$, where α has the form $\beta\, \mathcal{U}_I\, \gamma$: Recall that in this case $r^{\hat{D}_j}$ consists of tuples of the form $(\bar{a}, j, j')$ with $j, j' \leq k$. We have that $|r^{\hat{D}_j}| \leq |adom(\bar{D})|^m \cdot k^2$.

We conclude that $|r^{\hat{D}_j}| \leq |adom(\bar{D})|^m \cdot k^2$, for all $r \in \hat{\mathsf{R}}$.

Form this and the upper bound on $i-q$, we obtain that $f(\bar{D}, \bar{\tau}) \leq |adom(\bar{D})|^m \cdot |\hat{\mathsf{R}}| \cdot k^3$. Since k only depends on ℓ and ϕ, we have that $f \lhd^s g$, for $s(x) := c \cdot x^m$ with the constant $c := |\hat{\mathsf{R}}| \cdot k^3$. ⊣

Note that if such a bound ℓ on the sequence τ of time stamps does not exist, we cannot guarantee any upper bound on f. To see this, consider the formula $\phi = (p(x)\, \mathcal{U}_{[0,1)}\, q(x)) \wedge (p'(x)\, \mathcal{U}_{[0,2)}\, q'(x))$ and a temporal database (D, τ), where the relations for p, p', q, and q' are all singletons and equal. We have that $g(\bar{D}, \bar{\tau}) = 1$, for all finite prefixes $(\bar{D}, \bar{\tau})$ of (D, τ). However, if τ contains a sequence $\tau_i, \ldots, \tau_{i+j}$ with $\tau_i = \tau_{i+1} = \cdots = \tau_{i+j-1} =$

$\tau_{i+j} - 1$, we have that $f(\bar{D}, \bar{\tau}) \geq j$, for the prefix $(\bar{D}, \bar{\tau})$ with $\bar{D} = (D_0, \ldots, D_{i+j})$ and $\bar{\tau} = (\tau_0, \ldots, \tau_{i+j})$. The reason for this is that our monitor stores at least the nonempty relations $p^{\hat{D}_i}_{p(x)\mathcal{U}_{[0,1)}q(x)}, \ldots, p^{\hat{D}_{i+j-1}}_{p(x)\mathcal{U}_{[0,1)}q(x)}$ at the end of the $(i+j+1)$st loop iteration. If we can choose j arbitrarily large, we can exceed $s(g(\bar{D}, \bar{\tau}))$, for any $s : \mathbb{N} \to \mathbb{N}$.

5.5.3 Discussion

Our result contrasts with a similar result described in [Cho95], where a related monitor construction for past-only, non-metric, and TSF domain independent FOTL formulae was shown to be polynomially bounded by the cardinality of the data appearing in the processed prefix. In particular, our result demonstrates that this polynomial bound on the monitor's space consumption can be maintained by our monitor construction even for the richer fragment consisting of bounded TSF domain independent MFOTL formulae. Recall that it is unknown whether the past-only fragment of MFOTL is as expressive as the fragment with both past and bounded future operators and whether formulae in the past-only fragment can be expressed as succinctly as those in the future-bounded fragment.

In [Tom03], Toman describes the use of two-sorted first-order logic (2-FOL) to query database histories. More specifically, he presents a data expiration technique to remove irrelevant data from a database history with respect to a given property expressed by a range-restricted 2-FOL formula. Note that our incremental constructions in Section 4.4 can be seen as a query-based data expiration technique as the constructions discard irrelevant data from the monitored timed temporal structure. A monitoring approach using Toman's data expiration technique for 2-FOL is bounded by a function with a stack of exponentials [Tom03]. The height of the stack is given by the quantifier depth of the given 2-FOL formula. The polynomial upper bound of the presented MFOTL monitor thus suggests that MFOTL is better suited for monitoring than 2-FOL whenever the property under consideration is expressible as a TSF domain independent MFOTL formula that can be handled by the monitor $\mathcal{M}(\phi)$.

Our monitoring approach is based on executions represented as timed temporal structures with natural-numbered time stamps, yielding a fictitious clock time semantics. It is important to note that our choice of a fictitious clock time semantics is decisive in order to establish a bounded history encoding. In the setting with finite relations, one might also be tempted to chose the real numbers $\mathbb{R}$ as an alternative time domain. While our constructions continue to work without changes also in the presence of real-numbered time stamps, the choice of a dense time domain such as the real numbers cannot achieve a bounded history encoding. The reason is that in a dense time domain, the number of distinct time stamps that may occur within the time window induced by a bounded MFOTL formula cannot be bounded. This makes it impossible to bound the size of the auxiliary relations and thus prevents a bounded history encoding.

Finally, we remark that it is open whether Theorem 5.5.3 can be carried over to temporal structures with possibly infinite relations and automatic representations. To begin with, it is not clear how to define the function g that measures the size of the automatic representations. In particular, g should be independent of the length of the prefix. Moreover, establishing tight upper bounds on the size of automata for automatic structures is difficult and only a few results for specific automatic structures exist (see [Kla04]).

5.6 Summary and Outlook

In this chapter, we focussed on a special case of our runtime monitoring approach, namely the case where relations are always finite. We first explained the difficulties involved with finite relations and introduced the notion of TSF domain independence as a characterization of those MFOTL formulae that can be monitored using our approach for finite relations. We then presented an extension of an existing rewrite procedure to ensure that only those formulae are admitted for monitoring that can be handled by our approach. We then presented a number of space optimizations and went on to prove that—under a technical restriction—the space consumed by our runtime monitor for finite relations is polynomially bounded by the cardinality of the set of data elements appearing in the processed prefix.

Before concluding this chapter, let us briefly address the question whether, by requiring relations to be finite, our monitoring approach is still practically useful. As a first indication of its usefulness, we noted in Example 5.2.4 that the example property from Section 1.3 is expressible as a bounded TSF domain independent MFOTL formula. As a result, it can be handled by our monitoring approach also in the context where relations are all finite. Moreover, also the properties presented in Example 4.1.1 or the one in Section 4.5 can be expressed as bounded TSF domain independent MFOTL formulae and thus handled by our monitor for finite relations. More evidence of the practical feasibility of our approach will be presented in the forthcoming chapters.

We believe that the use of (timed) sequences of first-order structures with finite relations to represent system executions is natural. This not only corresponds to (timed) temporal databases or data streams, it also matches the intuition that only a finite amount of new information about a monitored system can become available at each time point. While runtime monitoring approaches similar to our own have been used for the checking of dynamic integrity constraints in databases, our runtime monitoring approach also offers a top-down alternative to bottom-up data stream management approaches. By automatically constructing a correct runtime monitor from a declarative property specification, our approach delegates all implementation and optimization aspects to the automated monitor construction. It remains to be seen whether our own or similar declarative monitoring approaches can be sufficiently optimized in order to

compete with domain-specific and algorithm-based monitoring techniques also from a performance perspective.

Regarding future work, we point out that our rewriting-based procedure to identify a subset of the TSF domain independent MFOTL formulae represents only one of several possible approaches. An alterative approach to ensure finite relations was adopted in an implementation of Chomicki's approach for monitoring TSF domain independent past-only FOTL formulae [CT95]. The idea is to syntactically check whether a past-only FOTL formula can be handled by their monitoring approach based on a temporal extension of the class of evaluable formulae [GT91, Dem92]. The class of evaluable formulae represents a larger subset of the class of domain independent formulae than the class of safe-range formulae. It remains to be seen whether the class of evaluable formulae can also be extended to MFOTL formulae.

Part II

Applications

"In theory, theory and practice are the same. In practice, they are not."

Lawrence Peter "Yogi" Berra

Chapter 6

A Prototypical Monitoring Framework

To practically validate some of our theoretical results, we implemented our monitoring approach for finite relations in a prototypical runtime monitoring framework. The framework allows for generating runtime monitors for properties expressible as bounded and TSF domain independent MFOTL formulae. Its main purpose is to provide an environment and tool for conducting experiments. Specifically, we used the framework to validate our approach by experimentally analyzing the space consumption of runtime monitors generated for selected formula classes.

The structure of this chapter is as follows. In Section 6.1, we give a brief overview of our prototype. In Section 6.2, we first describe our methodology for experimentally validating the practical feasibility of our monitoring approach. We then present and discuss the results of a general analysis of our runtime monitors' space consumption for various formula classes, including the example property of our running example.

6.1 Implementation

This section presents our implementation. We first give a brief overview and explain the typical usage of the framework. We then describe the individual components of the architecture in more detail. Finally, we also explain how the framework can be used for the offline analysis of log files.

6.1.1 Overview

The framework was implemented using the Java programming language and provides general purpose monitoring capabilities for properties expressible as bounded and TSF domain independent MFOTL formulae. Moreover, it allows for online monitoring and offline analysis alike. Generated runtime monitors can be embedded into an event-based distributed system as dedicated monitoring components and process events in an online fashion. Alternatively, generated monitors can also be used for the offline analysis of arbitrary log files.

Usage. To instantiate a runtime monitor, a first-order signature S must be defined using a simple declarative language. Alternatively, an existing signature may be selected from a dedicated signature store. In a next step, a desired property is specified by an appropriate bounded and TSF domain independent MFOTL formula ϕ over the selected signature. The MFOTL formula is also specified using a concrete syntax version of MFOTL, which provides commonly used derived temporal operators such as $\Diamond_I$ and Boolean connectives such as $\vee$ as primitive operators. The reason to provide direct syntax and implementations for derived operators is twofold. First, this often allows for specifying properties more succinctly. Second, they can be implemented more efficiently and with less overhead. The concrete textual syntax is very close to the abstract syntax defined in Definition 2.4.16. It is given in Appendix B.2. In another step, one or several so-called event sinks are newly defined or selected from a set of available event sinks. The specified MFOTL formula is then lexically analyzed, parsed, and checked for well-formedness and consistency with the selected signature.

If the formula passes all required checks, a new monitor $\mathcal{M}(\phi)$ for ϕ is automatically instantiated and associated with the selected event sinks. Event sinks are abstract end points where monitored applications or other components send new state or event information to. As an event sink receives a new event (i.e., a time-stamped first-order structure with finite relations), it relays the event to all associated monitors. Upon activation of the monitor $\mathcal{M}(\phi)$, the monitor is notified of received events and processes them in their order of arrival. As soon as the monitored property can be evaluated with respect to a time point $q \in \mathbb{N}$, the monitor $\mathcal{M}(\phi)$ outputs the tuples that satisfy (or violate) ϕ at q. Depending on the monitor's configuration, the results of each time point are output to the console or written to a text file.

Limitations. Note that the current version of our framework is not a production-quality monitoring framework yet. While we implemented space optimization techniques such as those described in Sections 5.4.1 and 5.4.2, the current version does not implement the techniques described in Sections 5.4.3 and 5.4.4. Moreover, our monitors do not employ any particular formula rewriting strategy as typically used in database query processing to improve the processing speed of our monitors. Consequently, the main purpose of the current monitoring framework is to provide an environment and tool for conducting experiments. In this chapter, we report on the analysis of the space consumption of our monitors in the context of our running example and for related formula classes.

6.1.2 Architecture

Figure 6.1 presents the high-level view of the architecture of our runtime monitoring framework. We now briefly describe its main components.

Signature Manager

The *signature manager* component provides the ability to lexically analyze, parse, and store arbitrarily defined first-order signatures. Accordingly, the signature manager component consists of three sub-components, namely a lexical analyzer (lexer for short), a parser, and a signature store. As usual, the lexer reads a signature definition as a stream of input characters from the command line or a text file and groups them into tokens. The tokens are then passed to the parser. The parser then reads a stream of such tokens and tries to recognize the grammatical structure of the signature definition. The result of a successful parsing step is an abstract syntax tree providing an in-memory representation of the input signature. The abstract syntax tree forms the basis for further processing and can be stored in the signature store. Previously stored signatures can also be loaded from the signature store.

We implemented both the lexer and the parser using the ANTLR v3 parser generator [Par07]. In Appendix B.1, we show the lexer and the parser grammars from which ANTLR v3 generates the Java code that analyzes and parses new signature definitions. The grammars also define the concrete syntax to specify new signatures.

Example 6.1.1. *(Running example) The signature S_{TPS} of our running example can be expressed by the following concrete signature definition:*

```
TPS-Signature:
predicates: trans(tID,cID), report(tID,cID);
```

Property Manager

Similar to the signature manager, the *property manager* component consists of dedicated sub-components to tokenize and parse property specifications expressed using a concrete MFOTL syntax. Moreover, two additional sub-components are responsible for the analysis and the transformation of the temporal property specifications into first-order formulae over extended signatures. For a each bounded TSF domain independent MFOTL formula ϕ that is successfully parsed, analyzed, and transformed, a new monitor component $\mathcal{M}(\phi)$ is automatically instantiated.

For the implementation of all sub-components of the property manager, we also made use of ANTLR v3. As above, we used individual ANTLR v3 grammars to define the concrete syntax of MFOTL and to generate the Java code for a respective lexer and a parser. In addition, we used several tree grammars to analyze and transform the input formulae. In Appendix B.2, we present the respective grammar specification from which ANTLR v3 generated the basic Java code of our lexer and the parser. The grammar also defines the concrete MFOTL syntax used to define properties that shall be monitored.

Example 6.1.2. *(Running example) Using our concrete syntax and assuming the above signature, we can determine violations of our example property by the following formula:*

```
((trans(?t,?c) and (once[0,30](
     exists ?t1. (trans(?t1,?c)
        and (eventually[0,2] report(?t1,?c))))))
 and (not (eventually[0,2] report(?t,?c))))
```

Monitor Controller

The *monitor controller* is the core component of our runtime monitoring framework. Its main capabilities are the instantiation of new runtime monitors from bounded and TSF domain independent MFOTL formulae and the management (e.g., activation, pausing, and de-activation) of any runtime monitor that is available in the monitor store of the monitoring controller component. Each monitor is a separate sub-component with an own data store maintaining the relations and auxiliary relations required to incrementally process a timed temporal structure with finite relations. The monitors are implementations of the online algorithm given in Figure 4.2.

At each time point when a monitor receives a new event (i.e., a first-order structure with an associated time stamp) from an event sink, the monitor updates its data store, specifically the maintained auxiliary relations and the list of unevaluated subformulae, according to the definitions given in Sections 4.4 and 4.6. As soon as all the auxiliary relations required to evaluate the monitored property with respect to the evaluation index are built, the monitor evaluates the transformed first-order formula over the extended signature and outputs all satisfying assignments at that time point. Technically, the updating of auxiliary relations and the evaluation of the transformed first-order formula and its subformulae is implemented by a canonical mapping of first-order formulae to relational algebra operations [Ull88].

For example, let D denote a relational database over a signature $S = (\mathsf{C}, \mathsf{R}, a)$ with $p, q \in \mathsf{R}$ and $a(p) = a(q) = 3$. Moreover, let $x, y, z, v, w \in \mathsf{V}$, where V is a set of variables disjoint from C and R. The first-order formula $p(x, y, z) \wedge q(v, y, w)$ is evaluated over D by computing a natural join operation of the relations p^D and q^D. Observe that the resulting relation $(p \wedge q)^D$ is 5-ary and that $(p \wedge q)^D \subseteq adom(D)^5$.

Event Sink Controller

The *event sink controller* component provides the ability to define and instantiate event sinks. Moreover, it allows for the registration of runtime monitors with one or several such event sinks. Following the publish/subscribe paradigm, event sinks provide an abstract interface for arbitrary internal or external components to send relevant state or event information. As new information is made available to an event sink (e.g., from a component processing log files or a monitored system component), the event sink controller leverages an observer pattern [GHJV95] to ensure that all registered runtime monitors receive this information for further processing. The use of event sinks as

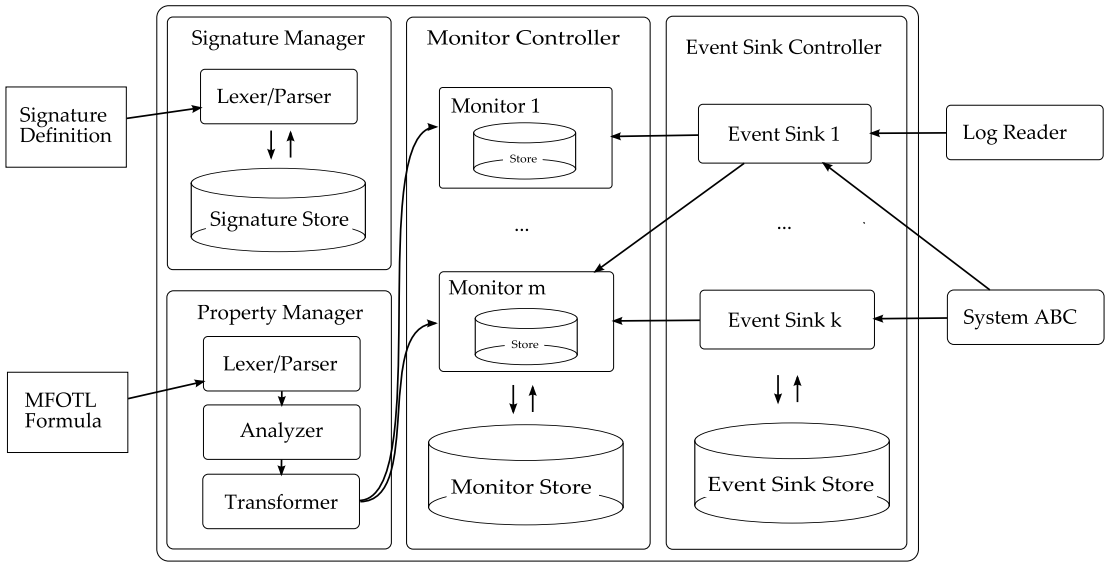


Figure 6.1: Architecture of the prototypical monitoring framework (simplified).

abstract end points thus allows for the integration of monitoring in a loosely coupled way and supports both offline and online monitoring alike.

6.1.3 Offline Analysis of Log Files

For the offline analysis of log files, our monitoring framework provides a simple console interface. To perform an offline analysis of a log file, we start the console application with three inputs: a signature definition file, a bounded and TSF domain independent MFOTL formula, and a log file. Both the input signature and the formula are specified using the concrete syntax given in Appendix B. Of course, the formula and the log file must correspond to the signature. If this is the case, a runtime monitor is automatically created for the input MFOTL formula. A dedicated log reader component then reads the log file line-by-line and sends each row to the generated monitor for incremental processing.

The log file is expected to be a comma-separated text file, where—except for the title row—each row represents a time point with an associated time stamp and a first-order structure over the associated signature. In Figure 6.2, we provide a tabular representation of such a log file for the signature S_{TPS} of our running example. The entries in the first column are interpreted as time stamps and the other columns represent the tuple values of singleton relations associated with predicates defined in the input signature. A log file thus represents a timed database history over the input signature, where all relations are singletons. The entries in the log file are assumed to be linearly ordered by the time stamps provided in the first column. In our current implementation of the log reader component, time stamps are expected to be specified in UNIX time, i.e., in elapsed seconds since 01.01.1970 00:00 Coordinated Universal Time (UTC). Note

though that our framework also handles smaller time units for the time stamps such as milliseconds.

We used the console version of our monitoring framework for our experimental validation, which we describe in the next section.

6.2 Experimental Validation

In this section, we describe the experiments made to validate our monitoring approach for the setting with finite relations and bounded TSF domain independent MFOTL formulae. Our goal was to complement Lemma 5.5.3 with a more practical understanding of the space consumed by the monitors generated for several formula classes and how this changes under different parameter settings. In particular, we thus wanted to validate whether monitoring practically relevant formulae was indeed feasible.

In the next section, we first describe our validation methodology. Finally, we present our results and discuss their significance.

For the remainder of this section, let $S = (\mathsf{C}, \mathsf{R}, a)$ be a signature, (D, τ) with $D = (D_0, D_1, \dots)$ and $\tau = (\tau_0, \tau_1, \dots)$ be a timed temporal database over S, and let ϕ be a bounded and TSF domain independent MFOTL formula over S. Moreover, we write ϕ_{TPS} to denote the MFOTL formula defining our example property from Example 2.4.20.

6.2.1 Methodology

We now present our validation methodology. To understand the rationale behind it, recall that our runtime monitors are online algorithms that process possibly infinite timed temporal structures. Moreover, note that the space consumption of a runtime monitor at each time point depends both on the property being monitored and on the prefix of the timed temporal structure observed up to that time point. To properly assess the space consumption of our online runtime monitors, we thus conducted a steady-state analysis for each of them as described in Appendix A.

We conducted the analysis for several concrete MFOTL formulae, each representing a specific formula class. Moreover, each monitor was investigated under a variety of parameter combinations. As parameters we considered the size of the sample domain from which input data (i.e., timed temporal structures) were generated and the relative event frequency, i.e., the average number of past or future time points that must be considered to evaluate bounded MFOTL formulae at a given time point.

In the following, we present the individual steps of our experimental validation. We first present and motivate the investigated formula classes. We then explain our procedure to generate the timed temporal structures required to experimentally analyze each monitor. Finally, we explain how we measured the space consumption and how we estimated the steady-state performance of our monitors.

Analyzed Formula Classes

We investigated the space consumption of our runtime monitors for several formula classes.

Formula classification. We classified bounded TSF domain independent MFOTL formulae over S as follows:

Past-only formulae Π_k: For $k \in \mathbb{N}$, we denote as Π_k the class of TSF domain independent past-only MFOTL formulae with k nested past operators. For example, the formula $\phi := \alpha \vee (\bullet_{[0,5)}\, \beta \wedge (\blacklozenge_{[4,9)}\, \gamma))$ with $\alpha, \beta, \gamma \in \mathsf{FOL}$ is an element of Π_2.

Future-only formulae Φ_k: For $k \in \mathbb{N}$, we denote as Φ_k the class of TSF domain independent future-only bounded MFOTL formulae with k nested future operators. For example, the formula $\psi := \bigcirc(\alpha \vee (\beta\, \mathcal{U}_{[2,8)}\, (\bigcirc\, \gamma)))$ with $\alpha, \beta, \gamma \in \mathsf{FOL}$ is an element of Φ_3.

Alternating formulae Λ_k: For $k \in \mathbb{N}$, denote as Λ_k the class of TSF domain independent bounded MFOTL formulae with k nested and alternating past and future operators. For example, the formula $\psi := \bigcirc(\alpha \vee (\beta\, \mathcal{S}_{[2,8)}\, \gamma))$ with $\alpha, \beta, \gamma \in \mathsf{FOL}$ is an element of Λ_2.

Note that the outermost temporal operator $\Box$ is not counted in the classification. Moreover, observe that $\Pi_0 = \Phi_0 = \Lambda_0 = \mathsf{FOL}$ and that $\Lambda_1 = \Pi_1 \cup \Phi_1$.

Investigated formulae. Table 6.1 shows the set of formulae over S_{TPS} that we investigated in our experiments. For the sake of comparison, we used the signature $S_{\text{TPS}} = (\mathsf{C}, \mathsf{R}, a)$ from our running example for all investigated formulae (see Example 2.3.2). For each of these formulae, we analyzed the space consumption of the respective monitor under different parameter settings. In addition, we also analyzed the space consumption of the monitor $\mathcal{M}(\phi_{\text{TPS}})$ for our running example. Observe that all formulae are elements of either Π_k, Φ_k, or Λ_k, where $k \in \{1, 2\}$.

For the sake of comparison, the selected formulae all use metric temporal operators and are bounded both towards the past and towards the future. Moreover, we deliberately chose all formulae to induce a sliding window of the same size in terms of the maximal difference between relevant time stamps. As for our running example, this time window consisted of 32 days (i.e., 2,764,800 seconds) for all the investigated formulae. Furthermore, observe that none of the formulae includes negation, existential quantification, or any further restrictions on possible variable assignments. The reason is that in the fragment of TSF domain independent formulae the allowable use of negation can only further restrict the cardinality of the result. Similarly, existential quantification projects out individual columns from relations and thus reduces the size of all tuples included in the resulting relations. Also by using conditions

ID	Class	Formula
ϕ_{Π_1}	Π_1	$trans(t,c) \vee (\blacklozenge_{[0,33)}\ report(t,c))$
$\phi_{\Phi_{1a}}$	Φ_1	$trans(t,c) \vee (\Diamond_{[0,33)}\ report(t,c))$
$\phi_{\Phi_{1b}}$	Φ_1	$trans(t,c)\, \mathcal{U}_{[0,33)}\ report(t,c)$
$\phi_{\Pi_{2a}}$	Π_2	$trans(t,c) \wedge (\blacklozenge_{[0,17)}\ report(t,c) \wedge (\blacklozenge_{[0,17)}\ report(t,c)))$
$\phi_{\Pi_{2b}}$	Π_2	$trans(t,c) \wedge (\blacklozenge_{[0,17)}\ report(t,c) \vee (\blacklozenge_{[0,17)}\ report(t,c)))$
$\phi_{\Phi_{2a}}$	Φ_2	$trans(t,c) \wedge (\Diamond_{[0,17)}\ report(t,c) \wedge (\Diamond_{[0,17)}\ report(t,c)))$
$\phi_{\Phi_{2b}}$	Φ_2	$trans(t,c) \wedge (\Diamond_{[0,17)}\ report(t,c) \vee (\Diamond_{[0,17)}\ report(t,c)))$
$\phi_{\Lambda_{2a}}$	Λ_2	$trans(t,c) \wedge (\blacklozenge_{[0,17)}\ report(t,c) \wedge (\Diamond_{[0,17)}\ report(t,c)))$
$\phi_{\Lambda_{2b}}$	Λ_2	$trans(t,c) \wedge (\Diamond_{[0,17)}\ report(t,c) \wedge (\blacklozenge_{[0,17)}\ report(t,c)))$

Table 6.1: Analyzed formulae and their classes.

that restrict allowable variable assignments, the cardinality of the associated relations can only be restricted. For example, consider the formulae $\phi_1 = \blacklozenge_{[0,16)}\ report(t,c)$ and $\phi_2 = \blacklozenge_{[0,16)}\ report(t,c) \wedge t \prec 10$. It is not difficult to see that the set of satisfying variable assignments is larger for formula ϕ_1 than for formula ϕ_2. By using only disjunction or conjunction in the investigated formulae, we can thus more easily separate the effects caused by our constructions from those caused by specific (non-temporal) characteristics of the formula.

Also observe that the selected formulae do not use the temporal operators $\bullet_I$, $\bigcirc_I$, or $\mathcal{S}_I$. The reason for not considering $\bullet_I$ and $\bigcirc_I$ follows directly from the constructions of the respective auxiliary relations. Because both operators only require taking into account the time point either immediately preceding or succeeding the evaluation time point, the space required to store the respective auxiliary relations mostly depends on the subformula that occurs in the scope of the operator and not on the incremental construction. The reason for omitting $\mathcal{S}_I$ from our analysis is that storing the respective auxiliary relations usually requires less space than in the case of the operator $\blacklozenge_I$. Note that for a tuple $\bar{a} \in \mathbb{N}^n$ to represent a satisfying assignment to the free variables in some formula $\beta(\bar{x})\ \mathcal{S}_I\ \gamma(\bar{x})$ at a time point $i \in \mathbb{N}$, either $\bar{a} \in \gamma^{D_i}$ or $\bar{a} \in \gamma^{D_j}$ at some time point $j < i$ with $\tau_i - \tau_j \in I$ and $\bar{a} \in \beta^{D_k}$ for all $k \in \mathbb{N}$ with $j < k \leq i$. In contrast, for $\bar{a}$ to be a satisfying assignment to the free variables in a formula $\blacklozenge_I\ \gamma(\bar{x})$ at i, we only require $\bar{a} \in \gamma^{D_j}$ at any $j \in \mathbb{N}$ with $j \leq i$ and $\tau_i - \tau_j \in \mathbb{N}$.

Input Generation

We now describe the procedure used to generate the input data for our experiments. First of all, we fixed the signature S_{TPS} to be used for all experiments. For the generation of a timed temporal structure over S_{TPS}, we then proceeded as follows. To generate a single first-order structure over S_{TPS}, for each predicate $r \in \mathsf{R}$, we generated a singleton relation by sampling a pair of values, each value from a predefined sample space using a discrete uniform distribution. As sample spaces, we used $\Omega_{5\times200}$, $\Omega_{5\times100}$, $\Omega_{50\times200}$,

$\Omega_{50\times1000}$, or $\Omega_{1000\times2000}$, where $\Omega_{c\times t}$ denotes the (composite) sample space consisting of a set of c unique customer names (cID) and t unique transaction identification numbers (tIDs), for $c \in \{5, 50, 1000\}$ and $t \in \{200, 1000, 2000\}$. For example, using the sample space $\Omega_{50\times1000}$, we may have generated the singleton relations $\{(819, \text{Megan})\}$ and $\{(449, \text{Barry})\}$ for the predicates $trans$ and $report$, respectively. In a subsequent step, each generated first-order structure over S_{TPS} was associated with a time stamp represented in UNIX time. The time stamp was drawn from the domain of time stamps Ω_τ ranging from 01.01.2005 00:00:00 UTC through 01.01.2009 23:59:59 UTC, again using a discrete uniform distribution. We then repeated this sampling process for the generation first-order structures with associated time stamps 5,000, 10,000, 15,000, 20,000, or 25,000 times. Finally, we ordered the generated timed first-order structures in ascending order according to the time stamps. As a result of this process, we obtained a set of (finite) timed temporal structures over S_{TPS}, where all relations in the same timed temporal structure were singletons containing data tuples from the same sample spaces (i.e., $\Omega_{5\times200}$, $\Omega_{5\times1000}$, $\Omega_{50\times200}$, $\Omega_{50\times1000}$, or $\Omega_{1000\times2000}$) and where each first-order structure was associated with a time stamp ranging from from 01.01.2005 00:00:00 UTC through 01.01.2009 23:59:59 UTC.

Figure 6.2 depicts a part of an input sequence as generated according to the explained procedure using the sample space $\Omega_{50\times1000}$. Intuitively, the generated input sequences can be understood as recorded data streams or log files produced by the TPS. A smaller sample space such as $\Omega_{5\times200}$ represents a setting where a small number of customers repeatedly engages in transactions and where the same transaction identification numbers re-occur several times. In other words, by using the sample space $\Omega_{5\times200}$ to generate an input sequence, the likelihood that two equal tuples appear at two time points with similar time stamps is larger than in the case where a larger sample space is used. A large sample space such as $\Omega_{1000\times2000}$ represents a situation where a large number of distinct customers engages in a large number of distinct transactions. Here, the likelihood of generated violations of our example property is significantly smaller.

Because the formula ϕ_{TPS} and the formulae depicted in Table 6.1 all induce sliding windows with a size of 32 days, the different lengths of the generated timed temporal structures simulate scenarios with different event frequencies. For example, a generated timed temporal structure consisting of 5,000 first-order structures and associated time stamps, which are approximately uniformly distributed over the 4 years induced by Ω_τ, simulates a scenario where, on average, a runtime monitor must take into account approximately 110 past or future time points to evaluate a formula at each time point. In general, a formula with an induced sliding window size of w time units and a timed temporal structure consisting of k first-order structures with time stamps uniformly distributed over a time period of δ time units yield an average relative event frequency of $k\frac{w}{\delta}$. For the generated input sequences with lengths between 5,000 and

Time stamp	trans.tID	trans.cID	report.tID	report.cID
01.01.2005 08:36:22	680	Hilel	995	Dorian
01.01.2005 09:19:50	70	Dana	19	Kasper
01.01.2005 10:45:35	118	Kimberly	940	Fleur
...	...	...	...	...
15.10.2006 10:17:53	753	Ann	23	Zelenia
15.10.2006 13:57:16	133	Farrah	996	Kylan
15.10.2006 14:56:42	819	Megan	449	Barry
...	...	...	...	...
01.01.2009 02:46:38	482	Ivan	213	Arthur
01.01.2009 04:32:23	90	Kyla	146	Adele
01.01.2009 06:35:24	106	Kylie	108	Kyra

Figure 6.2: Example of a finite timed temporal structure with singleton relations over S_{TPS}. Each row in the table represents a first-order structure over S_{TPS} with an associated time stamp. The time stamps were sampled from a time period ranging from 01.01.2005 00:00:00 UTC through 01.01.2009 23:59:59 UTC. The transaction identification numbers (tID) were sampled from a domain of natural numbers ranging from 1 to 1,000 and the customer IDs (cIDs) were sampled from a domain consisting of 50 unique names.

25,000 events and our formulae, we thus obtained 5 scenarios with the (approximate) event frequencies 110, 220, 330, 440, and 550.

Remark 6.2.1. *Because our runtime monitoring approach can be applied in various application contexts, each requiring its own domain-specific signature, we cannot generally know the true probability distributions from which the processed timed temporal structures are sampled. Even for specific application contexts, it may be difficult to realistically estimate the distribution of the data appearing in the timed temporal structures of monitored systems. For our experiments, we thus selected a relatively simple method to generate the input data for our experiments. If, in a real-world application of our monitoring approach, the input data appearing in the processed timed temporal structures are approximately distributed as those generated by the approach described in this section, the results reported in this section are representative of the prospective space consumption of a monitor in such a setting. If, however, the input data appearing in the processed timed temporal structures follows a different distribution, then the results reported in this chapter are not necessarily indicative of the prospective space consumption of our monitor constructions in such a context.*

Measurements

We now describe how we measured the space consumption of our monitors and the size of the relevant active domain.

Measuring space consumption. Similar to Section 5.5, we measured the space consumption of a runtime monitor $\mathcal{M}(\phi)$ by computing the sum of the cardinalities of all auxiliary relations maintained by $\mathcal{M}(\phi)$ at the end of each loop iteration. More specifically, if $i \in \mathbb{N}$ denotes the current time point and $q \in \mathbb{N}$ denotes the next evaluation index with $0 < q \leq i$ (see Figure 4.2), we measured the space consumption of $\mathcal{M}(\phi)$ at the time point i by summing up the cardinalities of all auxiliary relations $p_\alpha^{\hat{D}_j}, r_\alpha^{\hat{D}_j}$, or $s_\alpha^{\hat{D}_j}$ associated with predicates p_α, r_α, or s_α, for all temporal subformulae α of ϕ at all time points $j \in \mathbb{N}$ with $q - 1 \leq j \leq i$. In the special case where $q = 0$, we only summed up the cardinalities of the relations and auxiliary relations at the time points $j' \in \mathbb{N}$ with $q \leq j' \leq i$. In both cases, we also added the cardinality of those relations associated with predicates $r \in \mathsf{R}$ that are maintained to incrementally update the auxiliary relations at the time point i. Recall that the cardinality of a finite relation is the number of unique tuples that it contains. Moreover, observe that the cardinality of a relation only gives a crude estimate regarding the physical space required to store the same information.

We mention that in certain cases we did not take into consideration the cardinality of all auxiliary relations. This was the case whenever the auxiliary relation $p_\alpha^{\hat{D}_j}$ is constructed by projection from the auxiliary relation $r_\alpha^{\hat{D}_j}$, for some temporal subformula α of ϕ. Consequently, for temporal subformulae α of the form $\alpha = \beta \,\mathcal{U}_I\, \gamma$, $\alpha = \Diamond_I \beta$, $\alpha = \blacklozenge_I \beta$, and $\alpha = \beta \,\mathcal{S}_I\, \gamma$, the cardinality of the auxiliary relations $p_\alpha^{\hat{D}_j}$ was not counted separately.

Relevant active domain. For each time point $i \in \mathbb{N}$, we also determined the cardinality of *relevant active domain* (rADOM) with respect to ϕ and (D, τ) at that time point. Intuitively, the relevant active domain is the set of all data constants that appeared in the relevant time window as seen from i. For the given formula ϕ, we can determine the size of this time window by summing up the upper bounds of all metric intervals appearing in ϕ and by taking into account possible operators $\bigcirc_I$ and $\bullet_I$. Observe that whenever the formula ϕ includes a temporal past operator $\mathcal{S}_I$ (or a temporal operator derived from $\mathcal{S}_I$) with $I = [0, \infty)$, then the relevant active domain comprises all data constants from the entire observed prefix. In this case, the relevant active domain and the active domain $adom(D_0, \ldots, D_i)$ as defined in Section 5.2.1 collide. Note that the relevant active domain only provides an idealized benchmark. Specifically, observe that storing the relevant active domain alone does not, in general, provide enough information to correctly evaluate ϕ. The reason is that data constants in the relevant prefix are only counted once and the information about the actual time point(s) at which they were contained in which relation is not maintained.

Example 6.2.2 (Running example)**.** *Consider once more the formula ϕ_{TPS} defining our example property as given in Example 4.3.4. Recall that the original formula is first rewritten into*

the formula

$$trans(t,c) \wedge \big(\blacklozenge_{[0,31)} \exists t'.trans(t',c) \wedge \Diamond_{[0,3)}\ report(t',c)\big) \wedge \neg(\Diamond_{[0,3)}\ report(t,c)),$$

where the temporal operator $\Box$ and the outermost quantifiers have been removed and the remainder of the formula is negated. This formula is then transformed into the formula

$$trans(t,c) \wedge p_{\blacklozenge_{[0,31)} \exists t'.trans(t',c)\wedge\Diamond_{[0,3)}\ report(t',c)}(c) \wedge \neg p_{\Diamond_{[0,3)}\ report(t,c)}(t,c),$$

where the temporal subformulae are replaced by auxiliary predicates.

For each time point $i \in \mathbb{N}$, some associated evaluation index $q \in \mathbb{N}$ with $0 < q \leq i$, and the lookahead index ℓ_{q-1}, we may determine the space consumption of the monitor $\mathcal{M}(\phi_{\text{TPS}})$ by computing

$$\Sigma_{j=q-1}^{i} |r^{\hat{D}_k}_{\blacklozenge_{[0,31)} \exists t'.trans(t',c)\wedge\Diamond_{[0,3)}\ report(t',c)}| + |r^{\hat{D}_j}_{\Diamond_{[0,3)}\ report(t',c)}| + \Sigma_{j=q+\ell_{q-1}-1}^{i} |report^{\hat{D}_j}|.$$

Note that for each temporal subformula α in our running example, the auxiliary relation for p_α is obtained by projection from the auxiliary relation for r_α. Moreover, observe that we only count the auxiliary relation for r_α once for each time point.

Experimental Analysis of Space Consumption

For each of the formulae presented in Table 6.1 as well as for the formula ϕ_{TPS}, we conducted a steady-state analysis as described in Appendix A. For each set of timed temporal structures corresponding to some event frequency scenario, we first determined the length of the warm-up phase by analyzing the evolution of the monitor's space consumption using the approach of Welch [Wel83]. For each formula ϕ and each parameter combination, we then used the method of batch means to determine a point estimate of the steady-state mean space consumption of the runtime monitor $\mathcal{M}(\phi)$. Recall that as parameters we considered the sample spaces $\Omega_{5\times200}, \Omega_{5\times1000}, \Omega_{50\times200}, \Omega_{50\times1000}$, and $\Omega_{1000\times2000}$ as well as the event frequencies 110, 220, 330, 440, and 550. In addition, we estimated the steady-state mean cardinality of the relevant active domain for each parameter combination. For each parameter combination, we finally determined the steady-state mean encoding ratio as the fraction of the steady-state mean space consumption of $\mathcal{M}(\phi)$ and the steady-state mean cardinality of the relevant active domain.

6.2.2 Results

We now present our results. To give an intuitive illustration of the nature of our steady-state results, we first present an example of how a monitor's space consumption evolves over time. We then present the results of a steady-state analysis of the space consumption of the runtime monitor generated for our running example under different parameter settings. Finally, we present the results of a steady-state analysis

of the formulae given in Table 6.1 for those parameter settings with an impact on the space consumption. While the main focus of the analysis was on space consumption, we also present results on the steady-state mean processing time of our monitors.

Example Evolution into Steady State

Figure 6.3 depicts a representative example of how the space consumption of a runtime monitor evolves into steady state. In particular, Figure 6.3(a) shows how the cardinality of the relevant relations and auxiliary relations maintained by the monitor $\mathcal{M}(\phi_{\text{TPS}})$ evolves over time as $\mathcal{M}(\phi_{\text{TPS}})$ incrementally processes a (finite) timed temporal database with singleton relations over S_{TPS}. The processed timed temporal database includes 5,000 linearly ordered first-order structures and was randomly generated from the sample space $\Omega_{50\times1000}$ as described in Section 6.2.1. For each time point, the chart shows the (average) sum of the cardinalities of all relevant relations and auxiliary relations at that time point (denoted by avg(|M|) and |M|, respectively). In addition, Figure 6.3(a) also shows the (average) cardinality of the relevant active domain for ϕ_{TPS} at each time point (denoted by avg(|rADOM|) and |rADOM|, respectively). While the time window induced by ϕ_{TPS} is constant in terms of the maximal difference between relevant time stamps, observe that the cardinality of the relevant active domain fluctuates over time. This is because both the number of structures occurring within the relevant time window as well as the set of unique data values contained in those structures are subject to the randomness induced by the input generation process described in Section 6.2.1. From the average cardinalities of the relations maintained by the monitor and also from the average cardinality of the relevant active domain up to the current time point, in Figure 6.3(a) we can also identify the approximate length of the warm-up phase and thus the time point where $\mathcal{M}(\phi_{\text{TPS}})$ enters the steady state. After about 500 time points, the monitor $\mathcal{M}(\phi_{\text{TPS}})$ seemingly starts to show steady-state behavior. Indeed, using the approach of Welch, we determined the length of the warm-up period to include the first 600 time points for this setting.

Complementing our illustration of the evolution of |M| and |rADOM| from Figure 6.3(a), the histograms in Figures 6.3(b) and 6.3(c) make explicit the frequency of the absolute space required to either store |M| or |rADOM| at each time point after $\mathcal{M}(\phi_{\text{TPS}})$ enters steady-state behavior. While the lower average space consumption of the monitor $\mathcal{M}(\phi_{\text{TPS}})$ in the depicted setting is evident already from Figure 6.3(a), the histograms also make clear how often the relations maintained by $\mathcal{M}(\phi_{\text{TPS}})$ become large. For the analyzed setting, the number of times where the relations maintained by $\mathcal{M}(\phi_{\text{TPS}})$ become large are negligibly rare. Note, however, that this must not hold in other settings.

		Event frequency				
ϕ_{TPS}		110	220	330	440	550
Sample space	$\Omega_{5\times200}$	24.7 ± 0.8	45.4 ± 1.1	65.8 ± 1.3	86.2 ± 1.4	106.7 ± 1.5
	$\Omega_{5\times1000}$	24.6 ± 0.9	45.3 ± 1	65.7 ± 1.2	86.3 ± 1.3	106.8 ± 1.4
	$\Omega_{50\times200}$	24.6 ± 0.9	45 ± 0.8	65.7 ± 1	86.2 ± 1.3	106.8 ± 1.4
	$\Omega_{50\times1000}$	24.8 ± 0.9	45.2 ± 0.8	65.7 ± 1.3	86.2 ± 1.4	106.6 ± 1.5
	$\Omega_{1000\times2000}$	24.6 ± 0.8	44.9 ± 0.7	65.5 ± 0.8	86 ± 1	106.4 ± 1.2

Table 6.2: Point estimates and confidence intervals ($\alpha = 0.05$) of the steady-state mean space consumption of $\mathcal{M}(\phi_{\text{TPS}})$ for different parameter settings.

		Event frequency				
rADOM		110	220	330	440	550
Sample space	$\Omega_{5\times200}$	147.2 ± 3.8	184.5 ± 0.8	214.1 ± 4.3	224.2 ± 7.3	235.5 ± 10.5
	$\Omega_{5\times1000}$	230.8 ± 4.7	393.3 ± 6.8	514.1 ± 6.3	610.4 ± 6.1	685.6 ± 5.9
	$\Omega_{50\times200}$	198 ± 4.2	249.8 ± 2.4	268.6 ± 7.5	269.4 ± 11	292.1 ± 12.3
	$\Omega_{50\times1000}$	288.5 ± 6.6	441.2 ± 5.8	563.6 ± 6.9	655.7 ± 5.9	733.1 ± 5.7
	$\Omega_{1000\times2000}$	460.5 ± 10.7	812.1 ± 11.1	$1,115 \pm 8.6$	$1,358 \pm 14.1$	$1,565.2 \pm 11.8$

Table 6.3: Point estimates and confidence intervals ($\alpha = 0.05$) of the steady-state mean cardinality of the relevant active domain for different parameter settings.

Steady-state Analysis of Running Example

While Figure 6.3 presents a single example of how the space consumed by $\mathcal{M}(\phi_{\text{TPS}})$ may evolve over time, Tables 6.2 and 6.3 as well as Figure 6.4 document the results of our steady-state analysis of $\mathcal{M}(\phi_{\text{TPS}})$'s space consumption for several parameter combinations. Specifically, Figure 6.4(a) presents the point estimates of the steady-state mean space consumption of the monitor $\mathcal{M}(\phi_{\text{TPS}})$ and the point estimates of the steady-state mean relevant active domain for 25 possible parameter combinations. The abscissa represents the event frequency and, in particular, marks the 5 investigated event frequencies as induced by the sets of processed timed temporal structures of varying length. The ordinate represents the total cardinality of all relations maintained by $\mathcal{M}(\phi_{\text{TPS}})$ or the cardinality of the relevant active domain. The chart includes the following graphs. The graph S-s avg(|M|) represents the point estimates of the steady-state mean space consumption of the monitor $\mathcal{M}(\phi_{\text{TPS}})$ for the 5 investigated event frequencies and all the investigated sample spaces $\Omega_{5\times200}$ through $\Omega_{1000\times2000}$. Because for any investigated event frequency the point estimates for the steady-state mean space consumption of $\mathcal{M}(\phi_{\text{TPS}})$ was not significantly different for any of the analyzed sample spaces, S-s avg(|M|) only shows 5 instead of 25 data points. See Table 6.2 for the actual estimates.

In contrast, the 5 graphs denoted S-s avg(|rADOM$_{c\times t}$|) for $c\times t \in \{5\times200, 5\times1000, 50\times200, 50\times1000, 1000\times2000\}$ each present the 5 point estimates of the steady-state mean

		Event frequency				
ϕ	$\Omega_{c\times t}$	110	220	330	440	550
ϕ_{Π_1}	$\Omega_{5\times200}$	113.7	222.8	331	439.3	547.6
	$\Omega_{1000\times2000}$	113.3	222.3	330.6	439.5	547.5
$\phi_{\Phi_{1a}}$	$\Omega_{5\times200}$	337.1	664.6	986.2	1,310	1,633.3
	$\Omega_{1000\times2000}$	335.7	662.2	984.7	1,310	1,632.6
$\phi_{\Phi_{1b}}$	$\Omega_{5\times200}$	448.8	885.5	1,312.1	1,742.7	2,172.6
	$\Omega_{1000\times2000}$	446.9	881.9	1,310.2	1,743.1	2,172.1
$\phi_{\Pi_{2a}}$	$\Omega_{5\times200}$	114.3	223.6	332.9	442	551
	$\Omega_{1000\times2000}$	114.3	223.2	332.6	442.1	550.8
$\phi_{\Pi_{2b}}$	$\Omega_{5\times200}$	3,121.4	11,723.8	25,328.3	43,460.7	65,991.7
	$\Omega_{1000\times2000}$	3,216.1	12,326.8	27,420.7	48,534.7	75,369.4
$\phi_{\Phi_{2a}}$	$\Omega_{5\times200}$	3,405.9	12,751.9	27,896.4	48,952.4	75,883.8
	$\Omega_{1000\times2000}$	3,382.7	12,671.9	27,803.4	48,966.6	75,837.3
$\phi_{\Phi_{2b}}$	$\Omega_{5\times200}$	6,389	24,273	52,689	91,568.3	140,642.2
	$\Omega_{1000\times2000}$	6,482	24,739.2	54,602.7	96,556.7	149,792.5
$\phi_{\Lambda_{2a}}$	$\Omega_{5\times200}$	170.5	334.6	497.3	659.5	821.8
	$\Omega_{1000\times2000}$	170.5	333.5	496.6	659.9	821.4
$\phi_{\Lambda_{2b}}$	$\Omega_{5\times200}$	3,328.9	12,620	27,825.1	48,930.1	75,949
	$\Omega_{1000\times2000}$	3,328.4	12,550.2	27,755.9	48,980.6	75,928.2

Table 6.4: Point estimates of the steady-state mean space consumption of $\mathcal{M}(\phi)$ for different parameter combinations for all formulae ϕ of Table 6.1.

cardinality of the relevant active domain for the respective sample space. For a better comparison of the relative space consumption of the monitor $\mathcal{M}(\phi_{\text{TPS}})$, in Figure 6.4(b) we present the point estimates of the steady-state mean encoding ratios for all the investigated parameter settings.

General Steady-state Analysis

Table 6.4 summarizes the results of our general analysis for the formulae of Table 6.1. Because for a given event frequency, again, the steady-state mean space consumption of the monitors was not significantly different for any of investigated sample spaces, we only provide the results for the sample spaces $\Omega_{5\times200}$ and $\Omega_{1000\times2000}$, where the respective differences were largest. In addition, in Figure 6.5, we depict the results of our analysis for each analyzed formula of Table 6.1 for the setting where $\Omega_{1000\times2000}$ was used as the sample space. As for our running example, we also show the steady-state encoding ratios for all formulae. For ease of readability, Table 6.4 only shows the point estimates. We remark that for all investigated formulae and parameter combinations, the positive and negative increments demarcating the confidence intervals for $\alpha = 0.05$ are all within 4 % of the respective point estimate.

	Event frequency				
Formula	110	220	330	440	550
ϕ_{TPS}	1.2	1.8	2.2	2.6	3.3
ϕ_{Π_1}	2.0	3.2	4.6	6.2	7.7
$\phi_{\Phi_{1a}}$	4.0	8.5	11.2	15.6	20.7
$\phi_{\Phi_{1b}}$	9.9	50.2	112.6	206.2	324.2
$\phi_{\Pi_{2a}}$	1.8	2.8	4.2	5.5	6.8
$\phi_{\Pi_{2b}}$	12.1	48.9	121.7	214.4	343.4
$\phi_{\Phi_{2a}}$	6.1	13.3	16.7	23.2	32.2
$\phi_{\Phi_{2b}}$	17.0	76.2	125.1	274.7	402.3
$\phi_{\Lambda_{2a}}$	3.8	7.8	10.4	14.3	19.2
$\phi_{\Lambda_{2b}}$	17.3	63.4	128.0	231.6	346.2

Table 6.5: Point estimates of the steady-state mean processing time (in milliseconds) of the monitor $\mathcal{M}(\phi)$ for the formula ϕ_{TPS} and the formulae of Table 6.1 for the sample space $\Omega_{1000\times2000}$ and different event frequencies.

Steady-state Processing Times

While the main focus of our experimental validation was on space consumption, Table 6.5 presents results on the processing performance of our monitors in steady state. In particular, it shows point estimates of the steady-state mean processing time of the analyzed monitors to process a single increment (i.e., to process a newly arrived temporal structure and to update all auxiliary relations accordingly). All numbers are given with milliseconds as the time unit. The table reports the point estimates for different event frequencies and the sample space $\Omega_{1000\times2000}$. The processing times for the generated timed temporal structures with the other sample spaces did not differ significantly. All experimental results were obtained on an IBM Thinkpad T60p with an Intel Core Duo T2600/2.16 GHz CPU, 2 GB of RAM, and running Windows XP.

6.2.3 Discussion

We now discuss the reported results and comment on their significance.

Running Example

First of all, let us consider the results of the steady-state analysis for our running example. To begin with, it is interesting to note that the choice of the sample space had a negligible impact on the steady-state mean space consumption of $\mathcal{M}(\phi_{\text{TPS}})$. In contrast, an increase in the event frequency resulted in a linear increase of the mean space consumed by $\mathcal{M}(\phi_{\text{TPS}})$ in steady state. Contrasting the steady-state space consumption of $\mathcal{M}(\phi_{\text{TPS}})$ with the steady-state mean cardinality of the relevant active domain,

we found that under the investigated parameter settings, our monitoring approach resulted in rather low encoding ratios (see Figure 6.4(b)). Because a larger sample space resulted in a larger steady-state mean cardinality of the relevant active domain, the steady-state encoding ratios of runtime monitors for larger sample spaces were smaller than those for small sample spaces. Note that our procedure to generate the input data for our experiments yields a low number of violations. Moreover, keep in mind that our analysis focussed on the steady-state *mean* space consumption and does not assert much about extremes (see Figure 6.3).

General Analysis

The insights gained from the steady-state analysis of the monitor $\mathcal{M}(\phi_{\text{TPS}})$ for our running example also hold for the monitors $\mathcal{M}(\phi_{\Pi_1}), \mathcal{M}(\phi_{\Phi_{1a}}), \mathcal{M}(\phi_{\Phi_{1b}}), \mathcal{M}(\phi_{\Pi_{2a}})$, and $\mathcal{M}(\phi_{\Lambda_{2a}})$, respectively. Under all the investigated parameter settings, the incremental updates of these runtime monitors resulted in a low steady-state mean space consumption and yielded equally low steady-state mean encoding ratios. Furthermore, an increased event frequency yielded a linearly increasing steady-state mean space consumption.

The situation looked differently for the monitors $\mathcal{M}(\phi_{\Pi_{2b}}), \mathcal{M}(\phi_{\Phi_{2a}}), \mathcal{M}(\phi_{\Phi_{2b}})$, and $\mathcal{M}(\phi_{\Lambda_{2b}})$ though. While the use of disjunctions in the temporal subformulae of $\phi_{\Pi_{2b}}$ and $\phi_{\Phi_{2b}}$ naturally resulted in large relations and thus a large steady-state mean space consumption of the monitors $\mathcal{M}(\phi_{\Pi_{2b}})$ and $\mathcal{M}(\phi_{\Phi_{2b}})$, also the monitors $\mathcal{M}(\phi_{\Phi_{2a}})$ and $\mathcal{M}(\phi_{\Lambda_{2b}})$ yielded comparably large auxiliary relations. Given the fact that the formulae $\phi_{\Phi_{2a}}$ and $\phi_{\Lambda_{2b}}$ only include conjunctions, which restrict the cardinality of the final result, this may come as a surprise. The reason for the large cardinality of the respective relations lies in the incremental updating of the auxiliary relations for the nested temporal future subformula. Because this temporal subformula is in the scope of a temporal future subformula itself, the runtime monitor maintains a dynamically determined number of partially built extended structures for each time point between the latest evaluation index and the current event index. The number of such structures is bounded by the maximal number of possible time points that may occur within the interval induced by all the temporal future operators of the respective formula. Recall from Section 5.5 the need for a bound $\ell \in \mathbb{N}$ on the number of equal and adjacent time stamps in the sequence τ in order to obtain a bound on the maximal number of possible time points.

The use of temporal future operators does not necessarily lead to a large space consumption. In spite of the nested future operator in formula $\phi_{\Phi_{2a}}$, the steady-state mean space consumption of the monitor $\mathcal{M}(\phi_{\Phi_{2a}})$ was almost as low as for the monitor $\mathcal{M}(\phi_{\Pi_1})$. Comparing the space consumption of the monitors $\mathcal{M}(\phi_{\Phi_{2a}})$ and $\mathcal{M}(\phi_{\Phi_{2b}})$, we see that the order in which past and future operators are nested had a significant impact on the steady-state mean space consumption of the respective runtime moni-

tor. While the nested use of a future operator in the scope of a temporal past operator had a negligible impact on the steady-state mean space consumption, the inverse case, namely the nesting of a temporal past operator in the scope of a temporal future operator such as $\Diamond_I$ or $\mathcal{U}_I$ resulted in a considerably increased steady-state mean space consumption.

The differences regarding the steady-state mean space consumption of the runtime monitors for the investigated formulae is only partially reflected in the point estimates on the steady-state mean processing times of the respective monitors. Not surprisingly, the worst performance occurred in the setting with the highest event frequency. In particular, in the setting with an event frequency of 550, we estimated the expected steady-state mean processing time (i.e., the time required to incrementally process a newly arrived temporal structure and to update all affected auxiliary relations accordingly) of the monitor $\mathcal{M}(\phi_{\Phi_{2b}})$ to take 0.402 seconds. In other words, in steady state the monitor $\mathcal{M}(\phi_{\Phi_{2b}})$ is expected to take on average 0.402 seconds to perform each incremental updating step. Other runtime monitors with a high steady-state mean space consumption also exhibited an increased steady-state mean processing time. Specifically, this was true for the monitors $\mathcal{M}(\phi_{\Phi_{1b}})$, $\mathcal{M}(\phi_{\Pi_{2b}})$, and $\mathcal{M}(\phi_{\Lambda_{2b}})$, for which we obtained estimates around 0.3 seconds in the setting with the highest event frequency. The monitors $\mathcal{M}(\phi_{\Pi_{2a}})$ and $\mathcal{M}(\phi_{\Phi_{2a}})$, however, demonstrated that despite a high steady-state mean space consumption, the steady-state mean processing time can still be low. Similarly, we estimated the steady-state mean processing times of the runtime monitors $\mathcal{M}(\phi_{\text{TPS}})$, $\mathcal{M}(\phi_{\Pi_1})$, $\mathcal{M}(\phi_{\Phi_{1a}})$, $\mathcal{M}(\phi_{\Phi_{2a}})$, and $\mathcal{M}(\phi_{\Lambda_{2a}})$ to be between 1.2 and 20.7 milliseconds for all investigated event frequencies.

Considering that our implementation is still prototypical and no effort has been spent on the implementation of optimized data structures and other performance optimization techniques, our results suggest that the online monitoring of all investigated formulae can indeed be feasible in many practical settings. Of course, future research will have to further substantiate this conjecture.

While the use of future operators tended to result in a larger steady-state mean space consumption, note that their use might substantially shorten property specifications. As a result, there might be a trade-off between a more succinct specification that uses future operators and a longer specification that relies on only past operators. The former would result in a smaller number of more space consuming auxiliary relations, whereas the latter would result in a larger number of less space consuming auxiliary relations. We emphasize that it remains an open problem to prove whether or how such a succinctness result for MFOTL, similar to the results for propositional temporal logics mentioned in Section 2.4.6, can be established. Also note that some properties expressible within bounded MFOTL may not be expressible using only past operators at all.

Further Remarks

Because the space consumption of our monitoring approach does not only depend on the property under consideration but also on the timed temporal structure that is processed by the respective monitor, the interpretation of our results requires care. In particular, our results only carry over to practical settings where an observed system behavior approximately follows the input distributions that we used in our experiments. In this context, it is also important to note that the characteristics of real systems may change over time and thus might not have stable steady-state distributions. In spite of this, an experimental analysis of the steady-state mean space consumption of an online monitoring algorithm as described in this chapter still constitutes a valuable tool to better understand the performance of the investigated algorithm under well-defined conditions. Specifically, the ability to control individual parameters makes possible a refined sensitivity analysis of the algorithm under scrutiny.

Independent of the above concerns, it would be interesting to compare our results with competing approaches. However, two runtime monitoring approaches are only comparable if their specification languages are equally expressive. If this is not the case, the approach that offers more expressiveness trivially prevails in terms of the properties that can be specified. On the other hand, the approach that provides less expressiveness typically succeeds in terms of the space and time required to decide whether some observed prefix satisfies or violates a given property.

We are not aware of any competing runtime monitoring approach for properties expressed using bounded and TSF domain independent MFOTL formulae. While most other monitoring approaches provide less expressiveness, it might be possible to extend Toman's logical data expiration approach for 2-FOL to also handle 3-FOL formulae [Tom03]. This would provide a benchmark for properties expressed in MFOTL. However, neither an extension of Toman's technique to 3-FOL nor a respective monitoring approach that implements it are currently available. Moreover, as 2-FOL is known to be more expressive than FOTL, a comparison of the relative performance of the two approaches would not necessarily be meaningful either.

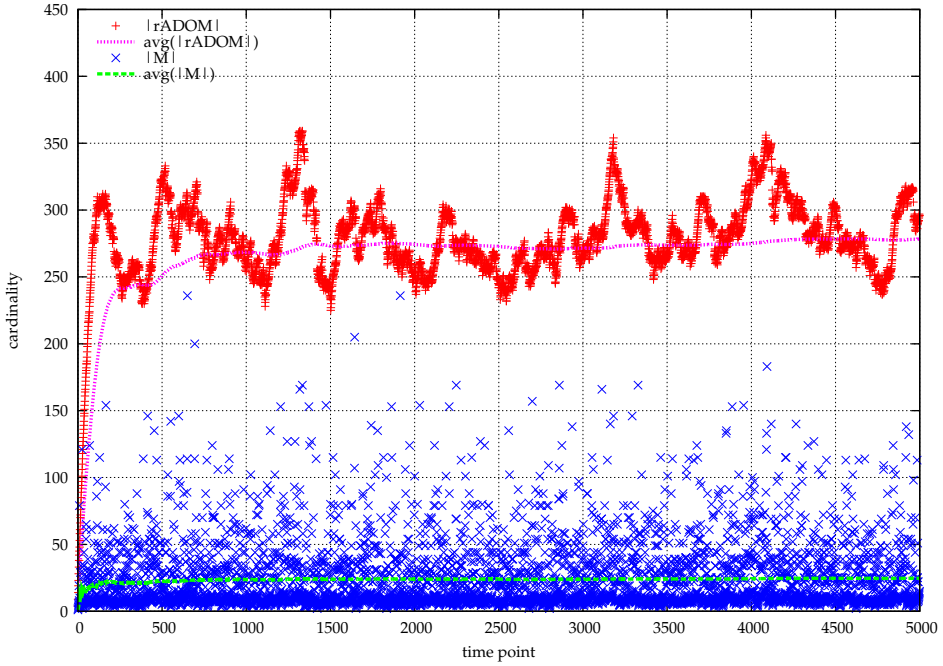
For different reasons, the linear road benchmark [ACG+04], which was proposed to compare different stream processing systems, cannot be used either. One reason is that it focusses on processing performance and scalability. Another reason is that the queries used to run the benchmark require aggregate operators to compute sums or averages of the data elements in the monitored streams. Aggregate operators, however, are not first-order definable and thus not provided by MFOTL.

6.3 Summary and Outlook

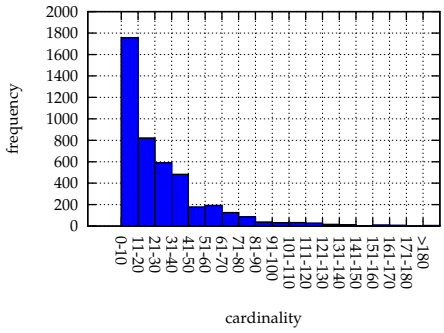
In the first section of this chapter, we briefly described our prototypical runtime monitoring framework. The framework allows for generating runtime monitors for properties specified by bounded and TSF domain independent MFOTL formulae. It permits both the online and offline monitoring of arbitrary system behavior that can be represented in terms of timed temporal structures with finite relations.

In the second section of this chapter, we described how we validated the practical feasibility of our monitoring approach by analyzing the space consumption of the runtime monitors generated by our framework. We first focussed on our running example and investigated the steady-state mean space consumption the monitor $\mathcal{M}(\phi_{\text{TPS}})$ under a variety of parameter settings. Under all the investigated parameter settings, the monitor $\mathcal{M}(\phi_{\text{TPS}})$ showed both a low steady-state mean space consumption and an equally low steady-state mean encoding ratio. This suggests that our monitoring approach could be feasibly applied in a respective monitoring setting. In a second step, we also investigated the space consumption of runtime monitors for representative formulae of several simple but seemingly practical formula classes. All investigated formulae could be handled by our monitor without problems. While most formulae resulted in a low steady-state mean space consumption of the respective monitors, the nesting of temporal future operators yielded a comparably large steady-state mean space consumption. We complemented our analysis with results on the steady-state mean processing times of the investigated runtime monitors. Taken together, our results strongly suggest that the presented runtime monitoring approach is feasible for many practical problems.

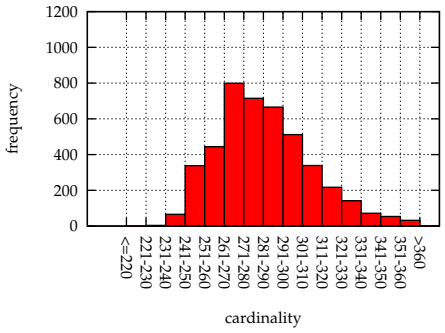
Our results suggest several areas for further study. First of all, it would be interesting to investigate alternative constructions and additional optimization techniques, particularly geared towards a more space-efficient handling of the future operator $\mathcal{U}_I$ and the derived operator $\Diamond_I$. Moreover, it is important to settle the question whether the bounded MFOTL fragment is more expressive than the past-only MFOTL fragment. If the two fragments were equally expressive, it would be natural to investigate to what extent the bounded MFOTL fragment could result in more succinct property specifications. Finally, it will be important to further expand our analysis of the steady-state space consumption of our approach, including a comprehensive investigation of additional formula classes and other input distributions. Once our algorithms and data structures have been further optimized, the analysis should also more strongly focus on other parameters such as the steady-state mean processing time.



(a) Evolution of the (average) cardinality of the relevant active domain and the (average) space consumption of $\mathcal{M}(\phi_{\text{TPS}})$.

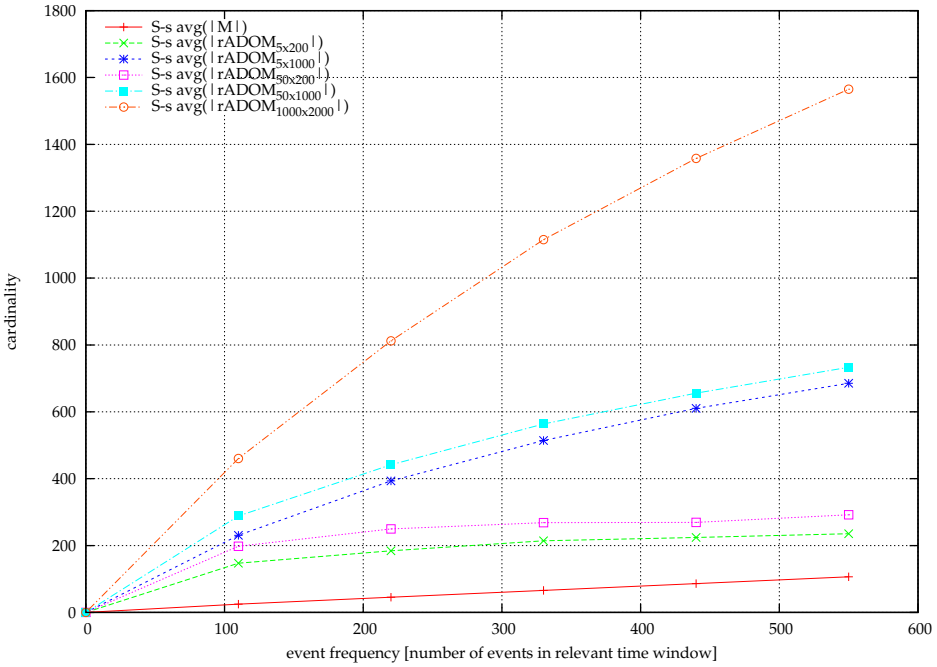


(b) Frequency of cardinalities maintained by $\mathcal{M}(\phi_{\text{TPS}})$ ($n = 4400$).

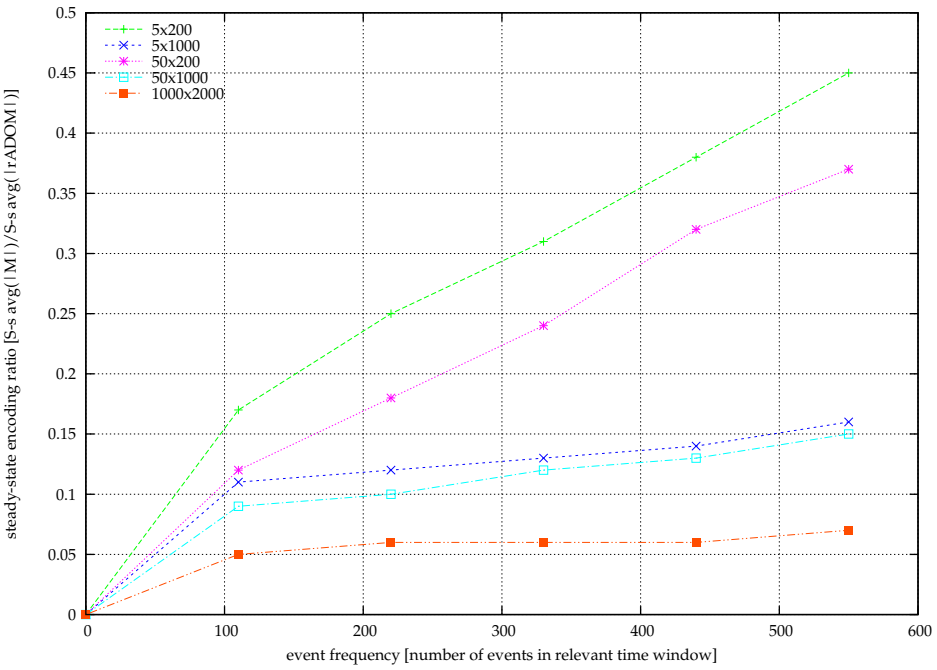


(c) Frequency of cardinalities of rADOM ($n = 4400$).

Figure 6.3: Example evolution of the space consumption of the monitor $\mathcal{M}(\phi_{\text{TPS}})$. The processed timed temporal database consisted of 5,000 first-order structures with singleton relations. Each structure was sampled from the sample space $\Omega_{50\times1000}$ using discrete uniform distributions and was associated with a time stamp from the sample space Ω_τ ranging over the time period from 01.01.2005 00:00:00 UTC through 01.01.2009 23:59:59 UTC.

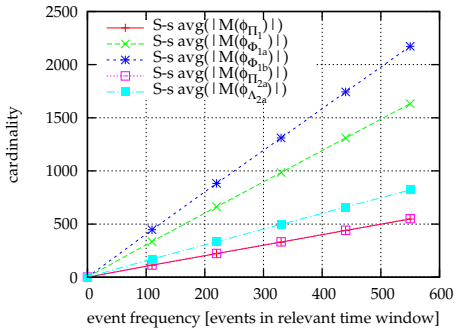


(a) Steady-state mean space consumption of $\mathcal{M}(\phi_{\text{TPS}})$ and size of relevant active domain for different event frequencies and sample spaces.

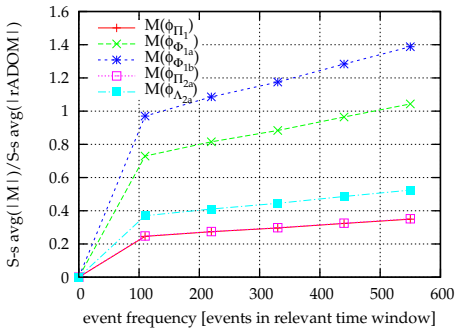


(b) Steady-state mean encoding ratios of $\mathcal{M}(\phi_{\text{TPS}})$ for different event frequencies and sample spaces.

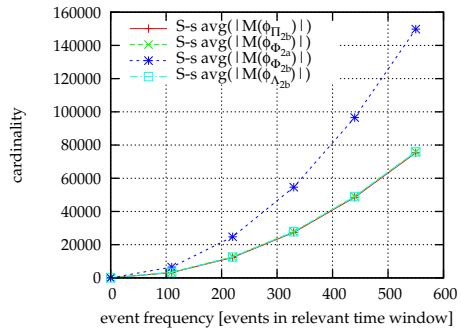
Figure 6.4: Point estimates of the steady-state space consumption of $\mathcal{M}(\phi_{\text{TPS}})$ for different parameter settings.



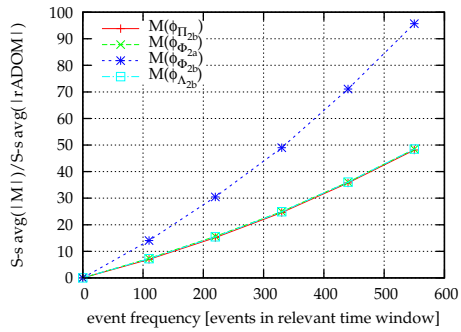
(a) Steady-state mean space consumption of $\mathcal{M}(\phi)$, where $\phi \in \{\phi_{\Pi_1}, \phi_{\Phi_{1a}}, \phi_{\Phi_{1b}}, \phi_{\Pi_{2a}}, \phi_{\Lambda_{2a}}\}$.



(b) Steady-state mean encoding ratios of $\mathcal{M}(\phi)$, where $\phi \in \{\phi_{\Pi_1}, \phi_{\Phi_{1a}}, \phi_{\Phi_{1b}}, \phi_{\Pi_{2a}}, \phi_{\Lambda_{2a}}\}$.



(c) Steady-state mean space consumption of $\mathcal{M}(\phi)$, where $\phi \in \{\phi_{\Pi_{2b}}, \phi_{\Phi_{2a}}, \phi_{\Phi_{2b}}, \phi_{\Lambda_{2b}}\}$.



(d) Steady-state mean encoding ratios of $\mathcal{M}(\phi)$, where $\phi \in \{\phi_{\Pi_{2b}}, \phi_{\Phi_{2a}}, \phi_{\Phi_{2b}}, \phi_{\Lambda_{2b}}\}$.

Figure 6.5: Point estimates of the steady-state space consumption of $\mathcal{M}(\phi)$ and the respective steady-state mean encoding ratios for the formulae of Table 6.1 for the setting with the sample space $\Omega_{1000\times2000}$ and the event frequencies 110, 220, 330, 440, and 550.

"Liberty is obedience to the law
which one has laid down for oneself."

Jean-Jacques Rousseau

Chapter 7

Case Study I: Compliance

Recent years have witnessed a growing amount of regulatory requirements directed towards businesses, particularly publicly traded companies. Prominent examples of such regulations include the Sarbanes-Oxley Act [Sox02], the USA Patriot Act [Pat01], the Bank Secrecy Act [BSA70], and de facto standards such as the International Financial Reporting Standards [Sta08]. To achieve and maintain compliance with an increasing amount of new and constantly changing regulations, organizations must address relevant requirements on different levels [AvKM+06]. On the level of information technology, for example, sensitive information must be adequately protected by means of access control or encryption. Another approach is to continuously monitor regulatory requirements by means of runtime monitoring [GLM+05, GMP06]. To this end, relevant requirements must be formally captured using a sufficiently expressive property specification language. Moreover, a suitable monitoring method for the selected property specification language is required.

In this chapter, we illustrate the use of MFOTL to formally express regulatory requirements. Moreover, we show to what extent the formalized requirements can be monitored using our runtime monitoring approach for finite relations as presented in Chapters 4 and 5.

This chapter is structured as follows. In Section 7.1, we state our objectives and present a set of regulatory requirements that we used in our case study. Section 7.2 then explains how the presented requirements can be formally expressed as bounded and TSF domain independent MFOTL formulae. In Section 7.3, we investigate to what extent the monitoring of the formalized requirements is practically feasible. We conclude the chapter with a summary in Section 7.4.

7.1 Overview

We first clarify the objectives of the case study described in this chapter. We then present a set of example requirements, which we used to conduct our case study.

7.1.1 Objectives

The objectives of this case study were twofold:

1. Determine to what extent regulatory requirements can be expressed as bounded and TSF domain independent MFOTL formulae.
2. Investigate whether monitoring the resulting formulae is practically feasible.

To address the first objective, we investigated whether a set of example requirements could be formally captured using bounded and TSF domain independent MFOTL formulae. Naturally, we only considered regulatory requirements that restrict the organizational behavior as reflected on the level of information technology (IT).

To address the second objective, we first investigated the relationship of the formalized requirements to the formula classes that we analyzed in the previous chapter. Moreover, we performed additional experiments similar to those described in the previous chapter.

7.1.2 Example Requirements

Figure 7.1 depicts a set of regulatory requirements. The requirements closely resemble but are a significantly shorted version of those made on financial institutions under the USA Patriot Act [Pat01], Section 326, in particular the Code of Federal Regulations (CFR) that implements it, 31 CFR paragraph 103.121.

(1) *A bank must obtain the following information prior to opening an account: name; date of birth; residential address; identification number.*
(2) *The bank must verify the identity of each customer, using the information obtained in accordance with the above requirements, within a reasonable time after the account is opened.*
(3) *The bank must close an account, after attempts to verify the customer's identity have failed.*
(4) *The bank must retain a record of all information obtained according to the above requirements.*
(5) *The bank must retain the recorded information for two years after an account is closed.*

Figure 7.1: Regulatory requirements inspired by 31 CFR 103.121.

Note that the requirements depicted in Figure 7.1 leave room for interpretation. For example, the second requirement uses the term "reasonable time" to qualify the amount of time that may elapse between the opening of a new account and the verification of the identify of the associated customer. The use of such vague terms is typical of most regulations. Their meanings normally depend on the type and size of the affected organization, the industry and country of operation, or the technological possibilities and must usually be determined by a legal domain expert. Because the formalization of the requirements requires concrete instantiations for all these terms, we made arbitrary assumptions with respect to their meaning. We emphasize that the selected interpretations do not influence the results of our case study.

Because the regulatory requirements in Figure 7.1 concern different banks, each of a different size, structure, etc., the regulatory text only assumes a small set of general concepts that each bank is requirement to implement. For example, each bank is expected to have a specific representation for accounts and each bank is required to collect certain customer information attributes. Moreover, a bank is expected to be able to execute certain actions on their accounts or customer information records. For example, banks are expected to be able to open and close accounts as well as to obtain, retain, verify, and discard customer information associated with these accounts. The regulation does not assume specific implementations for these objects or actions. Instead, it merely states general constraints that any implementation must satisfy.

In our formalization, we only focus on those aspects of the regulatory requirements that are necessary for monitoring. In particular, we treat the specific implementations of accounts, customer records, etc. as black boxes. We assume that the IT components that implement the respective functionality are instrumented in such a way that all necessary events can be observed and are reliably communicated to the runtime monitor.

7.2 Formalization of Regulatory Requirements

We now explain how the requirements from Figure 7.1 can be formalized as bounded and TSF domain independent MFOTL formulae. To this end, we first define an appropriate signature. We then formalize the requirements as formulae over this signature.

Because regulatory requirements are typically managed by each bank alone, our formalization is from the perspective of a single bank. An extension of our formalization to the case where a third party (e.g., an auditing company) continuously monitors a set of banks is straightforward.

7.2.1 Signature Definition

For our formalization, we use the signature $S_{\text{CFR}} = (\mathsf{R}, \mathsf{C}, a)$, where $\mathsf{R} = \{open, close, obtainInfo, retainInfo, discardInfo, verifyID\}$, $\mathsf{C} = \{failed\}$, and

$$a(r) = \begin{cases} 1 & \text{if } r \in \{open, close, obtainInfo, retainInfo, discardInfo\}, \text{ and} \\ 2 & \text{if } r \in \{verifyID\}. \end{cases}$$

The predicates in R are used to refer to all the events that are relevant in the monitored setting. As usual, we interpret formulae over S_{CFR} with respect to timed temporal structures over S_{CFR} and an infinite domain $|D|$, consisting of the natural numbers $\mathbb{N}$ (representing unique account identifiers) and a dedicated value $failed^{|D|}$. Every observed S_{CFR}-structure represents the occurrence of a set of events (i.e., relations). The events associated with $open$ and $close$ occur whenever an account is opened or closed,

respectively. The event associated with $obtainInfo$ signals the fact that a valid record of customer information has been obtained. By valid we mean that the obtained values for the attributes name, date of birth, residential address, and identification number are all meaningful. The event associated with $retainInfo$ occurs whenever the bank starts to retain a record of customer information associated with an account. In contrast, the event $discardInfo$ indicates that an existing customer record has been deleted. Finally, the event $verifyID$ occurs whenever the bank has verified the identity of a customer associated with an account using the previously obtained customer information. Its second attribute indicates whether the verification of a customer's identify has failed (denoted by $failed^{|D|}$). Because for a given bank all requirements concern individual accounts, we can use a single unique identifer in predicates to correlate associated events.

7.2.2 Formalized Requirements

We now present how we formalized the requirements from Figure 7.1 as MFOTL formulae over S_{CFR}. In all subsequent formulae, metric constraints are interpreted using days as the time unit.

Requirement (1)

Formula (7.1) expresses the requirement that before a new account is opened, at least the name, the date of birth, the address, and an identification number of the customer must be obtained.

$$\Box \forall a.\, open(a) \rightarrow \big(\blacklozenge\, obtainInfo(a) \big) \tag{7.1}$$

Recall that we assume that the predicate $obtainInfo$ only evaluates to true in case a valid and complete set of customer attributes has been obtained.

Requirement (2)

Formula (7.2) states that whenever the bank has opened a new account, it must verify the identity of the associated customer using the previously collected customer information within a reasonable time.

$$\Box\ \forall a.\, open(a) \rightarrow \big(\Diamond_{[0,3)}\, \exists result.\, verifyID(a, result) \big) \tag{7.2}$$

In our formalization, we interpreted the term "reasonable time" to mean at most 2 days. Note that for Formula (7.2) to be satisfied by a timed temporal structure, the outcome of the customer verification does not matter.

Requirement (3)

Formula (7.3) models the requirement that the bank must close a tentatively opened account whenever it cannot successfully verify a customer's identity.

$$\Box\ \forall a.\mathit{verifyID}(a, \mathit{failed}) \rightarrow \big(\ \Diamond_{[0,4)}\ \mathit{close}(a)\big) \tag{7.3}$$

In our formalization, we required that the account must be closed within three days.

Requirement (4)

Formula (7.4) expresses that a bank has to retain the record of obtained customer information.

$$\Box\ \forall a.\mathit{obtainInfo}(a) \rightarrow \big(\ \blacklozenge_{[0,2)}\ \mathit{retainInfo}(a)\big) \tag{7.4}$$

Because it would be too strict to require that the event that signals the successful gathering of customer information and the event that denotes the start of the retention process occur at exactly the same time, we allow for the retention process to be started within a day before the complete customer record was obtained.

Requirement (5)

Formula (7.5) expresses the constraint that a customer record may only be deleted if the corresponding account has been closed for at least two years (i.e., 730 days).

$$\Box \forall a.\mathit{discardInfo}(a) \rightarrow \big(\ \blacklozenge_{[730,\infty)}\ \mathit{close}(a)\big) \tag{7.5}$$

7.2.3 Discussion

Formulae (7.1)–(7.5) demonstrate that compliance requirements can be expressed as MFOTL formulae of the form $\Box\,\phi$, where ϕ is bounded and TSF domain independent. Hence, we can use our runtime monitoring algorithm for finite relations to check whether an observed timed temporal structure violates the any of these formulae.

Our formalization relies on several assumptions. In particular, we assume that all relevant events can be observed and that no events are duplicated, lost, or delayed. To this end, we require trustworthy signalling mechanisms on the part of the systems that implement and manage the respective business functionality (e.g., to open or close accounts and store or delete associated customer information). Moreover, we require a reliable communication infrastructure. To what extent these assumptions can be realized in practice does not concern us here.

7.3 Monitoring Compliance Requirements

In this section, we investigate the practical feasibility of our method for monitoring the formalized compliance requirements presented in the previous section.

We start with some observations. Subsequently, we describe additional experiments and discuss their results.

7.3.1 Observations

Observe that Formulae (7.1)–(7.5) are instances of two formula classes that we analyzed in Section 6.2. In particular, Formulae (7.1), (7.4), and (7.5) are instances of the class Π_1 and Formulae (7.2) and (7.3) are instances of the class Φ_1. Moreover, Formulae (7.2), (7.3), and (7.4) only finitely reach into the past or the future.

Recall that for monitoring, the outermost temporal operator as well as the universal quantifier are removed and the remaining formula is negated. See Section C.1 for the resulting formulae in concrete syntax. Except for the different predicates, the respective arities, the metric constraints, and the use of conjunctions instead of disjunctions, the formulae as used for monitoring are almost identical to the formulae ϕ_{Π_1} and $\phi_{\Phi_{1a}}$ from Table 6.1. As we measure space consumption by the cardinality of relevant relations and because we focus on (relative) event frequencies, neither the arity of the predicates nor the absolute size of the metric constraints play a role in our analysis. Moreover, whether we use a conjunction or a disjunction in formulae of the classes Π_1 and Φ_1 has no impact on the size of the maintained auxiliary relations and does not significantly affect the processing times in the analyzed setting either.

As a result of the above considerations, we can use the results from the analysis of the formulae ϕ_{Π_1} and $\phi_{\Phi_{1a}}$ from Table 6.1 as an indication of the feasibility of monitoring these formulae. In particular, we can carry over the result that the steady-state mean space consumption of the respective runtime monitors increases linearly with the frequency of events in the relevant time window (under the assumptions used in the experiments of Section 6.2). Similarly, we can carry over that the respective runtime monitors have a low steady-state mean processing time, which means that relevant events can be processed efficiently. From these observations, we expect the (online) runtime monitoring of Requirements (2)–(4) to be feasible in practical settings where events occur with event frequencies and distributions similar to the analyzed setting.

While we have not performed experiments for event frequencies higher than 550, from our constructions in Section 4.4 and the experimental results from Section 6.2, we expect that both the steady-state mean space consumption and the steady-state mean processing time increases linearly also for higher event frequencies. This strongly suggests that the Requirements (2)–(4) can be efficiently monitored also in settings where more than 550 new accounts are opened or verified within the 3 days assumed above.

Because their temporal operators may refer infinitely far into the past, the situation

is different for the Formulae (7.1) and (7.5). Specifically, the space consumption of the runtime monitors generated for these two formulae may increase infinitely with the number of processed structures. This can be seen as follows. Recall that each structure over the signature S_{CFR} represents a set of compliance-relevant events occurring at a given time point and with an associated time stamp. As time progresses, we expect the monitored bank to open new accounts for new customers, which are associated with new unique identifiers. Consequently, as the number of already processed identification numbers increases, the cardinality of the auxiliary relations maintained for the temporal subformulae $\blacklozenge\, obtainInfo(a)$ and $\blacklozenge_{[730,\infty)}\, close(a)$ in Formulae (7.1) and (7.5), respectively, increases inevitably.

We emphasize that this increase is due to the specific formalization of the requirement and the semantics of the formula. Specifically, it is not caused by our monitoring approach. Indeed, as shown in Section 5.5 the space consumption of our runtime monitors is polynomially bounded by the cardinality of the data that appeared in the processed prefix. In other words, the space consumption of our runtime monitors depends on the data elements in the processed prefix and not on the length of the prefix.

7.3.2 Experimental Analysis

To assess to what extent the Formulae (7.1) and (7.5) can nonetheless be monitored, we performed further experiments. In the following, we write $\phi_{R(1)}$ for Formula (7.1) and $\phi_{R(5)}$ for Formula (7.5).

Setup of the Experiments

The setup of our experiments was analogous to those described in Section 6.2.1.

Generation of input data. Because of the lack of realistic input data, we generated (finite) timed temporal structures over S_{CFR} using a procedure analogous to the one described in Section 6.2.1. Each structure was sampled from the sample space $\Omega_{25{,}000^6 \times 2}$. Specifically, for each of the 6 predicates in R, a singleton relation was randomly created by instantiating one of 25,000 unique account identifiers with equal probability. For the predicate $verifyID$, the generated relation also contained the dedicated value $failed^{|D|}$ in 10% of the cases. As described in Section 6.2.1, we also used the sample domain Ω_τ for generating time stamps from 01.01.2005 UTC through 01.01.2009 UTC.

Measurements. As described in Section 6.2.1, we measured the space consumption of the generated monitors by the cardinality of the relevant auxiliary relations. In addition, we calculated the active domain (ADOM) from all relations with associated predicates appearing in the formula. Moreover, for selected time points, we also measured

	Number of processed events							
Formula	500	1,000	2,500	5,000	10,000	15,000	20,000	25,000
$\phi_{R(1)}$	492	986	2,381	4,535	8,351	12,442	15,721	18,842
$\phi_{R(5)}$	499	992	2,456	4,911	9,862	13,858	17,721	20,828

Table 7.1: Average space consumption of $\mathcal{M}(\phi_{R(1)})$ and $\mathcal{M}(\phi_{R(5)})$ for monitoring a timed temporal S_{CFR}-structure as computed from 10 independent replications of the experiment.

	Number of processed events							
Formula	500	1,000	2,500	5,000	10,000	15,000	20,000	25,000
$\phi_{R(1)}$	1.5	4	14	56	146	533	1,620	2,242
$\phi_{R(5)}$	1.2	2.9	10	33	132	410	1,515	1,840

Table 7.2: Average time (in seconds) required by $\mathcal{M}(\phi_{R(1)})$ and $\mathcal{M}(\phi_{R(5)})$ for monitoring a timed temporal S_{CFR}-structure as computed from 10 independent replications of the experiment.

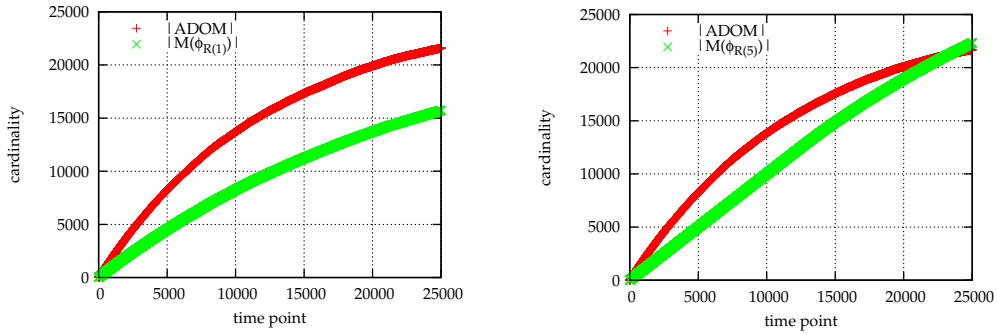
the time required to process a timed temporal structure up to that time point. Specifically, we measured after 500, 1000, 2,500, 5,000, 10,000, 15,0000, 20,000, and 25,000 time points.

Analysis and Results

We used our runtime monitoring framework to generate the monitors $\mathcal{M}(\phi_{R(1)})$ and $\mathcal{M}(\phi_{R(5)})$ and to process 10 independently generated timed temporal S_{CFR}-structures as explained above. Table 7.1 gives the average cardinalities of the relevant auxiliary relations maintained by $\mathcal{M}(\phi_{R(1)})$ and $\mathcal{M}(\phi_{R(5)})$ after having processed prefixes of the generated timed temporal structures of different lengths up to at most 25,000 time points. Similarly, Table 7.2 presents the average times required to process a prefix of the generated timed temporal structures up to at most 25,000 time points.

The results show that the space consumption of the monitors $\mathcal{M}(\phi_{R(1)})$ and $\mathcal{M}(\phi_{R(5)})$ increases linearly as they incrementally process finite prefixes of timed temporal S_{CFR}-structures. Because the procedure used to generate the input data, in particular, because of the sample space used to generate the data elements contained therein, the space consumption does not increase with time but with the cardinality of the data elements in the processed prefix. An example of this phenomenon is illustrated in Figure 7.2. Specifically, Figures 7.2(a) and Figures 7.2(b) depict example evolutions of the cardinality of the active domain and the cardinality of the auxiliary relations maintained by the monitors $\mathcal{M}(\phi_{R(1)})$ and $\mathcal{M}(\phi_{R(5)})$, respectively, as each monitor processes a continuously growing prefix of a timed temporal S_{CFR}-structure.

While the space consumption of the analyzed monitors increases linearly with the



(a) Space consumption of $\mathcal{M}(\phi_{R(1)})$.

(b) Space consumption of $\mathcal{M}(\phi_{R(5)})$.

Figure 7.2: Example evolution of the space consumption of the monitors $\mathcal{M}(\phi_{R(1)})$ and $\mathcal{M}(\phi_{R(5)})$. The processed timed temporal structures consisted of 25,000 first-order structures over S_{CFR}. Each structure was sampled from the sample space $\Omega_{25{,}000^6 \times 2}$ using discrete uniform distributions. Moreover, each structure was associated with a time stamp from from the sample space Ω_τ.

number of relevant data elements in the processed prefix, the required time to incrementally update the maintained auxiliary relations and to check for violations of the monitored formula increases more rapidly (with a power of about 1.25). This is due to the increasing size of the maintained auxiliary relation, which means that an ever larger amount of data has to be checked for a matching variable assignment. While the current implementation of the function that performs this check is rather naive, a more sophisticated incremental updating strategy should improve the performance of our monitors. In particular, by incrementally indexing new tuples that are added to the currently maintained auxiliary relation, we expect to obtain significantly improved average processing times.

7.3.3 Discussion

By relating our formalizations to the formulae analyzed in the previous chapter, we found three of the five formalized requirements to be directly suitable for online runtime monitoring using the approach presented in the previous chapters. For the settings analyzed in Section 6.2, the estimated performance of the respective runtime monitors is identical to the results obtained for the formulae ϕ_{Π_1} and $\phi_{\Phi_{1a}}$, respectively, from Table 6.1.

From our constructions and the results of our analysis, the space and time required for monitoring the Formulae (7.1) and (7.5) can become infeasible. The reason for this lies in the nature of the formalization and the data elements contained in the processed timed temporal structure and not in our monitoring method. As result, online monitoring arbitrary timed temporal structures with respect to these formulae is not feasible in general. We point out that the monitoring of these requirements may still be feasibly

applied in an offline scenario, where the length of the processed log file is bounded and the analysis is performed in batch mode.

Alternatively, other formalizations of the respective requirements can be used. For example, by associating also the temporal past operators with bounded intervals, online runtime monitoring of the resulting formulae will be feasible under the same assumptions as made in the Section 6.2.

7.4 Summary and Outlook

In this chapter, we demonstrated how regulatory requirements can be formally expressed by means of bounded and TSF domain independent MFOTL formulae. Three of the five formalized requirements were found to be directly suitable for online runtime monitoring using the approach presented in the previous chapters. The formalizations of the two remaining requirements made use of non-metric temporal past operators, which means they can refer to time points infinitely far in the past. While our runtime monitoring approach can also be used to check observed prefixes of timed temporal structures for violations of these two requirements, the space and time requirements to do so may grow arbitrarily large. Even though the respective space consumption is linearly bounded by the data elements occurring in the processed prefix, the absolute space required may continue to grow indefinitely. This prevents these requirements from being feasible for online runtime monitoring.

Regarding future work, given that many real-life regulatory requirements that are suitable for runtime monitoring can be formally expressed as formulae of the classes Π_1 and Φ_1, it would be interesting to investigate to what extent our runtime monitoring approach can be further optimized for monitoring formulae of these classes.

While we have only focussed on the actual monitoring, many practical aspects remain open before we can apply our runtime monitoring approach to the monitoring of regulatory requirements in a real setting. Because the manual instrumentation of IT components to signal relevant events to the runtime monitor, it would be interesting to investigate to what extent existing instrumentation approaches can be extended to our setting. Furthermore, there is the question of pre-defining an appropriate (re-)action to be executed whenever a violation of a monitored formula occurs. For example, the runtime monitor could notify another component and write the elements that caused the violation to a write-one-read-many and tamper-proof audit log file. Here, existing work on trigger languages in the area of active databases may provide a promising starting point.

"I generally avoid temptation unless I can't resist it."

Mae West

Chapter 8

Case Study II: Separation of Duty

Separation of duty (SoD) is an organizational security principle to prevent fraud and errors by distributing privileges among users in security-critical workflows. Over the years, different variants of separation of duty have been identified and various formalisms for specifying the respective SoD constraints have been proposed (e.g., [San88, Kuh97, GGF98, LW08]). The problem of dynamically monitoring or enforcing formalized SoD constraints, however, has received less attention (e.g., [SZ97, BBK09]).

In this chapter, we show how dynamic object-based separation of duty constraints can be expressed as past-only and TSF domain independent MFOTL formulae. Moreover, we further validate our runtime monitoring approach for finite relations by showing how it can be used to monitor the resulting constraints.

The chapter is structured as follows. In Section 8.1, we clarify our objectives, introduce the concept of role-based access control, and describe commonly-distinguished variants of SoD constraints. In Section 8.2, we demonstrate how different variants of SoD, in particular, dynamic object-based separation of duty, can be formalized by means of past-only and TSF domain independent MFOTL formulae. In Section 8.3, we present how we evaluated whether monitoring the formalized constraints is practically feasible. We conclude the chapter with a summary in Section 8.4.

8.1 Overview

We first state the objectives of the presented case study and briefly explain our approach. We then introduce role-based access control as the foundation on which we express separation of duty constraints. Subsequently, we summarize common variants of separation of duty.

8.1.1 Objectives

The objectives of this case study were as follows:

1. Determine to what extent different variants of separation of duty, in particular, dynamic object-based separation of duty, can be expressed as past-only and TSF domain independent MFOTL formulae.
2. Investigate the practical feasibility of monitoring the resulting formulae using our runtime monitoring approach for finite relations.

To address the first objective, we provide a formalization of different types of separation of duty constraints in the context of role-based access control. To address the second objective, we performed experiments similar to those described in Chapters 6 and 7.

8.1.2 Role-based Access Control

Separation of duty is usually studied and formalized in the context of role-based access control (RBAC) [San88, NP90, SZ97, Kuh97, GGF98, AS99, SM04, SSW05, LW08]. In (flat) RBAC, access to resources is controlled by assigning users to sets of roles, where each role is associated with a set of permissions. A user acquires a set of permissions by being assigned to one or many roles.

Formally, RBAC consists of the following components [SCFY96, SFK00, FSG+01]:

U : A set of *users*.

R : A set of *roles*, i.e., a job function with specific authority and responsibility.

A : A set of *actions*, i.e., a mode of access.

O : A set of *objects*, i.e., the resources involved in access control decisions.

$\mathsf{UA} \subseteq \mathsf{U} \times \mathsf{R}$: A *user assignment* relation that assigns users to roles.

$\mathsf{PA} \subseteq \mathsf{R} \times \mathsf{A} \times \mathsf{O}$: A *permission assignment* relation that assigns roles to sets of action-object pairs (also called permissions).

S : A set of *sessions*, i.e., abstract entities through which users interact with the system.

$user : \mathsf{S} \to \mathsf{U}$: The function $user$ maps each session $s \in \mathsf{S}$ to the single user $user(s)$ (constant for the lifetime of a session).

$roles : \mathsf{S} \to 2^{\mathsf{R}}$: The function $roles$ maps each session $s \in \mathsf{S}$ to a set of roles $roles(s) \subseteq \{r \mid (user(s), r) \in \mathsf{UA}\}$ (can change over time). The session s has the permissions $\bigcup_{r \in roles(s)} \{p \mid (p, r) \in \mathsf{PA}\}$.

The tuple $(\mathsf{UA}, \mathsf{PA})$ is called an RBAC *configuration*. An RBAC configuration defines which user is assigned to which roles and which roles are assigned to which action-object pairs. This statically restricts which users can possibly activate which roles and thereby acquire the associated permissions.

RBAC Administration

RBAC configurations can be modified by means of administrative actions. In particular, new users can be added to or removed from the set U, new roles can be added to or removed from the set R, and new actions or objects can be added to or removed from

the sets A or O, respectively. Furthermore, user assignments or permission assignments can be added to or removed from the relations UA or PA, respectively.

Sessions and Role Activation

A session represents the abstract entity through which a user interacts with an access-controlled system. At execution time, a user may create new or terminate running sessions. For each running session, a user can *activate* any subset of roles that are assigned to him in the current RBAC configuration. Hence, a session is a mapping of a single user to a set of roles. While active roles may be deactivated explicitly, it is often assumed that, once activated, roles remain active until the end of the associated session.

Use of Roles and Permissions

As a user engages in a session, he may *use* all the permissions that are associated with the activated roles in that session. In real systems, role activation is often done implicitly, i.e., it is activated automatically whenever a user desires to use a permission that is associated with a role which is statically assigned to him in the current RBAC configuration. For example, in a process management context, whenever a new permission is required to execute the next task in a running process instance and the associated user has the necessary permission through a static role assignment, the required role can be automatically activated and the use of the respective permission to execute the task will be allowed.

8.1.3 Separation of Duty

In the context of RBAC, separation of duty constraints are usually specified and enforced by means of mutually exclusive roles. Violations of separation of duty occur in the presence of incompatible roles. A pair of roles can be considered incompatible in the following three situations:

1. A user is statically assigned to a pair of mutually exclusive roles.
2. A user activates a pair of mutually exclusive roles under specific conditions (e.g., simultaneously, in the same session, or in relation to the same object).
3. A user executes permissions associated with two mutually exclusive roles under specific conditions (e.g., simultaneously, in the same session, or in relation to the same object).

These situations give rise to different variants of separation of duty. Constraints on static role membership are used to define *static separation of duty*, whereas constraints on role activation or the use of associated permissions define *dynamic separation of duty*.

Static Separation of Duty

Static separation of duty (SSoD) represents the simplest notion of SoD [SZ97, SSW05]. To ensure static separation of duty *no user may be a member of any two exclusive roles*. Violations of SSoD constraints can be statically prevented by ensuring that in a given RBAC configuration no user is assigned to a pair of mutually exclusive roles.

We point out that SSoD is a very strict and rather impractical notion. In particular, preventing users from being assigned to two conflicting roles may not be practically feasible in small organizations.

Dynamic Separation of Duty

Dynamic separation of duty is a more practical notion that provides more flexibility. The following types of dynamic SoD are commonly distinguished [SZ97, SSW05].

Simple dynamic separation of duty. Simple dynamic separation of duty (SDSoD) is the simplest form of dynamic separation of duty. It states that *a user may be a member of any two exclusive roles but must not activate them both at the same time*. While the term 'at the same time' leaves room for some interpretation, it is often taken to mean in the same session.

Object-based separation of duty. In object-based separation of duty (ObSoD), *a user may be a member of any two exclusive roles and may also activate them both at the same time, but he must not act upon the same object through both*. Again, the term 'at the same time' admits several interpretations. It often means in the same session, too.

Operational separation of duty. In operational separation of duty (OpSoD), *a user may be a member of some exclusive roles as long as the set of permissions acquired by activating these roles does not permit him to execute every step of a workflow*. Note that a user may activate conflicting roles in different (e.g., parallel or consecutive) sessions.

History-based separation of duty. Finally, history-based separation of duty (HbSoD) combines ObSoD and OpSoD to yield: *A user may be a member of some exclusive roles and the complete set of permissions acquired over these roles may cover an entire workflow, but the user must not use these permissions to perform all workflow steps on the same object.*

We point out that several variations of HbSoD have been proposed. An example are the history-based separation (and binding) of duty constraints presented by Sandhu and formalized using his transaction control expressions [San88].

8.2 Formalization of Separation of Duty Constraints

We now demonstrate how different types of SoD constraints can be formally expressed using MFOTL. We first present an appropriate MFOTL signature with dedicated predicates to capture the previously presented RBAC components. We also provide a set of syntactically derived state predicates that closely mimic the intuitive semantics of RBAC. We then present our formalization of different types of separation of duty.

8.2.1 Signature Definition

To formalize SoD constraints as MFOTL formulae over point-based (timed) temporal structures, we define the signature $S_{\text{SoD}} = (\mathsf{R}, \mathsf{C}, a)$ with $\mathsf{R} = \{U_S, U_F, R_S, R_F, A_S, A_F, O_S, O_F, UA_S, UA_F, PA_S, PA_F, S_S, S_F, user_S, user_F, roles_S, roles_F, X_S, X_F, PD_S, PD_F, exec\}$, $\mathsf{C} = \emptyset$, and

$$a(r) = \begin{cases} 1 & \text{if } r \in \{U_\star, R_\star, A_\star, O_\star, S_\star, PD_\star\}, \\ 2 & \text{if } r \in \{UA_\star, user_\star, roles_\star, X_\star\}, \text{ and} \\ 3 & \text{if } r \in \{PA_\star, exec\}, \end{cases}$$

where $\star \in \{S, F\}$.

The predicates in R represent all events that are relevant to monitor different variants of separation of duty in an RBAC context. Formulae over S_{SoD} are interpreted with respect to (timed) temporal structures over S_{SoD} and an infinite domain $|D|$, consisting of the natural numbers $\mathbb{N}$.

The set R includes a pair of predicate symbols for each set or function in RBAC. For example, for the set of users U, it provides the predicate symbols U_S and U_F. These predicates represent the set of users that are added to or removed from the set of RBAC users U at a given time point. Likewise, the predicates $R_\star$, $A_\star$, $O_\star$, $UA_\star$, $PA_\star$, and $S_\star$ for $\star \in \{S, F\}$ represent events providing the sets of tuples that are added to or removed from the respective RBAC sets at a given time point. The predicates $user_S$ and $user_F$ indicate which user has started a new session or whether a user session has just terminated, respectively, at a given time point. Similarly, the predicates $roles_S$ and $roles_F$ represent the activation or deactivation of a role in a session.

Furthermore, the predicates $X_\star$, $PD_\star$, for $\star \in \{S, F\}$, and $exec$ allow for capturing SoD requirements. The predicates X_S and X_F represent pairs of roles that start or stopped to be considered mutually exclusive and the predicate pair PD_S and PD_F define the sets of actions required to execute a particular workflow. Finally, the predicate $exec$ specifies which actions are executed on which object in which session.

To simplify our formalization, we derive a (state) predicate for each pair of regular (event) predicates in R. Let V be a countably infinite set of variables disjoint from $\mathsf{R} \cup \mathsf{C}$. For $u, r, r', a, o, o', s \in \mathsf{V}$, we define $U(u) := \neg U_F(u)\ \mathcal{S}\ U_S(u)$, $R(r) := \neg R_F(r)\ \mathcal{S}\ R_S(r)$,

$A(a) := \neg A_F(a)\ \mathcal{S}\ A_S(a)$, $O(o) := \neg O_F(o)\ \mathcal{S}\ O_S(o)$, $S(s) := \neg S_F(s)\ \mathcal{S}\ S_S(s)$, $UA(u,r) := \neg UA_F(u,r)\ \mathcal{S}\ UA_S(u,r)$, $PA(r,a,o) := \neg PA_F(r,a,o)\ \mathcal{S}\ PA_S(r,a,o)$. Moreover, we define $user(s,u) := \neg user_F(s,u)\ \mathcal{S}\ user_S(s,u)$, $roles(s,r) := \neg roles_F(s,r)\ \mathcal{S}\ roles_S(s,r)$, $X(r,r') := \neg X_F(r,r')\ \mathcal{S}\ X_S(r,r')$, and $PD(a) := \neg PD_F(a)\ \mathcal{S}\ PD_S(a)$. Recall that our semantics is point-based and note that all the derived state predicates defined above are TSF domain independent.

8.2.2 Formalization

We now formalize the different variants of separation of duty as introduced above.

General RBAC properties

The following constraints are required to provide a proper RBAC semantics.

Formula (8.1) defines that each running session is associated with exactly one user.

$$\Box\ \forall s : S.\forall u, u' : U.user(s,u) \wedge user(s,u') \rightarrow u \approx u' \tag{8.1}$$

Formula (8.2), expresses the constraint that in a session only those roles may be activated which are presently assigned to the user associated with the session.

$$\Box\ \forall s : S.\forall r : R.roles_S(s,r) \rightarrow \exists u : U.user(s,u) \wedge UA(u,r) \tag{8.2}$$

Formula (8.3) expresses that only permissions of activated roles can be executed.

$$\Box\ \forall s : S.\forall a : A.\forall o : O.exec(s,a,o) \rightarrow \exists r : R.roles(s,r) \wedge PA(r,a,o) \tag{8.3}$$

In the following, we assume the constraints expressed by Formulae 8.1–8.3 are always satisfied.

Static separation of duty

Formula (8.4) expresses that no user must be assigned to a pair of roles that are considered mutually exclusive.

$$\Box\ \forall r, r' : R.X(r,r') \rightarrow \neg\exists u : U.UA(u,r) \wedge UA(u,r') \tag{8.4}$$

Recall that a session is always associated with the same user and that the user remains constant for the lifetime of the session.

Dynamic separation of duty

In the following, we present formalizations of the different types of dynamic separation of duty constraints introduced above.

Simple dynamic separation of duty. Formula (8.5) formalizes that a user may be a member of any two exclusive roles as long as he does not activate them both in the same session.

$$\Box\ \forall r, r' : R.X(r, r') \rightarrow \neg\exists s : S.roles(s, r) \wedge roles(s, r') \tag{8.5}$$

Object-based separation of duty. Formula (8.6) expresses that a user may be a member of any two exclusive roles and may also activate them both at the same time (i.e., in the same session), but he must not act upon the same object through both.

$$\begin{aligned}\Box\ &\forall s : S.\forall a, a' : A.\forall o : O.\forall r, r' : R.\\ &exec(s, a, o) \wedge roles(s, r) \wedge X(r, r') \wedge PA(r, a, o) \wedge PA(r', a', o)\\ &\quad \rightarrow \big(\neg exec(s, a', o)\ \mathcal{S}\ roles_S(s, r') \wedge PA(r', a', o)\big)\end{aligned} \tag{8.6}$$

In other words, Formula (8.6) prevents the execution of an action on an object whenever the same user has executed another action on the same object associated with a conflicting role in a single session.

Operational separation of duty. Formula (8.7) expresses that a user may be a member of some exclusive roles as long as the set of actions acquired by activating these roles does not permit him to execute every step of a workflow.

$$\begin{aligned}\Box\ &\big(\exists a', a'' : A.PD(a') \wedge PD(a'') \wedge \neg(a' \approx a'')\big)\\ &\quad \rightarrow \forall u : U.\exists a : A.PD(a)\\ &\qquad \wedge \neg\big(\blacklozenge\, \exists s : S.\exists r : R.\exists o : O.user(s, u) \wedge roles(s, r) \wedge PA(r, a, o)\big)\end{aligned} \tag{8.7}$$

In other words, Formula (8.7) states that whenever a critical workflow consists of at least two workflow steps, then for all users there is always at least one workflow action that has never been activated in any session up to the current time point. We point out that we might as well remove the antecedent from the above implication and turn it into a separate constraint.

History-based separation of duty. Formula (8.8) formalizes the HbSoD constraint that a user may be a member of some mutually exclusive roles and the complete set of authorizations acquired over these roles may cover an entire workflow, but a user must not perform all the workflow steps involving the same object.

$$\begin{aligned}\Box\ &\forall u : U.\forall a : A.\forall o : O.PD(a)\\ &\wedge \big(\forall a' : A.PD(a') \wedge \neg(a \approx a') \rightarrow \big(\blacklozenge\, \exists s : S.exec(s, a', o) \wedge user(s, u)\big)\big)\\ &\quad \rightarrow \neg\big(\exists s' : S.exec(s', a, o) \wedge user(s', u)\big)\end{aligned} \tag{8.8}$$

More specifically, Formula (8.8) prevents any user from activating any role associated with an action and an object if the same user previously executed all other actions (except that one) required to complete the workflow on the same object.

8.2.3 Discussion

Formulae (8.4)–(8.8) demonstrate that different variants of SoD can be formally expressed by means of past-only and TSF domain independent MFOTL formulae. Consequently, we can use our runtime monitoring approach for finite relations to detect violations of the respective SoD constraints. Whereas the presented formulae often look like first-order formulae, this is due to the use of syntactically derived state predicates, which hide the involved temporal subformulae. For example, Formula (8.2) involves no less than 5 temporal subformulae.

We point out that our formalization of the above SoD constraints is similar to the one of Mossakowski et al. [MDS03]. Specifically, our formalizations of SSoD, SDSoD, OpSoD, and HbSoD are essentially equivalent to the state-based FOTL formulae given in [MDS03]. Our formalization of dynamic ObSoD, however, differs from the one of Mossakowski et al. Specifically, Formula (8.6) prevents the execution of any action on an object whenever the same user has executed another action on the same object in a single session if both actions are associated with a pair of conflicting roles. In contrast, the ObSoD constraint proposed by Mossakowski et al. prevents a user from activating a role associated with an object in any session whenever the same user has executed an action associated with a conflicting role on the same object in the same or a different session. Moreover, while the approach of [MDS03] exclusively focusses on the formalization of SoD, we also provide a runtime monitoring approach for the formalized SoD constraints.

8.3 Monitoring Separation of Duty Constraints

We investigate the feasibility of runtime monitoring the SoD constraints presented in the previous section. We specifically focus on monitoring the SDSoD and ObSoD constraints expressed by Formulae (8.5) and (8.6), respectively.

8.3.1 Observations

Before we present our experiments, let us briefly discuss the feasibility of monitoring the OpSoD and HbSoD constraints expressed by Formulae (8.7) and (8.8). Because both formulae are unbounded in the past, they can be feasibly monitored only for finite temporal structures of a certain length. Here, the results reported in Chapter 7 on the space and time requirements for monitoring Formulae (7.1) and (7.5) still provide an indication of the extent to which the monitoring with unbounded past operators is feasible. Because the Formulae (8.7) and (8.8) are considerably more complex than Formulae (7.1) and (7.5) and make use of additional temporal subformulae, however, the respective monitors for Formulae (8.7) and (8.8) perform less well in practice than the respective monitors for offline monitoring Formulae (7.1) and (7.5).

8.3.2 Experimental Analysis

To assess to what extent Formulae (8.5) and (8.6) can be monitored, we performed experiments similar to those described in Section 7.3.2. In the following, we often write ϕ_{SDSoD} for Formula (8.5) and ϕ_{ObSoD} for Formula (8.6).

Setup of the Experiments

We first summarize the setup of our experiments.

Generation of input data. We generated (finite) temporal structures over S_{SoD} using a procedure similar the one described in Section 6.2.1. To generate individual S_{SoD}-structures, we used the sample domain $\Omega_{200\times50\times10\times1000\times1000}$, consisting of 200 users, 50 roles, 10 actions, 1,000 objects, and 1,000 sessions. All generated temporal structures over S_{SoD} consisted of 5,000 S_{SoD}-structures.

In contrast to the input generation procedure described in Section 6.2.1, we assumed that some events occur with a higher probability than others. In particular, after an initial start-up phase used to initialize the RBAC configuration, we assumed that only 5% of the events were RBAC administration events (i.e., modifications of the sets U, R, A, O, and the RBAC configuration $(\mathsf{UA}, \mathsf{PA})$) or modifications of the set of mutually exclusive roles. We further assumed that 25% of the observed events were session creation/termination or role activation/deactivation events. The remaining 70% events represented executions of activated permissions.

Measurements. We measured the space consumption of the monitors $\mathcal{M}(\phi_{\text{SDSoD}})$ and $\mathcal{M}(\phi_{\text{ObSoD}})$ by the cardinality of the relevant auxiliary relations as explained in Section 6.2.1. As a reference point, we also calculated the active domain (ADOM) from all relations occurring in the processed prefix. Moreover, we measured the time required to process a temporal structure up to selected time points, i.e., after 1,000, 2,000, 3,000, 4,000, and 5,000 processed S_{SoD}-structures.

Analysis and Results

We used the runtime monitoring framework described in Chapter 6 to generate the monitors $\mathcal{M}(\phi_{\text{SDSoD}})$ and $\mathcal{M}(\phi_{\text{ObSoD}})$. Each of the monitors then processed 10 independently generated temporal S_{SoD}-structures as explained above.

Table 8.1 summarizes the average space consumption of the monitors $\mathcal{M}(\phi_{\text{SDSoD}})$ and $\mathcal{M}(\phi_{\text{ObSoD}})$ at specific time points, as obtained from running the conducted experiments. As reference points, we also determined the average space consumption of a (hypothetical) monitor $\mathcal{M}(\phi_{\text{ObSoD-RBAC}})$ and the active domain. The average space consumption of $\mathcal{M}(\phi_{\text{ObSoD-RBAC}})$ was derived from the one of $\mathcal{M}(\phi_{\text{ObSoD}})$ by counting only those auxiliary relations that keep track of past executions in all active sessions. All

	Number of processed events				
	1,000	2,000	3,000	4,000	5,000
$\mathcal{M}(\phi_{\text{SDSoD}})$	464	743	769	862	912
$\mathcal{M}(\phi_{\text{ObSoD}})$	832	1,681	1,943	1,928	2,188
$\mathcal{M}(\phi_{\text{ObSoD-RBAC}})$	319	724	867	843	921
ADOM	954	1,878	2,013	2,176	2,229

Table 8.1: Average space consumption of $\mathcal{M}(\phi_{\text{ObSoD}})$, $\mathcal{M}(\phi_{\text{SDSoD}})$, and $\mathcal{M}(\phi_{\text{ObSoD-RBAC}})$ for monitoring a finite temporal S_{SoD}-structure as computed from 10 independent replications of the experiment.

	Number of processed events				
	1,000	2,000	3,000	4,000	5,000
$\mathcal{M}(\phi_{\text{SDSoD}})$	32	179	462	731	997
$\mathcal{M}(\phi_{\text{ObSoD}})$	84	447	1,319	2,345	3,433

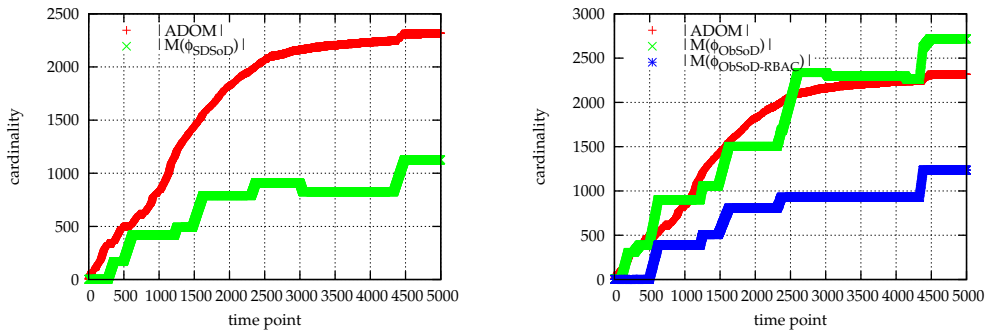
Table 8.2: Average time (in seconds) required by $\mathcal{M}(\phi_{\text{ObSoD}})$ for monitoring a finite temporal S_{SoD}-structure as computed from 10 independent replications of the experiment.

the RBAC-specific auxiliary relations were ignored. This reflects a setting wherein the current RBAC configuration and all users' activated roles can be directly obtained by means of querying the RBAC system.

Table 8.2 summarizes the average times required by the monitors $\mathcal{M}(\phi_{\text{SDSoD}})$ and $\mathcal{M}(\phi_{\text{ObSoD}})$ to process a prefix of the generated temporal S_{SoD}-structures of length up to at most 5,000 time points.

For the investigated input data, the average space consumed by $\mathcal{M}(\phi_{\text{SDSoD}})$ fluctuated around a constant level, depending on the number of currently running sessions and the number of activated roles. Furthermore, the average time for processing another 1,000 S_{SoD}-structures stayed relatively constant after the initial start-up phase. After the start-up phase, the processing of a single S_{SoD}-structure required about 0.2 seconds. A representative example of the evolution of the space consumption of $\mathcal{M}(\phi_{\text{SDSoD}})$ is shown in Figure 8.18.1(a).

Similarly, the average space consumption of the monitor $\mathcal{M}(\phi_{\text{ObSoD}})$ increases with the number of elements in the current RBAC sets as well as with the number of executed permissions in sessions. The average space consumed by $\mathcal{M}(\phi_{\text{ObSoD}})$ increased at a similar speed like the cardinality of the active domain. As indicated by the average space consumption of $\mathcal{M}(\phi_{\text{ObSoD-RBAC}})$, the tracking of previously started executions in the same session accounted for approximately half of the space consumption of $\mathcal{M}(\phi_{\text{ObSoD}})$. The other half of the space consumed by $\mathcal{M}(\phi_{\text{ObSoD}})$ is due to the maintenance of the RBAC-specific information such as the current RBAC configuration and currently active roles and sessions. A representative example of the evolution of the space consumption of $\mathcal{M}(\phi_{\text{ObSoD}})$ is depicted in Figure 8.18.1(b).



(a) Space consumption of $\mathcal{M}(\phi_{\text{SDSoD}})$. (b) Space consumption of $\mathcal{M}(\phi_{\text{ObSoD}})$.

Figure 8.1: Example evolution of the space consumption of the monitors $\mathcal{M}(\phi_{\text{SDSoD}})$, $\mathcal{M}(\phi_{\text{ObSoD}})$, and $\mathcal{M}(\phi_{\text{ObSoD-RBAC}})$ as they process a temporal structure consisting of 5,000 first-order S_{SoD}-structures.

8.3.3 Discussion

The results obtained from our initial experiments indicate that runtime monitoring dynamic separation of duty constraints such as SDSoD or ObSoD may be feasible in many practical settings. To further substantiate our findings, however, additional experiments, ideally based on real RBAC events, and a more comprehensive analysis would be required.

Monitoring the presented constraints can be more generally feasible if the average processing performance of our prototypical monitors is further improved. For example, it will be interesting to see to what extent the processing performance of our monitors can be improved by the implementation of more sophisticated and incremental indexing techniques for the maintained auxiliary relations.

Moreover, as indicated by the average space consumption of $\mathcal{M}(\phi_{\text{ObSoD-RBAC}})$, in many practical application scenarios it may be possible to substitute some of the derived state predicates by direct queries. For example, in an existing RBAC system, the relations UA and PA constituting the current RBAC configurations as well as the sets U, R, A, and O are often maintained in a database. Instead of incrementally constructing these sets from the observed events, a dedicated SoD monitor could directly query the respective sets in order to obtain interpretations of the respective predicates for the current time point.

In certain situations, it might be useful to extend the presented SDSoD and ObSoD constraints with metric constraints. For example, this would enable the online monitoring of an alternative version of the ObSoD constraint presented in Formula (8.6), where consecutive actions that are associated with mutually exclusive roles must not be executed on the same object in any session that is associated with the same user. We point out that thus the online monitoring of many variants of history-based separation and binding of duty constraints presented by Sandhu becomes directly feasible subject to the results presented in Section 6.2. Similarly, metric versions of Formulae (8.7)

and (8.8) would also allow for online monitoring OpSoD and HbSoD, respectively.

8.4 Summary and Outlook

In this chapter, we have demonstrated how various SoD constraints can be expressed by means of past-only and TSF domain independent MFOTL formulae of the form $\Box\,\phi$, where ϕ is a past-only formula. We then investigated to what extent the presented formulae expressing simple dynamic separation of duty or object-based dynamic separation of duty can be monitored with our prototypical runtime monitoring framework for finite relations. While both constraints could be monitored without problems, the measured processing times indicated that our prototype should be improved. This is particularly true for the monitor $\mathcal{M}(\phi_{\text{ObSoD}})$, which, on average, required about one second to compute an incremental updating step.

"Those who cannot remember the past are condemned to repeat it."

George Santayana

Chapter 9

Related Work

Whereas Section 2.4 reviewed alternative means of property specification and Section 3.2 classified existing work on runtime monitoring with respect to a simple typology, in this chapter we review closely related work in more detail and explicitly distinguish our own from other monitoring approaches.

The chapter is organized as follows. Section 9.1 starts with an overview and some general comments. Related work from the area of dynamic integrity checking and temporal databases is summarized in Section 9.3. We consider relevant work from the field of software program monitoring in Section 9.2. Finally, we also discuss some related work in the context of data stream management and complex event processing.

9.1 Overview

The majority of work related to our monitoring method appeared in one of three application domains: software program monitoring, dynamic integrity checking for databases, or data stream management. While most work in the area of software program monitoring focussed on properties defined in propositional specification languages [Run08], work on dynamic integrity checking or data stream management naturally concentrated on more expressive specification languages such as FOTL or 2-FOL.

Despite the large amount of work on (specification-based) runtime monitoring, competing monitoring methods are restricted in the properties they can handle. They either support only properties expressed in propositional temporal logics and thus cannot cope with variables ranging over infinite domains [KPA03, HR04a, TR05, BLS06, MNP06], do not provide both universal and existential quantification [RGL01, HJL03, BGHS04b, DSS+05, MNP06, NM07] or merely in restricted ways [Tom03, BGHS04b, SSLK06, Sto07, HV08], do not allow arbitrary quantifier alternation [LS87, BGHS04b], cannot handle unrestricted negation [Cho95, LS87, SW95, Tom03, BGHS04b, SSLK06, Sto07], do not provide quantitative temporal operators [LS87, RGL01, Sto07, HV08], or cannot simultaneously handle both past and future temporal operators [LS87, Cho95,

SW95, RH05, MNP06, Sto07, NM07, HV08].

In our presentation of related work, we focus on runtime monitoring methods for execution models and properties defined in specification languages with an expressiveness similar to MFOTL. That being said, we do not discuss in detail the large amount of work on runtime monitoring with propositional specification languages. Because of the large amount of work on the topic of monitoring, a comprehensive survey of each and every approach is out of scope of this dissertation. Instead, we mention some domain-specific entry points into the subject. For an early and comprehensive overview on runtime monitoring concepts in the context of program execution monitoring and debugging, see [PN81]. For a taxonomy and discussion of software fault monitoring, we refer to [DGa04]. For an overview of dynamic integrity checking methods for databases, see [Pac97] or the related work sections of [Cho95] and [SW95]. For an overview of issues and related work in the context of data stream management, see [BBD+02] or [ABW03].

9.2 Software Program Monitoring

A large proportion of recent work on runtime monitoring was done in the context of software program monitoring [Run08]. The goal of software program monitoring is to complement verification techniques such as model checking or testing with techniques to monitor or analyze the execution of software programs. While the majority of the proposed approaches rely on a propositional state/event model (e.g., [KPA03, HR04a, TR05, BLS06, MNP06]), several recently proposed approaches extended propositional runtime monitoring methods with support for variables or possibly some form of quantification [BGHS04b, SSLK06, Sto07, BRH07, HV08].

9.2.1 Propositional Runtime Monitoring

The majority of runtime monitoring approaches uses (future-only or past-only) LTL as their specification language [LKK+99, Dru00, KKL+01, Gei01, GH01, HR01a, HR01b, HR02, Dru03, FS04, KVK+04, HR04a, SB06, BLS06]. Other runtime monitoring approaches have been proposed for properties defined using quantitative propositional temporal logics such as MTL, TPTL, or variants thereof [ML97, GD00, Dru00, Dru03, Gei03, KPA03, HJL03, TR05, Dru06, BLS06, MNP07, BLS07b, AK07]. Some monitoring approaches are based on properties specified using regular expressions [SR03, vL06] or directly as automata [Sch00, dR05, LBW05].

Most propositional runtime monitoring methods are based on the construction of a finite state automaton that recognizes the language induced by a given property specification. In particular, for properties defined using LTL or regular expressions, the generation of a respective runtime monitor may be based on modified Büchi au-

tomata [GH01, Gei01, SR03, ABLS05, vL06, BLS06, BLS07b], alternating automata [FS04, Dru06, SB06], or dependency graphs [LKK+99, KKL+01, KVK+04]. Analogously, for properties specified using MTL, TPTL, or other quantitative propositional temporal logics, runtime monitoring techniques may leverage timed automata [GD00, Gei03, MNP07], event clock automata [ABLS05, BLS06, BLS07b], or (constraint or dataflow) graphs [ML97, Dru00, Dru03]. In other cases, rewriting-based monitoring techniques were used [HR01a, HR01b, HJL03, KPA03, HR04a, TR05, RH05]. A trace-storing approach for past-only LTL based on dynamic programming was described in [HR02, HR04a]. Moreover, a transformation-based technique based on difference decision diagrams was recently described in [AK07].

An approach for the monitoring of propositional safety properties in distributed systems is investigated in [SVAR04]. Global properties of the distributed system are specified using PD-LTL, a distributed variant of past-only LTL. The proposed algorithm monitors the global property by monitoring a local property de-centrally, that is, at each node in the distributed system individually.

Notable examples of propositional runtime verification tools are JPaX [HR01a] and Java-MaC [LKK+99, KKL+01, KVK+04], two frameworks for instrumenting and monitoring Java programs at execution time with respect to property specifications expressed using (variants of) LTL. Similarly, the commercial tools Temporal Rover and DBRover [Dru00, Dru03] generate executable code from property specifications written in LTL or MTL, which is directly embedded at specific positions in the program code.

9.2.2 Runtime Monitoring with Data

Several authors recently started to investigate runtime monitoring techniques that support variables and possibly some form of quantification [BGHS04b, SSLK06, Sto07, BRH07, HV08].

The runtime monitoring approach of [BGHS04b] is based on EAGLE, a simple yet expressive specification language. EAGLE is based on recursive rule definitions over three basic temporal connectives (next-time, previous-time, and concatenation), supports variables and a restricted form of quantification, and allows for embedding logics such as LTL, a variant of MTL, and extended regular expressions. Being implemented as a Java library, EAGLE's semantics is defined with respect to sequences of (Java) program states. In contrast to our own approach, EAGLE provides only restricted forms of existential quantification and negation. The non-trace-storing monitoring algorithm is based on formula rewriting. For a given EAGLE formula ϕ, at each state s the formula ϕ is evaluated and rewritten into a new formula ϕ' such that ϕ holds in s if and only if ϕ' also holds in the next state s'. Because of EAGLE's expressiveness, checking satisfiability is undecidable in general. For the EAGLE fragment capable of expressing

future-only LTL, the space (and time) required to evaluate a future-only LTL formula is exponential in the length of the formula and independent of the length of the processed trace [BGHS04a]. More recently, Barringer et al. also described RULER [BRH07], a rule-based property specification language similar to EAGLE. Compared to EAGLE, the authors expect the specification of properties to be simpler with RULER. Several temporal logics can be embedded in RULER, too. Moreover, RULER specifications are expected to lend themselves to the generation of more efficient runtime monitors.

An extension of the Java-MaC toolset [LKK+99, KKL+01, KVK+04] for the specification of properties using variables and a restricted form of quantification was described in [SSLK06]. Properties are specified using the specification language LC_v, a variant of first-order temporal logic with restricted negation and quantification. Similar to the freeze quantifier of [AH94], LC_v provides an additional attribute quantifier, which binds a variable in a formula to an attribute value of the most recent occurrence of a specified event. Essentially, an event is a time-stamped singleton relation. The semantics of the logic is defined with respect to timed temporal databases with singleton relations. The monitoring technique is non-trace-storing and based on the construction of an acyclic graph. For a given formula ϕ, the monitor maintains an acyclic graph that represents the dependencies between all subformulae of ϕ. Each node corresponds to a subformula and is associated with variables that keep track of relevant information as new states are processed in a bottom-up and incremental fashion. The space required for monitoring a given formula depends on the data in the observed prefix of the trace and is exponential in the quantifier nesting depth.

A further runtime monitoring approach for property specifications with restricted support of variables, negation, and quantification was described in [Sto07]. As property specification language, the authors use next-free, parameterized LTL (pLTL), a syntactically restricted variant of future-only FOTL, which provides a special implication connective. The implication connective ensures that each variable occurring in a parameterized formula is bound by a value occurring in the current state. For the construction of a runtime monitor from a given pLTL formula, an existing construction of an alternating finite automaton from a given LTL formula [Var96] is extended to yield a parameterized alternating automaton, which is augmented with a map to maintain the current bindings for each subformula. For online monitoring the automaton is traversed in a breadth-first fashion. This requires that each state is only processed once and guarantees that violations are detected as soon as they occur.

A runtime monitoring approach for properties specified in LTL-FO$^+$, a variant of future-only FOTL with unary predicates only, was proposed in the context of business process monitoring [HV08]. Quantifiers are evaluated using a restricted form of active domain semantics [Ull88]. System executions are represented as sequences of flat XML messages, that is, a special case of temporal databases. A flat XML message is essentially a database over a signature with unary relations only. The XML tags occurring in

a given message correspond to predicate symbols in the signature and the set of values contained in each tag correspond to the relation that is associated with the respective predicate. The ordering between tags can be preserved by using binary instead of unary predicates and associating each value together with its position in the message. The non-trace-storing monitoring algorithm, inspired by [GPVW96], is based on an on-the-fly construction of a finite state automaton that incrementally processes new messages as they arrive. More specifically, upon reception of a new message the runtime monitor is updated by calling its update function, which takes it into the next state. Intuitively, each state of the runtime monitor represents one possible continuation of the observed trace to satisfy the given property. The update function decomposes and evaluates all formulae that must be satisfied in the current state and produces all the properties that must hold in the subsequent state. At each time point during monitoring, the runtime monitor can decide if the monitored property is violated, satisfied, or whether nothing can be said yet.

In the approach of [KMSF01], events gathered during program execution are stored in a database. After the monitored program terminates, the program execution is analyzed by means of SQL queries. A similar approach is described in [GOA05], where properties are expressed using a language called Program Trace Query Language (PTQL). PTQL is based on relational queries that are interpreted over a timed database history that stores the entire execution trace. A compiler instruments a given Java program and executes a set of given PTQL queries in an online fashion, i.e., whenever a relevant event is added to the database. No history encoding is provided and, as a result, the size of the database history for storing the events grows linearly over time. In this sense, the approach is strongly reminiscent of any 'naive' monitoring approach based on databases. The approach might benefit from Toman's data expiration strategy [Tom03]. However, the size of the history would still be non-elementary in the cardinality of the set of data constants occurring in the processed prefix.

Summing up, none of the presented monitoring approaches provides an expressiveness that matches the one of our online runtime monitoring approach for MFOTL. In particular, none of them provides support for unrestricted negation and quantification. Moreover, also for the setting with finite relations, the ability to arbitrarily nest both metric past and bounded future operators is unmatched by previous approaches.

9.3 Dynamic Integrity Checking

Several authors investigated the use of closed FOTL formulae for the specification of dynamic integrity constraints or temporal triggers in databases [Kun84, Kun85, LS87, LF89, HS91, Cho95, GL96].

In Chomicki's dynamic integrity checking method [Cho95], integrity constraints are expressed as closed past-only (M)FOTL formulae, which are interpreted over (timed)

database histories. In a nutshell, the monitoring method works as follows: For each temporal subformula of a given integrity constraint ϕ, the current database state is augmented with one or several auxiliary relations to form an extended database state. A set of first-order formulae defines how each auxiliary relation is incrementally updated from the previous extended database state as a new database state becomes available. Instead of evaluating ϕ over the entire database history, at each time point, a transformed first-order formula $\hat{\phi}$ is evaluated over the current extended database state. To enable the incremental updating of the auxiliary relations in a bottom-up fashion, however, the first-order formulae induced by each temporal subformula must be domain independent [Fag82]. To analyze the space consumption of his integrity checking method, Chomicki introduced the notion of a history encoding and showed that the history encoding provided by his method is polynomially bounded. An implementation based on an active database management system for integrity constraints expressed as past-only FOTL formulae was described in [CT95].

Our own approach extends the work of Chomicki along several lines. First, we generalized Chomicki's approach to handle timed temporal automatic structures. The use of automatic structures allows for the unrestricted use of negation and quantification in property specifications. Second, we extended the approach to properties specified by bounded MFOTL formulae. This enables the arbitrary use of both past and bounded future operators to specify properties. In spite of this richer syntactic fragment, our method provides a polynomially bounded history encoding for the setting with finite relations. Third, we also extended the rewrite procedure of [CTB01] to support metric temporal operators and introduced dual operators to handle a larger set of formulae. Finally, we generalized Chomicki's notion of history encoding into a general method for the analysis of the space consumption of online runtime monitors.

Lipeck et al. proposed several similar methods for checking integrity constraints defined within a restricted fragment of future-only FOTL, called the biquantified fragment [LS87, LF89, HS91, GL96]. Outside temporal operators, biquantified formulae only allow (implicit) universal quantification and thus do not support quantifier alternation [LS87, CN95]. Under this restriction, the standard construction of a finite automaton, referred to as transition graph in [LS87], which recognizes exactly the language induced by a given property specification can be leveraged as the monitoring method [Wol83]. The resulting monitoring methods are non-trace-storing and provide a bounded history encoding. In the case of infinite domains, however, the approach of [LS87] may require that an infinite number of substitutions be stored. A possible solution, proposed in [HS91], is to use so-called substitution descriptions, which finitely describe the possibly infinite substitution sets. This is reminiscent of our use of automatic presentations for representing structures and, in particular, infinite relations.

A further related approach, focussed on the evaluation of temporal triggers in databases, was proposed in [SW95]. In contrast to our own approach, the temporal

triggers are specified using either PTL or FTL, that is, past-only or future-only variants of temporal logic with predicates. Instead of standard first-order quantifiers, both logics provide a freeze quantifier to bind variables to values or relations to temporal contexts (similar to TPTL [AH94]). The monitoring method for FTL is based on the maintenance of a directed, rooted, and acyclic and-or graph (called a requirements graph), which captures all the information in the processed prefix that is required to correctly evaluate a given temporal trigger condition. The monitoring method for PTL is similar to the one of Chomicki. In particular, it is based on the introduction and incremental updating of auxiliary relations for each temporal subformula occurring in a given temporal trigger. Both methods are non-trace-storing and incremental but the described implementation does not provide a bounded history encoding [Cho95].

The following work from the context of databases and integrity checking is less closely related but also worthwhile mentioning.

The integrity checking approach proposed by Kung [Kun84, Kun85] also uses a variant of FOTL for the specification of static and dynamic database constraints. In contrast to our own approach and other work on dynamic integrity checking, the proposed monitoring method requires predefined operation specifications with well-defined semantics and does not allow for monitoring arbitrary sequences of database states.

The idea of using relations for monitoring complex systems was also expressed in [Sno88]. The described monitoring framework captures relevant state or event information about the monitored system in a database history, where are relations are singletons. Properties are specified as TQuel [Sno87] queries, which are transformed into equivalent relational algebra expressions that are incrementally evaluated over the history. The temporal operators of TQuel are all expressible in past-only FOTL [Cho95]. The approach shares many similarities with and can be seen as a precursor of recent work on data stream management, in particular [CcC+02].

Also related is [Mor92], where TRIO (a metric first-order temporal logic very similar to MFOTL) [GMM90] specifications are evaluated over timed database histories and timed temporal databases over *finite* domains using a history checking algorithm. Because of the restriction to finite domains, however, the monitoring method is of a propositional nature. In particular, the first-order formalism is merely syntactic sugar that makes the specification of propositional properties more convenient.

Finally, we remark that our work is also related to temporal databases [ÖS95, CT05]. Specification languages such as FOTL, MFOTL, and 2-FOL have been proposed and investigated as query languages for temporal databases [Cho94, CT98]. While temporal databases allow for the evaluation of queries about the data stored in the temporal database in an ad-hoc fashion, in runtime monitoring, integrity checking, or temporal trigger evaluation, the query or constraint is fixed a priori and only the data that are necessary to evaluate the fixed query are preserved. The problem of determining which data can be removed from a temporal database is known as logical data expira-

tion [Tom03]. For runtime monitoring, query-based expiration techniques are particularly important. Indeed, runtime monitoring methods providing a bounded history encoding are instances of query-based logical data expiration operators.

9.4 Data Stream Management

Data stream management is concerned with the processing of continuous queries over data streams. Recall that a data stream is a sequence of time-stamped first-order structures, where each structure consists of a singleton relation only. Queries are evaluated continually, that is, the same query is re-evaluated upon arrival of each new structure and each structure is only processed once [BW01, BBD+02, GLdB07]. Observe that open (M)FOTL formulae of the form $\Box\,\phi(\bar{x})$ correspond to (continuous) queries. Data stream management thus shares many similarities with our runtime monitoring approach.

Query languages proposed in the area of data stream management range include less expressive event-condition-action languages as commonly used in the context of complex event processing and active databases. For example, Gehani et al. described an approach for monitoring trigger conditions in an object-oriented context [GJS92]. Trigger conditions for each object are specified using event expressions, which are based on regular expressions. Event expressions may contain variables but quantification over these variables is not supported. From a given event expression, the monitoring method automatically constructs a finite state automaton that recognizes the same language. This provides a bounded history encoding [Cho95]. Even for the case without variables, however, the size of the constructed automaton is super-exponential in the length of the event expression [SM73, SW95]. Other work on the specification and monitoring of event patterns includes [BG98, SSS+03]. For example, in [BG98] event patterns are specified using a specification language based on Timed Calculus of Communicating Systems (TCCS) [Yi90]. A rather expressive algebra for monitoring event streams with support for parameterized composite events and support for aggregate queries was described in [ADW05]. Moreover, an approach for efficient pattern matching over event streams was recently presented in [ADGI08]. A further example is LOLA [DSS+05], a formalism for runtime monitoring based on a functional language over finite streams. The algorithm for online monitoring of queries in this language follows a partial evaluation strategy: it incrementally constructs output streams from input streams, while maintaining a store of partially evaluated expressions for forward references. For a restricted class of specifications, characterized syntactically, the algorithms memory requirement is independent of the length of the input streams.

More expressive query languages are used in data stream management systems like Aurora/Borealis [CcC+02, ABc+05], STREAM [ABW06], and System S [AAB+06]. Each system uses a different approach to formulating continuous queries. In the case of

STREAM, continuous queries are formulated in the continuous query language (CQL) [ABW03, ABW06], a declarative query language similar to SQL. In contrast, both System S and the Aurora/Borealis systems use a semi-imperative querying approach, which works by applying different types of stream operators to a set of input streams to produce an according set of output streams [CcC+02, ABc+05, GAW+08]. As opposed to the traditional database field, where SQL has become the query language of choice, no standard query language for data streams has been adopted yet [ZJM+08]. Moreover, both the semantics and the expressiveness of continuous query languages are often ad-hoc and not well understood. A further problem is the computation of bounded synopses [BBD+02], that is, data summaries that guarantee a bounded history encoding. Also here, query-based data expiration techniques [Tom03] provide an important starting point. A characterization of a subclass of queries that can be computed with bounded memory was presented in [ABB+04]. As recently demonstrated, however, for many continuous query languages based on a SQL-style duplicate semantics such as CQL, the computation of bounded synopses is impossible [Tom07].

Our runtime monitoring approach with finite relations for properties specified as bounded and TSF domain independent MFOTL formulae can thus be used also as a data stream management approach. Specifically, the polynomially bounded space consumption of our approach makes possible the computation of bounded synopses.

"Philosophy begins in wonder.
And, at the end, when philosophic
thought has done its best, the wonder
remains."

Alfred North Whitehead

Chapter 10

Conclusion

We conclude this dissertation with a summary of our main results and a discussion of open problems for future work.

10.1 Summary

In this dissertation, we first motivated the use of timed temporal first-order structures as a general means for representing executions of a monitored system. We then introduced metric first-order temporal logic as an expressive property specification language and briefly reviewed alternative means for formally defining temporal system properties. We subsequently gave a brief introduction to the field of runtime monitoring and provided a classification of applications of this fundamental technology.

After this general introduction, we presented an automata-based runtime monitoring approach for an expressive fragment of a metric first-order temporal logic. The supported fragment allows the arbitrary nesting of temporal past and bounded future operators and supports unrestricted use of negation and quantification. The use of automata thus substantially generalizes both the kinds of structures and the class of formulae that can be monitored. Moreover, it eliminates the limitations that arise in databases, where relations must be finite. The resulting expressiveness is unparalleled in previous work on runtime monitoring.

Having presented our approach in full generality, we separately considered the special case of our approach of the setting where all relations are finite. Under this restriction, relations can be represented directly as tables, which enables the implementation of our monitoring approach by leveraging standard database technology and allows for several types of space optimizations. Moreover, we presented an extension of an existing rewrite procedure to ensure that only such formulae are admitted for runtime monitoring that can also be monitored in this setting. As our main result, we then proved that the space required by our monitoring approach is polynomially bounded by the cardinality of the data appearing in the processed prefix of the moni-

tored timed temporal structure. Our result contrasts with the space consumption of a logical data expiration technique for 2-FOL formulae, which is non-elementary in the data occurring in the processed prefix.

The remaining chapters focussed on the validation and potential applications of our monitoring approach. We first described a prototypical implementation of our approach for the setting with finite relations. We then presented the results of an experimental steady-state analysis of the space consumption of our runtime monitors for several interesting formula classes and compared our results with a simple benchmark. For the investigated setting, our general results showed that runtime monitoring of many interesting metric first-order temporal logical properties is indeed feasible. We complemented the results of this general analysis with two case studies, which focussed on the formalization and monitoring of compliance requirements and separation of duty constraints. The investigated case studies corroborate the initial hypothesis that metric first-order temporal logic is expressive enough to naturally specify and effectively monitor an interesting variety of real-life requirements.

10.2 Future Work

While our work provided solutions to previous problems, it naturally also resulted in new challenges and uncovered open problems for future work. In the following, we list open problems that deserve further study.

Theoretical Problems

On the theoretical side, we identified the following areas for future work.

Execution model. Regarding the execution model and thus the semantical foundation of our property specification language, areas of future work include the investigation of alternative time semantics and the theoretical analysis of the structures used to represent states or events.

While our choice of temporal structures with a point-based time semantics is appropriate for runtime monitoring in the context of event-based systems or data streams, an interval-based time semantics might be more appropriate for monitoring the evolution of system states. Because system states last for a non-atomic amount of time, it would be natural to assign a time interval rather than a single time point to each structure representing observable state information. Moreover, in other settings a hybrid model, combining both point-based and interval-based aspects, might be most appropriate. While state predicates can be synthesized from event predicates (see Chapter 8), it would be interesting to also investigate to what extent state predicates can be dealt with directly and how this impacts our monitor constructions.

The general version of our monitoring approach is based on automatic structures and the representation of these structures by means of finite state automata. While this generality enables unrestricted negation and quantification, it may come at the price of a significant space complexity. Indeed, the determinization of a non-deterministic finite state automation resulting from an existential quantification may yield an exponential blow-up in the number of states of the resulting automaton.

It would thus be interesting to investigate to what extent Theorem 5.5.3 can be carried over to temporal structures with possibly infinite relations and automatic representations. A first challenge here is to define an appropriate function g that measures the size of the automatic representations. In particular, g should be independent of the length of the prefix. The second challenge then would be to establish tight upper bounds on the size of the resulting automata for automatic structures.

Similar to the special case with finite relations, there might be alternatives to automatic structures, which are closed under all operations required to effectively decide first-order logic. As a possible alternative to automatic representations, it would be interesting to investigate constraint databases for the representation of structures [KKR95, KPL00].

Finally, it will be interesting to see to what extent automatic (or other finitely presentable) structures can be used in other monitoring approaches, thereby solving the problems they have with infinite relations.

Specification language. For applications such as the monitoring of stock prices, the ability to dynamically evaluate aggregate functions (e.g., the computation of the average price of a traded stock over the past two days) is crucial. While the computation of aggregates is beyond first-order logic, aggregate functions such as `sum`, `average`, `min`, `max`, or `count` are standard in commercial database systems (see [Klu82, OOM87, HLNW01] for such extensions to relational algebra and relational calculus). To accommodate our runtime monitoring approach for applications such as stock price monitoring, it would be interesting to extend MFOTL with aggregate operators and to study the resulting complexity implications.

Another important open problem is the question whether the bounded MFOTL fragment is more expressive than the past-only MFOTL fragment. In case the two fragments were equally expressive, it would be interesting to investigate whether one can find an equivalence-preserving translation between the two syntactical fragments and to what extent the bounded MFOTL fragment could result in more succinct property specifications than the past-only MFOTL fragment.

Finally, for the setting with finite relations, we could investigate alternatives to our rewrite procedure for the identification of monitorable formulae. One approach might be to extend the syntactical procedure used in [CT95], which builds on class of evaluable formulae [GT91, Dem92]. The class of evaluable formulae represents a larger sub-

set of the class of domain independent formulae than the class of safe-range formulae.

Monitoring Method. Our experiments revealed that the space consumption of our monitors may be significantly larger for formulae involving future operators than for formulae consisting of only past operators. As a result, it will be interesting to investigate to what extent our constructions for temporal future operators can be improved. In doing so, we will evaluate alternative data structures and algorithms for efficiently and incrementally updating relations, which is at the heart of our monitoring algorithm.

Practical Aspects

With respect to practical applications of our runtime monitoring approach, we consider the following areas interesting for future work.

Prototype. Regarding our prototypical runtime monitoring framework, there are numerous areas of possible improvement. In particular, we would like to incorporate additional optimizations (e.g., algebraic transformations and context-based optimizations) in order to further minimize the size of maintained auxiliary relations. Moreover, we want to extend our framework with more efficient data structures and indexing strategies for the representation and modification of tables. Specifically, we would like to investigate to what extent the stored relations could more efficiently organized and accessed, for example, by modifying standard database techniques such as the indexed sequential access method (ISAM) or B+ trees. In addition, it might be interesting to investigate to what extent existing optimization techniques for query evaluation such as cost-based query execution planning can be used in the context of our runtime monitoring framework [JK84, Ioa96].

Applications. In this dissertation, we exclusively focussed on detecting violations or satisfactions of complex properties. In areas such as complex event processing or in the context of database triggers, the detection of a situation of interest typically triggers an action as specified in a well-defined action language. For many applications, it would be interesting to gain a better understanding of how these two aspects fit together and under what conditions which types of action specifications can achieve which effects.

Experiments. Finally, we would like to extend our experiments to better assess the relative performance of our runtime monitors. In particular, this requires a more comprehensive analysis of the steady-state mean space consumption and steady-state mean processing times of our approach, including the investigation of additional formula classes and more realistic input distributions.

Appendix A

Statistical Background

To analyze the space consumption of our online runtime monitors, we used statistical techniques from simulation theory, in particular, for the analysis of outputs produced by non-terminating, discrete-time event simulations [Law80, Ale07, Law07]. In the following, we briefly explain these techniques and relate them to our setting.

For the remainder of this chapter, let $S = (\mathsf{C}, \mathsf{R}, a)$ be a signature, (D, τ) with $D = (D_0, D_1, \dots)$ and $\tau = (\tau_0, \tau_1, \dots)$ be a timed temporal database over S, and let ϕ be a bounded and TSF domain independent MFOTL formula over S.

A.1 Runtime Monitors and Stochastic Processes

From a statistical perspective, the timed temporal database (D, τ) is a realization of a (discrete-time) stochastic process, i.e., a sequence of random variables defined over a common set of possible outcomes, called the sample space. Recall from basic probability theory that a random variable $X : \Omega \to \mathbb{R}$ is a function that maps all elements of some sample space Ω to real numbers. Moreover, the (cumulative) distribution function $F(x)$ of a random variable X is defined as $F(x) = P(X \leq x)$, for each $x \in \mathbb{R}$, where $P(X \leq x)$ denotes the probability that the random variable X will take on a value no larger than x (see, e.g., [Law07]). As the runtime monitor $\mathcal{M}(\phi)$ incrementally processes (D, τ), it produces a realization of some stochastic output process. A random variable of the output stochastic process may represent an arbitrary characteristic of the runtime monitor such as the set of tuples that violate the monitored property at the current time point, the amount of space consumed by the runtime monitor at the given time point, or the time required to compute one of the previously mentioned sets for the actual time point. From a statistical viewpoint, $\mathcal{M}(\phi)$ can thus be seen as a function that maps a stochastic input process to a stochastic output process. Each realization of the stochastic input process thus produces a new realization of the stochastic output process. To statistically analyze the performance of $\mathcal{M}(\phi)$, several independent replications (i.e., single runs of $\mathcal{M}(\phi)$) are required.

In our analysis, the random variables of the stochastic output process represented the space consumption of the monitor $\mathcal{M}(\phi)$ at the respective time point.

A.2 Transient and Steady-state Behavior

To make valid statistical estimates about some characteristic of $\mathcal{M}(\phi)$, we must distinguish between its transient and steady-state behavior.

Let Ω be a sample space and let $(Y_0, Y_1, \dots)$ be a stochastic process, where each Y_i is a random variable over Ω. For the sake of concreteness, assume that the Y_is represent the space consumption of $\mathcal{M}(\phi)$ at each time point $i \in \mathbb{N}$. Moreover, for $i \in \mathbb{N}$, a real number $y \in \mathbb{R}$, and some initial conditions I, let $F_i(y|I) = P(Y_i \leq y|I)$ represent the conditional probability that the random variable Y_i takes on a value no larger than y, given the initial conditions I. In our concrete setting, for example, I captures the fact that all auxiliary relations are initially empty. The distribution $F_i(y|I)$ is called the *transient distribution* of the stochastic process at time point i for the initial conditions I. It is usually the case that $F_i(y|I) \neq F_j(y|J)$, for all time points $i, j \in \mathbb{N}$ and initial conditions I, J with $I \neq J$. If there is a distribution $F(y)$ such that $F_j(y|I)$ approximates $F(y)$ as j approaches ∞, for all $y \in \mathbb{R}$ and a set of initial conditions I, then $F(y)$ is called the *steady-state distribution* of the stochastic process $(Y_0, Y_1, \dots)$.

While the steady-state distribution only occurs in the limit (i.e., as i approaches ∞), there is often a finite time point $k \in \mathbb{N}$ such that for all $j > k$ the distributions $F_j(y|I)$ will be approximately the same. In this case, the analyzed system (i.e., $\mathcal{M}(\phi)$ in our concrete setting) is said to be in steady state. Note that the steady-state distribution $F(y)$ does not depend on the the initial conditions I. The finite sequence of time points $(0, 1, \dots, k)$ then represents the so-called *warm-up phase*. We emphasize that steady state does not mean that the random variables $Y_{k+1}, Y_{k+2}, \dots$ will all take on the same value in a particular realization of the stochastic process. Instead, in the steady state, the random variables all have approximately the same distribution. Moreover, the random variables are not necessarily independent but they approximately represent a covariance-stationary process. A discrete-time stochastic process $(X_0, X_1, \dots)$ is called *covariance-stationary* if both its mean and its variance are stationary over all time points and the covariance between two observations X_i and X_{i+j} with $i, j \in \mathbb{N}$ depends only on the difference j and not on the actual time points i and $i + j$. See [Law07] for a formal definition.

A.3 Steady-state Analysis

To analyze the performance of our runtime monitors, we focussed on steady-state performance parameters, specifically, on the space consumption in steady state. The rea-

son for focussing on the steady-state behavior is that online runtime monitors process possibly infinite timed temporal structures.

Let $\tilde{Y} = (Y_0, Y_1, \dots)$ be a stochastic output process and I some initial conditions. Moreover, let Y denote the steady-state random variable of interest and assume for $F_i(y|I) = P(Y_i \leq y|I)$ and $F(y) = P(Y \leq y)$ that $F_i(y|I)$ approximates $F(y)$ as i approaches ∞. A steady-state parameter ζ then is a characteristic function of Y such as the expected value (i.e., the mean) $E(Y)$ of Y. In our concrete setting, we are interested in the expected space consumption of our monitors in the steady state.

A.3.1 Problem of the Initial Transient

Because of the initial conditions I, estimating ζ from a finite prefix $(Y_0, Y_1, \dots, Y_{m-1})$ of the stochastic process $\tilde{Y}$ is difficult, for any $m \in \mathbb{N}$. The problem is that the distributions $F_i(y|I)$ are, in general, different from $F(y)$ because I is usually not representative of the steady-state behavior. As a result, any estimation of ζ based on a finite prefix of $\tilde{Y}$ will typically be biased and thus not representative of the true steady-state parameter. This is known as the problem of the initial transient in the simulation theory literature [Law07]. For example, if each random variable of the stochastic output process represents the space consumption of the runtime monitor at the respective time point, the sample mean $\bar{Y} = \sum_{i=0}^{m-1} Y_i/m$ is a biased estimator of $\nu = E(Y)$, for any $m \in \mathbb{N}$.

To overcome this problem, in our analysis, we used the approach of Welch [Wel83] to first determine the length $l \in \mathbb{N}$ of the warm-up phase. In a next step, the first l time points are deleted from $\tilde{Y}$. By deleting the first l time points from the observed realization of the stochastic output process, only the remaining realizations of the random variables in the stochastic process are considered to estimate ν. Note that the approach of Welch requires the generation of $n \geq 5$ replications of $\tilde{Y}$, each of sufficient length $m \in \mathbb{N}$ [Wel83].

A.3.2 Method of Batch Means

We now briefly present the method of batch means to obtain an (approximately) unbiased point estimate and a confidence interval for the mean ν of the stochastic output process $\tilde{Y}$. The advantage of this method is that it requires only a single but long replication of the monitoring experiment.

Assume that $l \in \mathbb{N}$ is the length of the warm-up period and that the number of observations $m \in \mathbb{N}$ is significantly larger than l. Let $X = (X_0, X_1, \dots, X_{m-1})$ denote the covariance-stationary process that results from deleting the first l observations from the prefix $(Y_0, Y_1, \dots, Y_{m+l-1})$ of $\tilde{Y}$. We can then divide the observations $X_0, X_1, \dots, X_{m-1}$ into n batches, each of length k, where $n, k \in \mathbb{N}$ and $m = nk$. The first batch consists of the observations $X_0, X_1, \dots, X_{k-1}$, the second batch of the observations $X_k, \dots, X_{2k-1}$,

and so on. For each batch $j \in \{1, \ldots, n\}$, we then determine the sample batch mean

$$\bar{X}_j(k) = \frac{\sum_{i=(j-1)k}^{jk-1} X_i}{k}.$$

Furthermore, we compute the grand sample mean $\bar{X}(n,k) = \sum_{j=1}^{n} \bar{X}_j/n = \sum_{i=0}^{m-1} X_i/m$.

For a large enough batch size k, the $\bar{X}_j$s are approximately uncorrelated and normally distributed (see [Law80]). Moreover, because $\tilde{Y}$ is assumed to be covariance-stationary with $E(Y) = \nu$, the $\bar{X}_j$s are asymptotically distributed (as k approaches ∞) as independent random variables with the same mean and variance. Consequently, an approximately unbiased point estimator for ν as well as an approximate $100(1-\alpha)$ percent confidence interval for ν can be computed by

$$\bar{X}(n,k) \pm t_{n-1,1-\alpha/2}\sqrt{\frac{S^2(n)}{n}}, \tag{A.1}$$

where

$$S^2(n) = \frac{\sum_{j=1}^{n} \left[\bar{X}_j(k) - \bar{X}(n,k)\right]^2}{n-1},$$

and $t_{n-1,1-\alpha/2}$ denotes the upper $1-\alpha/2$ critical point for the t-distribution with $n-1$ degrees of freedom.

A.3.3 Replication/Deletion Approach

As an alternative to the method of batch means, we now briefly present the replication/deletion approach to obtain an (approximately) unbiased point estimate and a confidence interval for the mean ν of the stochastic output process $\tilde{Y}$. The approach requires $n \geq 2$ replications of the monitoring experiment, each of length $m \in \mathbb{N}$, where m must be significantly larger than the length of the warm-up phase as determined using Welch's approach. Let $l \in \mathbb{N}$ be the length of the warm-up period and let Y_{ji} denote the ith observation of the jth replication, where $i \in \{0, 1, \ldots, m-1\}$ and $j \in \{0, 1, \ldots, n-1\}$. We then use the steady-state observations $Y_l, \ldots, Y_{m-1}$ to determine

$$X_j = \frac{\sum_{i=l}^{m-1} Y_{ji}}{m-l},$$

for each $j \in \{0, 1, \ldots, n-1\}$. Because the X_js are independent and identically distributed random variables with $E(X_j) \approx \nu$ (see [Law07]), we can compute an approximately unbiased point estimator for ν as well as an approximate $100(1-\alpha)$ percent confidence interval for ν by

$$\bar{X}(n) \pm t_{n-1,1-\alpha/2}\sqrt{\frac{S^2(n)}{n}}, \tag{A.2}$$

where as usual the sample mean $\bar{X}$(n) is defined as

$$\bar{X}(n) = \frac{\sum_{i=0}^{n-1} X_i}{n},$$

the sample variance $S^2(n)$ is defined as

$$S^2(n) = \frac{\sum_{i=0}^{n-1} \left[X_i - \bar{X}\right]^2}{n-1},$$

and $t_{n-1,1-\alpha/2}$ denotes the upper $1-\alpha/2$ critical point for the t-distribution with $n-1$ degrees of freedom.

Appendix B

Concrete Syntax

In the following, we provide the ANTLR v3 (ANother Tool for Language Recognition) grammar files that define the concrete textual syntax used to specify formulae in our runtime monitoring framework. ANTLR v3 generates efficient Java lexer and parser code from declarative lexer or parser grammars. See [Par07] for more information.

B.1 Signature Definition

The following ANTLR v3 grammar specifies the concrete syntax that is used to define new signatures.

```
// Signatures.g
grammar Signatures;
options {
    language = Java;
    output = AST;
    ASTLabelType = CommonTree;
}
// package names
@header{ package com.ibm.zurich.monitoring.language; }
@lexer::header{ package com.ibm.zurich.monitoring.language; }

// Parser
signatures : signatureName^ constantDef? predicateDef? EOF!;
signatureName : NAME ':'!;
constantDef : CDECLARATION^ ':'! CNAME (','! CNAME)* ';'!;
predicateDef : PDECLARATION^ ':'! predicate (','! predicate)*';'!;
// a predicate is either nullary or n-ary, where n>0
predicate : ( NAME^ | NAME^ '('! NAME (','! NAME)* ')'!);

// Lexer
CDECLARATION : ('c'|'C')('o'|'O')('n'|'N')('s'|'S')('t'|'T')
               ('a'|'A')('n'|'N')('t'|'T')('s'|'S');
PDECLARATION : ('p'|'P')('r'|'R')('e'|'E')('d'|'D')('i'|'I')
```

```
            ('c'|'C')('a'|'A')('t'|'T')('e'|'E')('s'|'S');
CNAME   : ( STRING_LITERAL | NUMBER );

//Smaller building blocks
fragment NUMBER : INTEGER | FLOAT;
fragment FLOAT: INTEGER '.' '0'..'9'+;
fragment INTEGER : POS_INT | NEG_INT;
fragment POS_INT : '0' | '+'? '1'..'9' '0'..'9'*;
fragment NEG_INT : '-' '1'..'9' '0'..'9'*;
NAME : LETTER (LETTER | DIGIT | '_')*;
fragment STRING_LITERAL : '"' NONCONTROL_CHAR* '"';
fragment NONCONTROL_CHAR : LETTER | DIGIT | SYMBOL | SPACE;
fragment LETTER : LOWER | UPPER;
fragment LOWER: 'a'..'z';
fragment UPPER: 'A'..'Z';
fragment DIGIT: '0'..'9';
fragment SPACE: ' ' | '\t';
fragment SYMBOL : '!' | '#'..'/' | ':'..'@' | '['..'`' | '{'..'~';
fragment NEWLINE : ('\r\n'|'\r'|'\n');
WS : (SPACE | NEWLINE)+ {skip();};
```

B.2 Property Specification

The following ANTLR v3 grammar defines the concrete MFOTL syntax that is used to express properties that shall be monitored.

```
// MFOTL.g
grammar MFOTL;
options {
language = Java;
output = AST;
ASTLabelType = CommonTree;
}
// package names
@header{ package com.ibm.zurich.monitoring.language; }
@lexer::header{ package com.ibm.zurich.monitoring.language; }

// Parser
mfotlFormula : formula^ EOF!;
formula : unaryTemporalFormula
        (binary_temp_operator^ interval unaryTemporalFormula)*;
unaryTemporalFormula : (unary_temp_operator^ interval)? quantifiedFormula;
quantifiedFormula : (quantifier^ VNAME ('.'!? VNAME)* '.'!)? impliesFormula;
impliesFormula : orFormula (IMPLIES^ orFormula)*;
orFormula : andFormula (OR^ andFormula)*;
andFormula : notFormula (AND^ notFormula)*;
notFormula : NOT^? (atomicFormula | LPAREN! formula RPAREN!);
```

```
atomicFormula :
              ( term EQUALS^ term
              | term SMALLER_THAN^ term
              | term LARGER_THAN^ term
              | term SMALLER_EQUALS^ term
              | term LARGER_EQUALS^ term
              | predicate
              | booleanValue
              );

// User-defined predicates (as declared in a separate signature)
predicate : PNAME^ LPAREN! t+=term? (','! t+=term)* RPAREN!;
term returns : (constant|variable);
constant : CNAME;
variable : VNAME;
booleanValue : (TRUE|FALSE);
quantifier : (EX_QUANTIFIER^|UNIV_QUANTIFIER^);
unary_temp_operator : (PREVIOUS|NEXT|ONCE|PAST_ALWAYS|EVENTUALLY|ALWAYS);
binary_temp_operator : (SINCE|UNTIL);
interval : LBOUND ','! RBOUND;

// Lexer
// Boolean primitives
TRUE : ('t'|'T')('r'|'R')('u'|'U')('e'|'E');
FALSE : ('f'|'F')('a'|'A')('l'|'L')('s'|'S')('e'|'E');
//Boolean Connectives
NOT : ('n'|'N')('o'|'O')('t'|'T');
AND : ('a'|'A')('n'|'N')('d'|'D');
OR  : ('o'|'O')('r'|'R'); //'OR';
IMPLIES : ('i'|'I')('m'|'M')('p'|'P')('l'|'L')('i'|'I')('e'|'E')('s'|'S');
//Quantifiers
EX_QUANTIFIER : ('e'|'E')('x'|'X')('i'|'I')('s'|'S')('t'|'T')('s'|'S');
UNIV_QUANTIFIER : ('f'|'F')('o'|'O')('r'|'R')('a'|'A')('l'|'L')('l'|'L');
//Temporal operators
SINCE : ('s'|'S')('i'|'I')('n'|'N')('c'|'C')('e'|'E');
UNTIL : ('u'|'U')('n'|'N')('t'|'T')('i'|'I')('l'|'L');
PREVIOUS : ('p'|'P')('r'|'R')('e'|'E')('v'|'V')
           ('i'|'I')('o'|'O')('u'|'U')('s'|'S');
NEXT  : ('n'|'N')('e'|'E')('x'|'X')('t'|'T');
ONCE  : ('o'|'O')('n'|'N')('c'|'C')('e'|'E');
PAST_ALWAYS  :
             (('p'|'P')('a'|'A')('s'|'S')('t'|'T')('_')?
             ('a'|'A')('l'|'L')('w'|'W')('a'|'A')('y'|'Y')('s'|'S')
             |('a'|'A')('l'|'L')('w'|'W')('a'|'A')('y'|'Y')('s'|'S')('_')?
             ('p'|'P')('a'|'A')('s'|'S')('t'|'T')
             );
EVENTUALLY  :
```

```
            (('e'|'E')('v'|'V')('e'|'E')('n'|'N')('t'|'T')('u'|'U')
            ('a'|'A')('l'|'L')('l'|'L')('y'|'Y')
            | ('s'|'S')('o'|'O')('m'|'M')('e'|'E')('t'|'T')('i'|'I')
            ('m'|'M')('e'|'E')('s'|'S')
            );
ALWAYS  : ('a'|'A')('l'|'L')('w'|'W')('a'|'A')('y'|'Y')('s'|'S');
//Interval bounds
LBOUND : '[' INTEGER;
RBOUND:  (INTEGER|INFTY) ']';
//Built-in predicates
EQUALS  : '=';
SMALLER_THAN : '<';
LARGER_THAN : '>';
SMALLER_EQUALS : '=<';
LARGER_EQUALS : '>=';

PNAME : NAME;
//Parentheses
LPAREN  : '(';
RPAREN  : ')';

// Variables (always start with a question mark)
VNAME : '?'NAME;
CNAME : ('"' STRING_LITERAL '"' | NUMBER);

//Smaller building blocks
fragment INFTY : '*';
fragment NUMBER : INTEGER | FLOAT;
fragment FLOAT: INTEGER '.' '0'..'9'+;
fragment INTEGER : POS_INT | NEG_INT;
fragment POS_INT : '0' | '+'? '1'..'9' '0'..'9'*;
fragment NEG_INT : '-' '1'..'9' '0'..'9'*;
NAME : LETTER (LETTER | DIGIT | '_')*;
fragment STRING_LITERAL : '"' NONCONTROL_CHAR* '"';
fragment NONCONTROL_CHAR : LETTER | DIGIT | SYMBOL | SPACE;
fragment LETTER : LOWER | UPPER;
fragment LOWER: 'a'..'z';
fragment UPPER: 'A'..'Z';
fragment DIGIT: '0'..'9';
fragment SPACE: ' ' | '\t';
fragment SYMBOL : '!' | '#'..'/' | ':'..'@' | '['..'`' | '{'..'~';
fragment NEWLINE : ('\r\n'|'\r'|'\n'); %
WS : (SPACE | NEWLINE)+ {skip();};
```

Appendix C

Signature Definitions and Formulae

In the following, we present the signature definitions and the formulae that we used to validate the feasibility of monitoring the properties of our case studies.

C.1 Compliance

We first present the signature definition. We then present the respective formulae one by one.

C.1.1 Signature Definition

The following is the definition of the signature S_{CFR}.

```
ComplianceSignature:
    constants:
        "failed";
    predicates:
        open(aID),
        close(aID),
        obtainInfo(aID),
        retainInfo(aID),
        discardInfo(aID),
        verifyID(aID,result);
```

C.1.2 Formulae

In the following, we present the formulae used to detect violations of the example requirements from Section 7.2. All metric constraints were expressed with seconds as the basic time unit.

Formula for Requirement (1)

```
open(?a) AND (NOT (once[0,*] obtainInfo(?a)))
```

Formula for Requirement (2)

```
open(?a) AND (NOT (sometimes[0,172800] (exists ?r. verifyID(?a,?r))))
```

Formula for Requirement (3)

```
verifyID(?a,"failed") AND (NOT (sometimes[0,259200] close(?a)))
```

Formula for Requirement (4)

```
obtainInfo(?a) AND (NOT (once[0,86400] retainInfo(?a)))
```

Formula for Requirement (5)

```
discardInfo(?a) AND (NOT (once[63072000,*] close(?a)))
```

C.2 Separation of Duties

We first present the definition of the signature S_{SoD} in our concrete syntax. We then present the formulae using for monitoring.

C.2.1 Signature Definition

The signature S_{CFR} is defined as follows.

```
SoDsignature:
    predicates:
        U_S(uID), U_F(uID),
        R_S(rID), R_F(rID),
        A_S(aID), A_F(aID),
        O_S(oID), O_F(oID),
        S_S(sID), S_F(sID),
        UA_S(uID,rID), UA_F(uID,rID),
        PA_S(rID,aID,oID), PA_F(rID,aID,oID),
        user_S(sID,uID), user_F(sID,uID),
        roles_S(sID,rID), roles_F(sID,rID),
        X_S(rID,rID), X_F(rID,rID),
        PD_S(aID), PD_F(aID),
        Exec(sID,aID,oID);
```

C.2.2 Formulae

In the following, we provide the formulae used to detect violations of the SoD constraints presented in Section 8.2.

Simple Dynamic Separation of Duty

```
((NOT S_F(?s)) since[0,*]S_S(?s))
 AND ((NOT R_F(?r)) since[0,*]R_S(?r))
 AND ((NOT R_F(?r1)) since[0,*]R_S(?r1))
 AND ((NOT X_F(?r,?r1)) since[0,*]X_S(?r,?r1))
 AND ((NOT roles_F(?s,?r)) since[0,*]roles_S(?s,?r))
 AND ((NOT roles_F(?s,?r1)) since[0,*]roles_S(?s,?r1))
```

Dynamic Object-based Separation of Duty

```
((NOT S_F(?s)) since[0,*]S_S(?s))
 AND ((NOT A_F(?a)) since[0,*]A_S(?a))
 AND ((NOT A_F(?a1)) since[0,*]A_S(?a1))
 AND ((NOT O_F(?o)) since[0,*]O_S(?o))
 AND ((NOT R_F(?r)) since[0,*]R_S(?r))
 AND ((NOT R_F(?r1)) since[0,*]R_S(?r1))
 AND (exec(?s,?a,?o)
 AND ((NOT roles_F(?s,?r)) since[0,*]roles_S(?s,?r))
 AND ((NOT X_F(?r,?r1)) since[0,*]X_S(?r,?r1))
 AND ((NOT PA_F(?r,?a,?o)) since[0,*] PA_S(?r,?a,?o))
 AND ((NOT PA_F(?r1,?a1,?o)) since[0,*] PA_S(?r1,?a1,?o))
 AND (NOT ((NOT exec(?s,?a1,?o)) since[0,*] ((roles_S(?s,?r1)
      AND ((NOT PA_F(?r1,?a1,?o)) since[0,*] PA_S(?r1,?a1,?o))))))
```

Dynamic Operational Separation of Duty

```
((exists ?a1.?a2. (((NOT A_F(?a1)) since[0,*]A_S(?a1))
                   AND ((NOT A_F(?a2)) since[0,*]A_S(?a2))
                   AND NOT (?a1 = ?a2)))
 AND
((NOT U_F(?u)) since[0,*]U_S(?u))
 AND NOT (exists ?a. ((NOT A_F(?a)) since[0,*]A_S(?a))
      AND ((NOT PD_F(?a)) since[0,*]PD_S(?a))
      AND (NOT (exists ?s.?r.?o.
                ((NOT S_F(?s)) since[0,*]S_S(?s))
                 AND ((NOT R_F(?r)) since[0,*]R_S(?r))
                 AND ((NOT O_F(?o)) since[0,*]O_S(?o))
                 AND ((NOT user_F(?s,?u)) since[0,*]user_S(?s,?u))
                 AND ((NOT roles_F(?s,?r)) since[0,*]roles_S(?s,?r))
                 AND ((NOT PA_F(?r,?a,?o)) since[0,*] PA_S(?r,?a,?o))))))
```

Dynamic History-based Separation of Duty

```
((NOT PD_F(?a)) since[0,*]PD_S(?a))
 AND (forall ?a1.
        (((NOT A_F(?a1)) since[0,*]A_S(?a1))
         AND ((NOT PD_F(?a1)) since[0,*]PD_S(?a1)))
         IMPLIES (exists ?s. ((NOT S_F(?s)) since[0,*]S_S(?s))
```

```
            AND exec(?s,?a1,?o)
            AND ((NOT user_F(?s,?u)) since[0,*]user_S(?s,?u))))
AND (exists ?s1.((NOT S_F(?s1)) since[0,*]S_S(?s1))
        AND exec(?s1,?a,?o)
        AND ((NOT user_F(?s1,?u)) since[0,*]user_S(?s1,?u)))
```

References

[AAB+06] Lisa Amini, Henrique Andrade, Ranjita Bhagwan, Frank Eskesen, Richard King, Philippe Selo, Yoonho Park, and Chitra Venkatramani. SPC: a distributed, scalable platform for data mining. In *Proceedings of the 4th International Workshop on Data Mining Standards, Services and Platforms (DMSSP '06)*, pages 27–37. ACM, 2006.

[ABB+04] Arvind Arasu, Brian Babcock, Shivnath Babu, Jon McAlister, and Jennifer Widom. Characterizing memory requirements for queries over continuous data streams. *ACM Transactions on Database Systems*, 29(1):162–194, 2004.

[ABc+05] Yanif Ahmad, Bradley Berg, Ugur Çetintemel, Mark Humphrey, Jeong-Hyon Hwang, Anjali Jhingran, Anurag Maskey, Olga Papaemmanouil, Alex Rasin, Nesime Tatbul, Wenjuan Xing, Ying Xing, and Stanley B. Zdonik. Distributed operation in the borealis stream processing engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '05)*, pages 882–884. ACM, 2005.

[ABG+05] Cyrille Artho, Howard Barringer, Allen Goldberg, Klaus Havelund, Sarfraz Khurshid, Mike Lowry, Corina Pasareanu, Grigore Roşu, Koushik Sen, Willem Visser, and Rich Washington. Combining test case generation and runtime verification. *Theoretical Computer Science*, 336(2-3):209–234, 2005.

[ABLS05] Oliver Arafat, Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification revisited. Technical Report TUM-I0518, Technische Universität München, 2005.

[ABW03] Arvind Arasu, Shivnath Babu, and Jennifer Widom. CQL: A language for continuous queries over streams and relations. In *Proceedings of the 9th International Workshop on Database Programming Languages*, volume 2921 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2003.

[ABW06] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: semantic foundations and query execution. *VLDB Journal*, 15(2):121–142, 2006.

[ACG+04] Arvind Arasu, Mitch Cherniack, Eduardo F. Galvez, David Maier, Anurag Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road: A stream data management benchmark. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB '04)*, pages 480–491. Morgan Kaufmann, 2004.

[ACM02] Eugene Asarin, Paul Caspi, and Oded Maler. Timed regular expressions. *Journal of the ACM*, 49(2):172–206, 2002.

[AD90] Rajeev Alur and David L. Dill. Automata for modeling real-time systems. In *Proceedings of the 17th International Colloquium on Automata, Languages and Programming (ICALP '90)*, volume 443 of *Lecture Notes in Computer Science*, pages 322–335. Springer, 1990.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[ADGI08] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. Efficient pattern matching over event streams. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*, pages 147–160. ACM, 2008.

[ADW05] Mingsheng Hong Mirek Riedewald Alan Demers, Johannes Gehrke and Walker White. A general algebra and implementation for monitoring event streams. Technical Report TR2005-1997, Cornell University, 2005.

[AF03] Martín Abadi and Cédric Fournet. Access control based on execution history. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2003)*. The Internet Society, 2003.

[AH92] Rajeev Alur and Thomas A. Henzinger. Logics and models of real time: A survey. In *Proceedings of the REX Workshop on Real Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 74–106. Springer, 1992.

[AH94] Rajeev Alur and Thomas A. Henzinger. A really temporal logic. *Journal of the ACM*, 41(1):181–204, 1994.

[AH99] Serge Abiteboul and Laurent Herr. Temportal connectives versus explicit timestamps to query temporal databases. *Journal of Computer and System Sciences*, 58(1):54–68, 1999.

[AHB03] Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. *Journal on Software Testing, Verification and Reliability*, 13(4):207–227, 2003.

[AHdB95] Serge Abiteboul, Laurent Herr, and Jan Van den Bussche. Temporal connectives versus explicit timestamps in temporal query languages. In *Proceedings of the International Workshop on Temporal Databases*, pages 43–57. Springer, 1995.

[AHdB96] Serge Abiteboul, Laurent Herr, and Jan Van den Bussche. Temporal versus first-order logic to query temporal databases. In *Proceedings of the 15th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '96)*, pages 49–57. ACM, 1996.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[AK07] Henrik Reif Andersen and Kåre J. Kristoffersen. Temporal runtime verification using monadic difference logic. *Computing Research Repository*, abs/0705.4604, 2007.

[Ale07] Christos Alexopoulos. Statistical analysis of simulation output: state of the art. In *Proceedings of the 39th Conference on Winter Simulation (WSC '07)*, pages 150–161. IEEE, 2007.

[Art05] Cyrille Artho. *Combining Static and Dynamic Analysis to Find Multi-threading Faults Beyond Data Races*. PhD thesis, ETH Zürich, 2005.

[AS85] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 1985.

[AS87] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[AS99] Gail-Joon Ahn and Ravi Sandhu. The RSL99 language for role-based separation of duty constraints. In *Proceedings of the 4th ACM workshop on role-based access control (RBAC '99)*, pages 43–54. ACM, 1999.

[AS06] Rahul Agarwal and Scott D. Stoller. Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables. In *Proceedings of the 4th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, pages 51–60. ACM, 2006.

[AvKM+06] Carl Abrams, Jürg von Känel, Samuel Müller, Birgit Pfitzmann, and Susanne Ruschka-Taylor. Optimized enterprise risk management. *IBM Systems Journal*, 46(2), 2006.

[AYL06] Ke Xu Alice Y. Liu, Samuel Müller. A static compliance checking framework for business process models. *IBM Systems Journal*, 46(2), 2006.

[Bae01] Jos C. M. Baeten. Timed process algebras. *Electronic Notes in Theoretical Computer Science*, 52(3):207–208, 2001.

[Bae05] Jos C. M. Baeten. A brief history of process algebra. *Theoretical Computer Science*, 335(2-3):131–146, 2005.

[BBD+02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '02)*, pages 1–16. ACM, 2002.

[BBK09] David Basin, Samuel J. Burri, and Günter Karjoth. Dynamic enforcement of abstract separation of duty constraints. IBM Research Report RZ 3726, IBM Research, 2009.

[BCM05] Patricia Bouyer, Fabrice Chevalier, and Nicolas Markey. On the expressiveness of TPTL and MTL. In *Proceedings of the 25th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS '05)*, volume 3821 of *Lecture Notes in Computer Science*, pages 432–443. Springer, 2005.

[BFMW01] Detlef Bartetzko, Clemens Fischer, Michael Möller, and Heike Wehrheim. Jass - Java with assertions. *Electronic Notes in Theoretical Computer Science*, 55(2):103–117, 2001.

[BG98] Lars Bækgaard and Jens Christian Godskesen. Real-time event control in active databases. *Journal of Systems and Software*, 42(3):263–271, 1998.

[BG04] Achim Blumensath and Erich Grädel. Finite presentations of infinite structures: Automata and interpretations. *Theory of Computing Systems*, 37(6):641–674, 2004.

[BGHS04a] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Program monitoring with LTL in EAGLE. *18th International Parallel and Distributed Processing Symposium (IPDPS '04)*, 17:264b, 2004.

[BGHS04b] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In *Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '04)*, volume 2937 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2004.

[BH06] Saddek Bensalem and Klaus Havelund. Dynamic deadlock analysis of multi-threaded programs. In *Revised Selected Papers of the First International Haifa Verification Conference on Hardware and Software Verification and Testing*, volume 3875 of *Lecture Notes in Computer Science*, pages 208–223. Springer, 2006.

[BJ08] Andreas Bauer and Jan Juerjens. Security protocols, properties, and their monitoring. In *Proceedings of the 4th international workshop on software engineering for secure systems (SESS '08)*, pages 33–40. ACM, 2008.

[BKMP08a] David Basin, Felix Klaedtke, Samuel Müller, and Birgit Pfitzmann. Runtime monitoring of metric first-order temporal properties. In *Proceedings of the 28th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS '08)*, Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2008.

[BKMP08b] David Basin, Felix Klaedtke, Samuel Müller, and Birgit Pfitzmann. Runtime monitoring of metric first-order temporal properties. Technical Report RZ 3702, IBM Research, 2008.

[BLS06] Andreas Bauer, Martin Leucker, and Christian Schallhart. Monitoring of real-time properties. In *Proceedings of the 26th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS '06)*, volume 4337 of *Lecture Notes in Computer Science*, pages 260–272. Springer, 2006.

[BLS07a] Andreas Bauer, Martin Leucker, and Christian Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In *Revised Selected Papers of the 7th International Workshop on Runtime Verification (RV '07)*, volume 4839 of *Lecture Notes in Computer Science*, pages 126–138, 2007.

[BLS07b] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. Technical Report TUM-I0724, Technische Universität München, 2007.

[Blu99] Achim Blumensath. Automatic structures. Diploma thesis, RWTH-Aachen, Aachen, Germany, 1999.

[BOS07] David Basin, Ernst-Ruediger Olderog, and Paul E. Sevinc. Specifying and analyzing security automata using CSP-OZ. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security (ASIACCS '07)*, pages 70–81. ACM, 2007.

[BRH07] Howard Barringer, David E. Rydeheard, and Klaus Havelund. Rule systems for run-time monitoring: From Eagle to RuleR. In *Proceedings of the*

7th International Workshop on Runtime Verification (RV '07), volume 4839 of *Lecture Notes in Computer Science*, pages 111–125. Springer, 2007.

[BSA70] Bank Secrecy Act of 1970, 1970. 31 USC 5311-5332 and 31 CFR 103.

[Büc62] Julius Richard Büchi. On a decision method in restricted second-order arithmetic. In *Proceedings of the 1960 International Congress on Logic, Methodology and Philosophy of Science*, pages 1–11. Stanford University Press, 1962.

[BW01] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *SIGMOD Record*, 30(3):109–120, 2001.

[BY03] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets, Advances in Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer, 2003.

[CcC+02] Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring streams: a new class of data management applications. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB '02)*, pages 215–226. VLDB Endowment, 2002.

[CdR04] Feng Chen, Marcelo d'Amorim, and Grigore Roşu. A formal monitoring-based framework for software development and analysis. In *Proceedings of the 6th International Conference on Formal Engineering Methods (ICFEM '04)*, volume 3308 of *Lecture Notes in Computer Science*, pages 357–372, 2004.

[Cho92a] Jan Chomicki. History-less checking of dynamic integrity constraints. In *Proceedings of the 8th International Conference on Data Engineering*, pages 557–564. IEEE Computer Society, 1992.

[Cho92b] Jan Chomicki. Real-time integrity constraints. In *Proceedings of the 11th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '92)*, pages 274–282. ACM, 1992.

[Cho94] Jan Chomicki. Temporal query languages: A survey. In *Proceedings of the 1st International Conference on Temporal Logic (ICTL '94)*, pages 506–534. Springer, 1994.

[Cho95] Jan Chomicki. Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Transactions on Database Systems*, 20(2):149–186, 1995.

[CL01] Jan Chomicki and Jorge Lobo. Monitors for history-based policies. In *Proceedings of the International Workshop on Policies for Distributed Systems and Networks (POLICY '01)*, pages 57–72. Springer, 2001.

[CN95] Jan Chomicki and Damian Niwiński. On the feasibility of checking temporal integrity constraints. *Journal of Computer and System Sciences*, 51(3):523–535, 1995.

[Cod70] Edgar F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[CR03] Feng Chen and Grigore Roşu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. *Electronic Notes in Theoretical Computer Science*, 89(2):1–20, 2003.

[CR05] Feng Chen and Grigore Roşu. Java-MOP: A monitoring oriented programming environment for java. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '05)*, volume 3440 of *Lecture Notes in Computer Science*, pages 546–550. Springer, 2005.

[CR06] Lori A. Clarke and David S. Rosenblum. A historical perspective on runtime assertion checking in software development. *SIGSOFT Software Engineering Notes*, 31(3):25–37, 2006.

[CS06] Alessandro Campi and Paola Spoletini. History checking of XML data streams. In *Proceedings of the 17th International Conference on Database and Expert Systems Applications (DEXA '06)*, pages 542–546. IEEE Computer Society, 2006.

[CT95] Jan Chomicki and David Toman. Implementing temporal integrity constraints using an active DBMS. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):566–582, 1995.

[CT98] Jan Chomicki and David Toman. Temporal logic in information systems. In Jan Chomicki and Gunter Saake, editors, *Logics for Databases and Information Systems*, chapter 3, pages 31–70. Kluwer Academic Publishers, 1998.

[CT05] Jan Chomicki and David Toman. Time in database systems. In Michael Fisher, Dov Gabbay, and Lluis Vila, editors, *Handbook of Temporal Reasoning in Artificial Intelligence*, chapter 19, pages 429–467. Elsevier, 2005.

[CTB01] Jan Chomicki, David Toman, and Michael H. Böhlen. Querying ATSQL databases with temporal logic. *ACM Transactions on Database Systems*, 26(2):145–178, 2001.

[Dem92] Robert Demolombe. Syntactical characterization of a subset of domain-independent formulas. *Journal of the ACM*, 39(1):71–94, 1992.

[DGa04] Nelly Delgado, Ann Quiroz Gates, and Steve Roach and. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering*, 30(12):859–872, 2004.

[DJLS08] Nikhil Dinesh, Aravind Joshi, Insup Lee, and Oleg Sokolsky. Checking traces for regulatory conformance. In *Proceedings of the 8th Workshop on Runtime Verification (RV '08)*, Lecture Notes in Computer Science. Springer, 2008.

[dR05] Marcelo d'Amorim and Grigore Roşu. Efficient monitoring of omega-languages. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV '05)*, volume 3576 of *Lecture Notes in Computer Science*, pages 364–378. Springer, 2005.

[Dru00] Doron Drusinsky. The Temporal Rover and the ATG Rover. In *Proceedings of the 7th International SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science*, pages 323–330. Springer, 2000.

[Dru03] Doron Drusinsky. Monitoring temporal rules combined with time series. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV '03)*, volume 2725 of *Lecture Notes in Computer Science*, pages 114–117. Springer, 2003.

[Dru06] Doron Drusinsky. On-line monitoring of metric temporal logic with time-series constraints using alternating finite automata. *Journal of Universal Computer Science*, 12(5):482–498, 2006.

[DSS+05] Ben D'Angelo, Sriram Sankaranarayanan, Csar Snchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, and Zohar Manna. LOLA: Runtime monitoring of synchronous systems. In *Proceedings of the 12th International Symposium on Temporal Representation and Reasoning (TIME '05)*, pages 166–174. IEEE Computer Society, 2005.

[DtSD05] Doron Drusinsky, Man tak Shing, and Kadir Alpaslan Demir. Test-time, run-time, and simulation-time temporal assertions in RSP. In *Proceedings of the 16th IEEE International Workshop on Rapid System Prototyping (RSP '05)*, pages 105–110. IEEE Computer Society, 2005.

[EFH+03] Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout. Reasoning with temporal logic on truncated paths. In *Proceedings 15th International Conference on Computer*

Aided Verification (CAV '03), volume 2725 of *Lecture Notes in Computer Science*, pages 27–39. Springer, 2003.

[EGNR98] Y. L. Ershov, S. S. Goncharov, A. Nerode, and J. B. Remmel, editors. *Handbook of Recursive Mathematics*. North Holland, 1998.

[Fag82] Ronald Fagin. Horn clauses and database dependencies. *Journal of the ACM*, 29(4):952–985, 1982.

[FS04] Bernd Finkbeiner and Henny Sipma. Checking finite traces using alternating automata. *Formal Methods in System Design*, 24(2):101–127, 2004.

[FSG+01] David F. Ferraiolo, Ravi S. Sandhu, Serban I. Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and Systems Security*, 4(3):224–274, 2001.

[GAW+08] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. SPADE: the System S declarative stream processing engine. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*, pages 1123–1134. ACM, 2008.

[GD00] Marc Geilen and Dennis Dams. An on-the-fly tableau construction for a real-time temporal logic. In *Proceedings of the 6th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT '00)*, volume 1926 of *Lecture Notes in Computer Science*, pages 276–290. Springer, 2000.

[Gei01] Marc Geilen. On the construction of monitors for temporal logic properties. *Electronic Notes in Theoretical Computer Science*, 55(2):181–199, 2001.

[Gei03] Marc Geilen. An improved on-the-fly tableau construction for a real-time temporal logic. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV '03)*, volume 2725 of *Lecture Notes in Computer Science*, pages 394–406. Springer, 2003.

[GGF98] Virgil D. Gligor, Serban I. Gavrila, and David Ferraiolo. On the formal definition of separation-of-duty policies and their composition. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 172–183. IEEE Computer Society, 1998.

[GH01] Dimitra Giannakopoulou and Klaus Havelund. Automata-based verification of temporal properties on running programs. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE '01)*, page 412. IEEE Computer Society, 2001.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, 1995.

[GJS92] Narain H. Gehani, Hosagrahar V. Jagadish, and Oded Shmueli. Event specification in an active object-oriented database. *SIGMOD Record*, 21(2):81–90, 1992.

[GL96] Michael Gertz and Udo W. Lipeck. Deriving optimized integrity monitoring triggers from dynamic integrity constraints. *Data and Knowledge Engineering*, 20(2):163–193, 1996.

[GLdB07] Yuri Gurevich, Dirk Leinders, and Jan Van den Bussche. A theory of stream queries. In *Proceedings of the 11th International Symposium on Database Programming Languages (DBPL '07)*, volume 4797 of *Lecture Notes in Computer Science*, pages 153–168. Springer, 2007.

[GLM+05] Christopher Giblin, Alice Y. Liu, Samuel Müller, Birgit Pfitzmann, and Xin Zhou. Regulations expressed as logical models (REALM). In *Proceedings of the 18th Annual Conference on Legal Knowledge and Information Systems (JURIX '05)*, volume 134 of *Frontiers in Artificial Intelligence and Applications*, pages 37–48. IOS Press, 2005.

[GMM90] Carlo Ghezzi, Dino Mandrioli, and Angelo Morzenti. TRIO: A logic language for executable specifications of real-time systems. *Journal of Systems and Software*, 12(2):107–123, 1990.

[GMP06] Christopher Giblin, Samuel Müller, and Birgit Pfitzmann. From regulatory policies to event monitoring rules: Towards model-driven compliance automation. IBM Research Report RZ 3662, IBM Research, 2006.

[GO03] Lukasz Golab and M. Tamer Özsu. Issues in data stream management. *SIGMOD Record*, 32(2):5–14, 2003.

[GOA05] Simon Goldsmith, Robert O'Callahan, and Alexander Aiken. Relational queries over program traces. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*, pages 385–402. ACM, 2005.

[GPSS80] Dov M. Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the temporal basis of fairness. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages (POPL '80)*, pages 163–173. ACM, 1980.

[GPVW96] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the*

15th IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV, pages 3–18. Chapman & Hall, 1996.

[GT91] Allen Van Gelder and Rodney W. Topor. Safety and translation of relational calculus. *ACM Transactions on Database Systems*, 16(2):235–278, 1991.

[HJL03] John Håkansson, Bengt Jonsson, and Ola Lundqvist. Generating online test oracles from temporal logic specifications. *International Journal on Software Tools for Technology Transfer*, 4(4):456–471, 2003.

[HLM+08] Anders Hessel, Kim Guldstrand Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing real-time systems using UPPAAL. In *Revised Selected Papers on Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 77–117. Springer, 2008.

[HLNW01] Lauri Hella, Leonid Libkin, Juha Nurmonen, and Limsoon Wong. Logics with aggregate operators. *Journal of the ACM*, 48(4):880–907, 2001.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[HPB+07] Manuel Hilty, Alexander Pretschner, David A. Basin, Christian Schaefer, and Thomas Walter. A policy language for distributed usage control. In *Proceedings of the 12th European Symposium On Research In Computer Security (ESORICS '07)*, volume 4734 of *Lecture Notes in Computer Science*, pages 531–546. Springer, 2007.

[HR01a] Klaus Havelund and Grigore Roşu. Monitoring Java programs with Java PathExplorer. *Electronic Notes in Theoretical Computer Science*, 55(2):200–217, 2001.

[HR01b] Klaus Havelund and Grigore Roşu. Monitoring programs using rewriting. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE '01)*, pages 135–143. IEEE Computer Society, 2001.

[HR01c] Klaus Havelund and Grigore Roşu. Testing linear temporal logic formulae on finite execution traces. Technical Report TR 01-08, Research Institute for Advanced Computer Science, 2001.

[HR02] Klaus Havelund and Grigore Roşu. Synthesizing monitors for safety properties. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '02)*, volume 2280 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002.

[HR04a] Klaus Havelund and Grigore Roşu. Efficient monitoring of safety properties. *International Journal on Software Tools and Technology Transfer*, 6(2):158–173, 2004.

[HR04b] Klaus Havelund and Grigore Roşu. An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design*, 24(2):189–215, 2004.

[HR04c] Yoram Hirshfeld and Alexander Moshe Rabinovich. Logics for real time: Decidability and complexity. *Fundamenta Informaticae*, 62(1):1–28, 2004.

[HS91] Klaus Hülsmann and Gunter Saake. Theoretical foundations of handling large substitution sets in temporal integrity monitoring. *Acta Informatica*, 28(4):365–407, 1991.

[HV08] Sylvain Hallé and Roger Villemaire. Runtime monitoring of message-based workflows with data. In *Proceedings of the 12th International IEEE Enterprise Distributed Object Computing Conference (EDOC '08)*, pages 63–72. IEEE Computer Society, 2008.

[Ioa96] Yannis E. Ioannidis. Query optimization. *ACM Computing Surveys*, 28(1):121–123, 1996.

[JK84] Matthias Jarke and Jürgen Koch. Query optimization in database systems. *ACM Computing Surveys*, 16(2):111–152, 1984.

[jms] J2EE java message service (JMS). `http://java.sun.com/products/jms/`.

[Kam68] H. Kamp. *Tense logic and the theory of linear order*. PhD thesis, University of California, 1968.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001.

[KKL+01] Moonjoo Kim, Sampath Kannan, Insup Lee, Oleg Sokolsky, and Mahesh Viswanathan. Java-MaC: a run-time assurance tool for java programs. *Electronic Notes in Theoretical Computer Science*, 55(2):218–235, 2001.

[KKR95] Paris C. Kanellakis, Gabriel M. Kuper, and Peter Z. Revesz. Constraint query languages. *Journal of Computer and System Sciences*, 51(1):26–52, 1995.

[Kla04] Felix Klaedtke. On the automata size for Presburger arithmetic. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science (LICS '04)*, pages 110–119. IEEE Computer Society, 2004.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.

[Klu82] Anthony C. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of the ACM*, 29(3):699–717, 1982.

[KMSF01] David Kortenkamp, Tod Milam, Reid G. Simmons, and Joaquín Lopez Fernández. Collecting and analyzing data from distributed control programs. *Electronic Notes in Theoretical Computer Science*, 55(2):236–254, 2001.

[KN95] Bakhadyr Khoussainov and Anil Nerode. Automatic presentations of structures. In *Proceedings of the International Workshop on Logical and Computational Complexity (LCC '94)*, volume 960 of *Lecture Notes in Computer Science*, pages 367–392. Springer, 1995.

[Koy90] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.

[KPA03] Kåre J. Kristoffersen, Christian Pedersen, and Henrik Reif Andersen. Runtime verification of Timed LTL using disjunctive normalized equation systems. *Electronic Notes in Theoretical Computer Science*, 89(2):1–16, 2003.

[KPL00] Gabriel Kuper, Jan Paredaens, and Leonid Libkin, editors. *Constraint Databases*. Springer, 2000.

[Kuh97] D. Richard Kuhn. Mutual exclusion of roles as a means of implementing separation of duty in role-based access control systems. In *Proceedings of the 2nd ACM Workshop on Role-based Access Control (RBAC '97)*, pages 23–30. ACM, 1997.

[Kun84] David Chenho Kung. A temporal framework for database specification and verification. In *Proceedings of 10th International Conference on Very Large Data Bases (VLDB '84)*, pages 91–99. Morgan Kaufmann, 1984.

[Kun85] David Chenho Kung. On verification of database temporal constraints. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data (SIGMOD '85)*, pages 169–179. ACM, 1985.

[KV01] Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.

[KVK+04] Moonzoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Java-MaC: A run-time assurance approach for java programs. *Formal Methods in System Design*, 24(2):129–155, 2004.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[Law80] Averill M. Law. A tutorial on statistical analysis of simulation output data. In *Proceedings of the 12th conference on Winter Simulation (WSC '80)*, pages 361–370. IEEE, 1980.

[Law07] Averill M. Law. *Simulation, Modeling & Analysis*. McGraw-Hill, 4th edition, 2007.

[LBW05] Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1-2):2–16, 2005.

[LF89] Udo W. Lipeck and Dasu Feng. Construction of deterministic transition graphs from dynamic integrity constraints. In *Proceedings of the 14th International Workshop on Graph-Theoretic Concepts in Computer Science*, volume 344 of *Lecture Notes in Computer Science*, pages 166–179. Springer, 1989.

[LKK+99] Insup Lee, Sampath Kannan, Moonjoo Kim, Oleg Sokolsky, and Mahesh Viswanathan. Runtime assurance based on formal specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '99)*, pages 279–287. CSREA Press, 1999.

[LMN04] Kim Guldstrand Larsen, Marius Mikucionis, and Brian Nielsen. Online testing of real-time systems using uppaal. In *Revised selected papers of the 4th International Workshop on Formal Approaches to Software Testing (FATES '04)*, volume 3395 of *Lecture Notes in Computer Science*, pages 79–94. Springer, 2004.

[LMS02] François Laroussinie, Nicolas Markey, and Ph. Schnoebelen. Temporal logic with forgettable past. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS '02)*, pages 383–392. IEEE Computer Society, 2002.

[log] Apache log4j. `http://logging.apache.org/log4j/index.html`.

[LPZ85] Orna Lichtenstein, Amir Pnueli, and Lenore D. Zuck. The glory of the past. In *Proceedings of the Conference on Logic of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 196–218. Springer, 1985.

[LS87] Udo Walter Lipeck and Gunter Saake. Monitoring dynamic integrity constraints based on temporal logic. *Information Systems*, 12(3):255–269, 1987.

[LS08] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 2008. in press.

[LW08] Ninghui Li and Qihua Wang. Beyond separation of duty: An algebra for specifying high-level security policies. *Journal of the ACM*, 55(3):1–46, 2008.

[LZ91] Udo W. Lipeck and Heren Zhou. Monitoring dynamic integrity constraints on finite state sequences and existence intervals. In *Third Workshop on Foundations of Models and Languages for Data and Objects (FMLDO), Informatik-Bericht 91/3*, pages 115–130, 1991.

[MDS03] Till Mossakowski, Michael Drouineaud, and Karsten Sohr. A temporal-logic extension of role-based access control covering dynamic separation of duties. In *Proceedings of the 10th International Symposium on Temporal Representation and Reasoning (TIME '03)*, pages 83–90. IEEE Computer Society, 2003.

[Mil82] Robin Milner. *A Calculus of Communicating Systems*. Springer, 1982.

[ML97] Aloysius K. Mok and Guangtian Liu. Efficient run-time monitoring of timing constraints. In *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, page 252. IEEE Computer Society, 1997.

[MLN04] Marius Mikucionis, Kim Guldstrand Larsen, and Brian Nielsen. T-UPPAAL: Online model-based testing of real-time systems. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE '04)*, pages 396–397. IEEE Computer Society, 2004.

[MNP06] Oded Maler, Dejan Nickovic, and Amir Pnueli. From MITL to timed automata. In *Proceedings of the 4th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS '06)*, volume 4202 of *Lecture Notes in Computer Science*, pages 274–289. Springer, 2006.

[MNP07] Oded Maler, Dejan Nickovic, and Amir Pnueli. On synthesizing controllers from bounded-response properties. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 95–107. Springer, 2007.

[MNP08] Oded Maler, Dejan Nickovic, and Amir Pnueli. Checking temporal properties of discrete, timed and continuous behaviors. In *Pillars of Computer Science, Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday*, volume 4800 of *Lecture Notes in Computer Science*, pages 475–505. Springer, 2008.

[Mor92] Angelo Morzenti. Validating real-time systems by executing logic specifications. In *Proceedings of the REX Workshop on Real-Time: Theory in Practice*, volume 600 of *Lecture Notes In Computer Science*, pages 502–525. Springer, 1992.

[MPW92a] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Information and Computation*, 100(1):1–40, 1992.

[MPW92b] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, II. *Information and Computation*, 100(1):41–77, 1992.

[Muk96] Madhavan Mukund. Finite state automata over infinite inputs. Technical Report TCS-96-2, Chennai Mathematical Institute, 1996.

[Mül06] Samuel Müller. A dependability perspective on enterprise compliance. IBM Research Report RZ 3667, IBM Research, 2006.

[NM07] Dejan Nickovic and Oded Maler. AMT: A property-based monitoring tool for analog systems. In *Proceedings of the 5th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS '07)*, volume 4763 of *Lecture Notes in Computer Science*, pages 304–319. Springer, 2007.

[NP90] Michael J. Nash and Keith R. Poland. Some conundrums concerning separation of duty. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 201–207. IEEE Computer Society, 1990.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 89–100. ACM, 2007.

[NST04] Prasad Naldurg, Koushik Sen, and Prasanna Thati. A temporal logic based framework for intrusion detection. In *Proceedings of the 24th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE '04)*, volume 3235 of *Lecture Notes in Computer Science*, pages 359–376. Springer, 2004.

[OOM87] G. Özsoyoğlu, Z. M. Özsoyoğlu, and V. Matos. Extending relational algebra and relational calculus with set-valued attributes and aggregate functions. *ACM Transactions on Database Systems*, 12(4):566–592, 1987.

[ÖS95] Gultekin Özsoyoglu and Richard T. Snodgrass. Temporal and real-time databases: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):513–532, 1995.

[Owe07] Thomas J. Owens. Survey of event processing. Technical Report AFRL-RI-RS-TM-2007-16, Air Force Research Laboratory, 2007.

[Pac97] Maria Amelia Pacheco e Silva. Dynamic integrity constraints definition and enforcement in databases: A classification framework. In *Proceedings of the IFIP TC11 Working Group 11.5, First Working Conference on Integrity and Internal Control in Information Systems*, pages 65–87. Chapman & Hall, Ltd., 1997.

[Pao69] Robert A. Di Paola. The recursive unsolvability of the decision problem for the class of definite formulas. *Journal of the ACM*, 16(2):324–327, 1969.

[Par07] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Bookshelf, 2007.

[Pat01] USA Patriot Act of 2001, 2001. Public Law 107-56, HR 3162 RDS.

[PN81] Bernhard Plattner and Jürg Nievergelt. Monitoring program execution: A survey. *IEEE Computer*, 14(11):76–93, 1981.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science (FOCS '77)*, pages 46–57. IEEE Computer Society, 1977.

[Pri67] Arthur N. Prior. *Past, Present and Future*. Oxford University Press, 1967.

[PZ06] Amir Pnueli and Aleksandr Zaks. PSL model checking and run-time verification via testers. In *Proceedings of the 14th International Symposium on Formal Methods (FM '06)*, volume 4085 of *Lecture Notes in Computer Science*, pages 573–586. Springer, 2006.

[RGL01] Muriel Roger and Jean Goubault-Larrecq. Log auditing through model-checking. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW '01)*, pages 220–234, 2001.

[RH05] Grigore Roşu and Klaus Havelund. Rewriting-based techniques for runtime verification. *Automated Software Engineering*, 12(2):151–197, 2005.

[RHKR01] Jürgen Ruf, Dirk W. Hoffmann, Thomas Kropf, and Wolfgang Rosenstiel. Simulation-guided property checking based on a multi-valued AR-automata. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE '01)*, pages 742–748. IEEE, 2001.

[Ros97] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.

[RR88] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58(1-3):249–261, 1988.

[Run08] *Proceedings of the 1st to 8th Workshop on Runtime Verification*, 2001–2008. `http://www.runtime-verification.org/`.

[RV03] Grigore Roşu and Mahesh Viswanathan. Testing extended regular language membership incrementally by rewriting. In *Proceedings of 14th International Conference on Rewriting Techniques and Applications (RTA '03)*, volume 2706 of *Lecture Notes in Computer Science*, pages 499–514. Springer, 2003.

[San88] Ravi S. Sandhu. Transaction control expressions for separation of duties. In *Proceedings of the 4th Aerospace Computer Security Applications Conference*, pages 282–286, 1988.

[SB06] Volker Stolz and Eric Bodden. Temporal assertions using aspectj. *Electronic Notes in Theoretical Computer Science*, 144(4):109–124, 2006.

[SBN+97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[SCFY96] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[Sch00] Fred B. Schneider. Enforceable security policies. *Information and System Security*, 3(1):30–50, 2000.

[SDJ+92] Steve Schneider, Jim Davies, D. M. Jackson, George M. Reed, Joy N. Reed, and A. W. Roscoe. Timed CSP: Theory and practice. In *Proceedings of the REX Workshop on Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 640–675. Springer, 1992.

[SFK00] Ravi S. Sandhu, David F. Ferraiolo, and D. Richard Kuhn. The NIST model for role-based access control: towards a unified standard. In *ACM Workshop on Role-Based Access Control*, pages 47–63. ACM, 2000.

[Sip96] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1996.

[SM73] Larry J. Stockmeyer and Albert R. Meyer. Word problems requiring exponential time: Preliminary report. In *Conference Record of the 5th Annual ACM Symposium on Theory of Computing (STOC '73)*, pages 1–9. ACM, 1973.

[SM04] Andreas Schaad and Jonathan Moffett. Separation, review and supervision controls in the context of a credit application process: a case study of organisational control principles. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC '04)*, pages 1380–1384. ACM, 2004.

[Sno87] Richard Snodgrass. The temporal query language TQuel. *ACM Transactions on Database Systems*, 12(2):247–298, 1987.

[Sno88] Richard Snodgrass. A relational approach to monitoring complex systems. *ACM Transactions on Computer Systems*, 6(2):157–195, 1988.

[Sox02] Sarbanes-Oxley Act of 2002, 2002. Public Law 107-204, 116 Stat. 745.

[SR03] Koushik Sen and Grigore Roşu. Generating optimal monitors for extended regular expressions. *Electronic Notes in Theoretical Computer Science*, 89(2):1–20, 2003.

[SSLK06] Oleg Sokolsky, Usa Sammapun, Insup Lee, and Jesung Kim. Run-time checking of dynamic properties. *Electronic Notes in Theoretical Computer Science*, 144(4):91–108, 2006.

[SSS+03] César Sánchez, Sriram Sankaranarayanan, Henny Sipma, Ting Zhang, David L. Dill, and Zohar Manna. Event correlation: Language and semantics. In *Proceedings of the 3rd International Conference on Embedded Software (EMSOFT '03)*, volume 2855 of *Lecture Notes in Computer Science*, pages 323–339. Springer, 2003.

[SSW05] Andreas Schaad, Pascal Spadone, and Helmut Weichsel. A case study of separation of duty properties in the context of the austrian "elaw" process. In *Proceedings of the 2005 ACM Symposium on Applied Computing (SAC '05)*, pages 1328–1332. ACM, 2005.

[Sta08] International Financial Reporting Standards. International accounting standards board, 2008.

[Sto07] Volker Stolz. Temporal assertions with parametrised propositions. In *Proceedings of the 7th International Workshop on Runtime Verification (RV '07)*, volume 4839 of *Lecture Notes in Computer Science*, pages 176–187. Springer, 2007.

[SVAR04] Koushik Sen, Abhay Vardhan, Gul Agha, and Grigore Roşu. Efficient decentralized monitoring of safety in distributed systems. In *Procedings of the 26th International Conference on Software Engineering (ICSE 2004)*, pages 418–427. IEEE Computer Society, 2004.

[SW95] A. Prasad Sistla and Ouri Wolfson. Temporal triggers in active databases. *IEEE Transactions on Knowledge and Data Engineering*, 7(3):471–486, 1995.

[SZ97] Richard Simon and Mary Ellen Zurko. Separation of duty in role-based environments. In *Proceedings of the 10th IEEE workshop on Computer Security Foundations (CSFW '97)*, page 183. IEEE Computer Society, 1997.

[Tho90] Wolfgang Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 133–192. MIT Press, 1990.

[TN96] David Toman and Damian Niwinski. First-order queries over temporal databases inexpressible in temporal logic. In *Proceedings of the 5th International Conference on Extending Database Technology (EDBT '96)*, pages 307–324. Springer, 1996.

[Tom03] David Toman. Logical data expiration. In Jan Chomicki, Ron van der Meyden, and Gunter Saake, editors, *Logics for Emerging Applications of Databases*, chapter 6, pages 203–238. Springer, 2003.

[Tom07] David Toman. On construction of holistic synopses under the duplicate semantics of streaming queries. In *Proceedings of the 14th International Symposium on Temporal Representation and Reasoning (TIME '07)*, pages 150–162. IEEE Computer Society, 2007.

[TOOD96] Debra J. Richardson T. Owen O'Malley and Laura K. Dillon. Efficient specification-based test oracles for critical systems. In *Proceedings of the 1996 California Software Symposium*, 1996.

[TR05] Prasanna Thati and Grigore Roşu. Monitoring algorithms for metric temporal logic specifications. *Electronic Notes in Theoretical Computer Science*, 113:145–162, 2005.

[Tru99] John K. Truss. *Discrete Mathematics for Computer Scientists*. International Computer Science Series. Addison-Wesley, 2nd edition, 1999.

[Ull88] Jeffery D. Ullman. *Priniciples of Database and Knowledge-Base Systems*. Computer Science Press, 1988.

[Var96] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Proceedings of the 8th Banff Higher Order Workshop on Logics for Concurrency : Structure versus Automata*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer, 1996.

[Vis00] Mahesh Viswanathan. *Foundations for the run-time analysis of software systems*. PhD thesis, University of Pennsylvania, 2000.

[vL06] Jan van Lunteren. High-performance pattern-matching for intrusion detection. In *Proceedings of the 25th IEEE International Conference on Computer Communications (INFOCOM '06)*. IEEE, 2006.

[VW86] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the Symposium on Logic in Computer Science (LICS '86)*. IEEE Computer Society, 1986.

[VW94] Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.

[Wel83] Peter D. Welch. The statistical analysis of simulation results. In Stephen S. Lavenberg, editor, *The Computer Performance Modeling Handbook*. Academic Press, 1983.

[Wol83] Pierre Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1/2):72–99, 1983.

[Yi90] Wang Yi. Real-time behaviour of asynchronous agents. In *Proceedings on Theories of Concurrency : Unification and Extension (CONCUR '90)*, volume 458 of *Lecture Notes in Computer Science*, pages 502–520. Springer, 1990.

[ZJM+08] Stan Zdonik, Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Mitch Cherniack, Ugur Cetintemel, and Richard Tibbetts. Towards a streaming SQL standard. In *Proceedings of the 34th International Conference on Very Large Databases (VLDB '34)*, pages 1379–1390. VLDB Endowment, 2008.

Curriculum Vitae

Personal Information

Name	Samuel Müller
Date of birth	12.12.1979
Place of birth	Winterthur, Switzerland
Nationality	Swiss

Education

August 2005 to May 2009	Doctoral studies at the Swiss Federal Institute of Technology (ETH) in Zurich, Switzerland
October 2000 to June 2006	Studies in economics at the University of Zurich, Switzerland (lic. oec. publ.)
October 2000 to June 2004	Studies in computer science at the University of Zurich, Switzerland (Dipl. Inform.)
October 2003 to January 2004	Exchange semester at the University of Lund, Sweden

Professional Experience

November 2004 to May 2009	Predoctoral researcher at the IBM Zurich Research Laboratory in Rüschlikon, Switzerland
October 2003 to April 2004	Master student at the IBM Germany Development Laboratory in Böblingen, Germany
July 2003 to October 2003	Systems engineer at Swiss Re in Zurich, Switzerland
February 2003 to June 2003	Teaching assistant at the University of Zurich, Switzerland
April 2000 to June 2003	Consultant at Beringer & Partner AG in Spreitenbach, Switzerland