UNIVERSITÄT
DES
SAARLANDES

Monitoring with Parameters

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

Peter Faymonville

Saarbrücken
2019

Dean of the faculty	Prof. Dr. Sebastian Hack
Date of colloquium	17.05.2019
Chair of the committee	Prof. Dr. Jan Reineke
Reviewers	Prof. Bernd Finkbeiner, PhD Prof. César Sánchez, PhD
Academic Assistant	Dr. Daniel Stan

Abstract

Runtime monitoring of embedded systems is a method to safeguard their reliable operation by detecting runtime failures within the system and recognizing unexpected environment behavior during system development and operation. Specification languages for runtime monitoring aim to provide the succinct, understandable, and formal specification of system and component properties.

This thesis explores runtime monitoring in the domain of autonomous unmanned aerial systems (UAS), which are a challenge for monitoring due to their dynamic physical environment, multitude of sensors with complex data and on-line decision-making capability. They are controllable via high-level control commands and have limited on-board computational resources with real-time constraints. Since the monitor itself is a critical system component, predictions on its use of resources, specifically the usage of memory, are a key requirement for their application.

There is a trade-off between expressivity and resource predictability: On one end of the spectrum, temporal logic-based specification languages satisfy the resource predictability requirement, since their automata-based monitoring implementations come with formal resource guarantees for execution time and memory. However, for UAS, these languages are not sufficiently expressive to handle the complex physical environment. On the other end, general purpose programming languages are sufficiently expressive, but lack resource predictability.

Starting from temporal logics, this thesis bridges the expressivity gap for monitoring specifications in three key aspects while maintaining the predictability of resource usage. First, we provide monitoring algorithms for linear-time temporal logic with parameters (PLTL), where the parameters bound the number of steps until an eventuality is satisfied. Second, we introduce Lola 2.0, which adds data parameterization to the stream specification language Lola. Data parameterization allows for quantification over a data domain. Third, we integrate real-time specifications in RTLola and add real-time sliding windows, which aggregate data over real-time intervals. For the combination of these extensions, we present a design-time specification analysis which provides resource guarantees.

We report on a case study on the application of the language in an autonomous

UAS. This case study has been carried out in collaboration with the German Aerospace Center (DLR Braunschweig). Component and system properties were specified together with domain experts in the developed stream specification language and evaluated in a real-time hardware-in-the-loop testbed with a complex environment simulation. The case study demonstrates that our language provides sufficient expressivity for practically relevant properties of autonomous vehicle systems while guaranteeing resource predictability.

Zusammenfassung

Laufzeitüberwachung von eingebetteten Systemen ist eine Methode zur Absicherung zuverlässigen Systemverhaltens. Sie erkennt zur Laufzeit sowohl Fehler innerhalb des Systems als auch unerwartetes Verhaltens der Systemumgebung während Systementwicklung und Systembetrieb. Spezifikationssprachen zur Laufzeitüberwachung ermöglichen die kompakte, verständliche und formale Spezifikation von Eigenschaften auf Komponenten- und Systemebene.

Die vorliegende Dissertation untersucht Laufzeitüberwachung in der Anwendungsdomäne von autonomen, unbemannten Flugsysteme (UAS), welche eine Vielzahl von Herausforderungen an die Überwachungsmethode stellt. Hierzu gehören die komplexe physische Umgebung, eine Vielfalt an Sensoren mit komplexen Datenströmen und die Möglichkeit, zur Laufzeit Kontrollentscheidungen zu treffen. Systeme dieser Art können mit abstrakten Befehlen auf hoher Ebene gesteuert werden, allerdings haben sie begrenzte Rechenressourcen an Bord, die Echtzeitanforderungen genügen muss. Da der Laufzeitmonitor in diesen Systemen selbst zur kritischen Systemkomponente wird, ist die Vorhersagbarkeit von Ressourcennutzung, spezifisch von Speichernutzung, eine Kernanforderung für die Anwendung.

Zwischen der Ausdrucksfähigkeit einer Spezifikationssprache und der Vorhersagbarkeit ihrer Ressourcennutzung besteht ein Zielkonflikt: An einem Ende des Spektrums gibt es Spezifikationssprachen, die auf Temporallogiken basieren. Diese erfüllen die Vorhersagbarkeitsanforderung, da ihre automatenbasierten Implementierungen formale Garantien zur Ressourcennutzung von Speicher und Laufzeit mitbringen. Leider sind sie nicht ausdrucksstark genug, um die Eigenschaften der komplexen physischen Umgebung von autonomen UAS zu beschreiben. Am anderen Ende des Spektrums stehen universelle Programmiersprachen, welche ausreichend ausdrucksstark sind, allerdings keine Vorhersagbarkeit ihrer Ressourcennutzung zulassen.

Beginnend von Temporallogiken beschreibt diese Dissertation einen Brückenschlag zwischen den beiden Enden des Spektrums. Sie erweitert die Ausdrucksfähigkeit der Spezifikationssprachen in drei Kernaspekten, während die Vorhersagbarkeit der Ressourcennutzung erhalten bleibt. Erstens werden Algorithmen zur

Überwachung mittels linear temporaler Logik mit Parametern (PLTL) beschrieben. Die Parameter repräsentieren hier die Einhaltung von Schrittgrenzen der modalen Operatoren. Zweitens werden Datenparameter in die strombasierte Spezifikationssprache Lola eingeführt. Die resultierende Sprache Lola 2.0 ermöglicht mittels der Datenparameter die Quantifizierung über Datendomänen. Drittens wird gezeigt, wie Realzeitanforderungen in RTLola integriert werden können. Mit gleitenden Fenstern über Realzeitintervalle können diese Anforderungen erfasst werden. Für die Kombination aus diesen Erweiterungen wird eine Spezifikationsanalysetechnik aufgezeigt, welche zum Entwicklungszeitpunkt der Spezifikationen Garantien über die Ressourcennutzung zulässt.

In einer praktischen Fallstudie wurde die Anwendung der entwickelten Spezifikationssprachen zur Überwachung unbemannter autonomer Flugsysteme in Kooperation mit dem deutschen Zentrum für Luft- und Raumfahrt (DLR) in Braunschweig erprobt. Spezifikationen auf Komponenten- und Systemebene wurden gemeinsam mit Domänenexperten entwickelt und auf einem echtzeitfähigen Prüfstand mit Umgebungssimulation erprobt. Diese Fallstudie zeigt, dass die Sprache ausreichend ausdrucksstark für praktisch relevante Spezifikationen von autonomen Fahrzeugen ist und gleichzeitig Vorhersagbarkeit von Ressourcennutzung zulässt.

Acknowledgements

This thesis would not have been possible without the support from several people in many different ways. First and foremost, this thesis would not exist without my supervisor, Bernd Finkbeiner, who shaped many of the research ideas and always had an open door. It was a decade-long journey from the sunny coast of San Diego with many unique experiences and encouragement.

Second, the excellent research environment of the Reactive Systems Group would not be the same without my colleagues, Lars Kuhtz, Rayna Dimitrova, Hans-Jörg Peter, Rüdiger Ehlers, Klaus Dräger, Michael Gerke, Andrey Kupriyanov, Markus Rabe, Hazem Torfah, Leander Tentrup, Felix Klein, Jesko Hecking-Harbusch, Martin Zimmermann, Alexander Weinert, Swen Jacobs, Mouhammad Sakr, Christopher Hahn, Norine Coenen, Noemi Passing, Maximilian Schwenger, Malte Schledjewski, Jana Hofmann, and Christa Schäfer, who were always open for inspirational discussions and collaboration over coffee and cake.

Third, the applied parts of this thesis would not have happened without the generous support from my collaborators at DLR Braunschweig. Sebastian Schirmer, Florian-Michael Adolf, Christoph Torens, Jörg Dittrich and colleagues gave the opportunity to test runtime monitoring in a real system and provided essential motivation and inspiration for this thesis.

Fourth, I would like to thank César Sánchez for joining the thesis committee and for performing an excellent and thorough review of this thesis.

Fifth, life in Saarbrücken outside of the university was greatly enriched by my friends and flatmates Anna, Maurice, Silke, Benni, Jana, Philip, Olli, Corinna, and the fellow volunteers from the volunteer fire department St. Johann.

Sixth, I am very grateful to my family, Rudolf, Irmgard, Jochen, Susanne, Christoph, and Claus for their unconditional support.

Finally and most importantly, I owe special thanks to my wife, Anna Marie, who provided support in crucial moments and had patience with my absentmindedness. Thank you for being part of the journey!

Contents

1 Introduction

Ensuring correctness of systems is an important goal in the development of computational systems. One answer to the problem of system correctness is design verification: ensuring error-freeness at design-time. Yet, existing methods for design verification are too expensive to apply to all systems: on the usability side the tools lack automation, the specification languages require significant training for users, the creation of complex specifications needs large manual effort, and many approaches are limited by their discrete abstractions. Also, even after design verification and testing, the operation of safety-critical systems in complex and dynamic environments may lead to unforeseen operating conditions, and additionally unpredictable hardware faults may lead to unsafe behavior.

To safeguard the system behavior at runtime, *runtime monitoring* is an established method to detect failures within the system and discharge environment assumptions of the specification by monitoring the system during its operation.

A shared challenge of the aforementioned verification approaches is the concrete definition of correctness for a system – we call a system correct if it adheres to its specification. One of the main challenges for specification is expressivity, as system properties are more easily specified in high-level languages, but the computational costs of such languages may be prohibitive.

The field of *runtime verification* [51, 48] refers to the systematic study of verification techniques to evaluate traces for verdicts based on a specification, originating in the larger field of *formal methods* and therefore heavily influenced by the development of formal specification languages and accompanying algorithms.

Monitoring as a Formal Method In the larger context of formal methods, runtime verification is a so-called *lightweight* formal method, due to its applicability to black-box systems, the expressiveness of the specification mechanisms, and the comparatively low computational complexity – it only observes a single execution. It is not a prerequisite to have a formalization or an abstraction of the system design and internals in the form of a model, before runtime verification can be applied. The creation and maintenance of system models of complex systems suitable for formal verification is a significant effort within an engineering process. The computational advantage stems from the fact that runtime verification only has to consider a single system execution, while static verification methods such as model checking reason over all possible system executions. Therefore, more expressive specification mechanisms can be used as concrete data values are provided along the execution, without running into memory resource limits. However, static and dynamic verification methods can also be combined for a system: if for example model checking has been performed on a discrete abstraction of a system, runtime verification can complement the model checking phase by verifying the inherent assumptions of the abstraction at runtime. Two main alternatives of monitoring are differentiated: *Offline* monitoring refers to the method where the full trace is already available on a storage medium. In this case, the trace can be processed in forward or reverse direction, random positions can be accessed, and the length of the trace is known a-priori. *Online* monitoring is the method where the length of trace is not known beforehand, and the trace becomes available to the monitor one event at a time. The monitor may run concurrently to the system under observation, and the trace is processed in a forward manner.

Autonomous Unmanned Aerial Systems One of the main application areas of monitoring and runtime verification are reactive embedded systems (also referred to as cyber physical systems [49], if they are networked). The distinctive feature of this class of systems is the interaction between a physical environment and the digital control system. These systems consist of at least one sensor, a processing unit, and an actuator. As many cyber physical systems are used in safety-critical applications, a number of norms and standards exist for their development, for example the IEC 61508 [64] family of process norms, which are instantiated for various industries such as automotive [44]; or similar norms for aviation [58, 59]. These are enforced either through product liability laws for manufacturers or due to regulatory processes.

As a challenging application benchmark for runtime monitoring, we present a representative class of systems in Chapter 6: autonomous, unmanned aerial systems (UAS). For UAS, the standardization work for the certification and safety of these system is currently ongoing [46]. The challenges for monitoring specifications for these systems are (1) the dynamic, physical environment, (2) the dynamic high-level control commands from the ground station, and (3) the limited computational resources on-board with real-time constraints. Since no on-board pilot is present and the communication channels to the ground station are unreliable, the system needs to take autonomous decisions and may need to switch to a safe state on a monitor decision. Since this establishes the runtime monitor as a critical system component, the integrity of the monitor itself is of particular importance. The predictability of the usage of memory resources by the monitor is a key requirement for this application.

On the other hand, to meet the requirements for runtime monitoring of this benchmark application, the specification language has to be sufficiently expressive to cover properties of the physical environment and complex sensor data, and at the same time provide resource guarantees to enable the realization of the runtime monitor with sufficient integrity.

Expressivity in Runtime Monitoring The design spectrum concerning the expressivity of specification languages in runtime monitoring ranges from restrictive temporal logic-based approaches to full programming languages. In increasing order of expressivity, the following main classes of specification languages have been used for runtime monitoring: Variants of linear-time temporal logics like LTL [39, 17, 13] have been adapted from verification applications to monitoring applications. Further, extensions of classic LTL towards more expressive temporal logics such as metric interval temporal logic (MITL) have been introduced in [55]. In the context of real-time signals, runtime monitoring for signal-temporal logic (STL) was discussed in [27, 28]. Moreover, related formalisms based on state machines, regular expressions, and automata have been applied for monitoring, such as quantitative event automata (QEA) [60] and quantitative regular expressions (QRE) [4]. Beyond these, stream-based approaches such as Lola [24], TeSSLa [23], and Striver [40] have been adapted from synchronous programming languages for monitoring. Seperately, rule-based approaches such as Eagle and RuleR [6, 7] have been introduced. Finally, monitoring and stream processing have been performed with full programming languages, such as in Apache Spark [76].

There exists an essential trade-off between the expressivity of the specification language and the resource predictability. For the benchmark application of autonomous UAS, the requirements towards the expressivity of the specification language and the resource guarantees could not be met by any of the aforementioned approaches. Since autonomous UAS fly in a dynamic, physical environment with continuous observations of real-valued signals, classic logic- or automata-based formalisms are not sufficient to express the necessary properties (for more details, see Chapter 6). This is also the case for STL-based monitoring, since it cannot express properties of online statistics of the incoming sensor data due to the Boolean abstraction. More expressive formalisms, such as rule-based approaches and full programming languages are not able to provide the necessary resource guarantees in the monitor implementation. The existing stream-based approaches with their equation-style specifications present the closest match to the requirements of the application from a usability standpoint, but still are not equipped to handle complex sensor data and dynamic lists of high-level commands in the form of waypoints for the aircraft or able to fully express the time-bounded reachability property for these waypoints.

Therefore, in this thesis, we demonstrate how to extend the expressivity while maintaining the resource predictability. The initial step here is the integration of different types of parameterization in Chapters 2 and 4.

Parameterization in Runtime Monitoring One step towards more expressive specification languages for monitoring is to incorporate the concept of *parameterization*, which can be added to both logic- and stream-based formalisms. A number of earlier approaches have been proposed to integrate different forms of parameterization into formal specification languages for runtime monitoring. In the context of monitoring Java programs, LTL with parametrised propositions was proposed in [69]. This work was extended to next-free LTL with free variables and binding in [68]. For the parametric monitoring of Java objects, the JavaMOP framework was used to monitor object instances in [45]. In a continuation, parametric monitoring with slicing on execution traces was introduced in [62]. More expressive variants of logic for runtime monitoring where considered in [14, 8], where first-order LTL was used, and in [15], where first-order logic for policies was introduced. Rule-based specifications with data parameters were described in [7], where a parameterized version of RuleR was introduced.

The described approaches fall short of our application requirements, as the logic-based approaches are not sufficiently expressive and usable, and the monitor implementations for parametric rule-based specifications do not provide resource guarantees. As we will show, the expressivity gain through parameters brings us one step closer to a realistically usable and sufficiently expressive stream-based monitor specification language. This language and especially its corresponding monitor implementation is able to provide resource guarantees for our benchmark application.

Handling real-time behavior A second necessary addition for expressivity is the handling of real-time behavior, where incoming events are not necessarily arriving in a synchronized manner and incoming signals are real-valued. While this problem alone could be solved with a straightforward semantic extension of the existing specification mechanism, resource predictions require a careful balance between immediate reaction on asynchronous events and deterministic timing for the monitor outputs. Earlier work on real-time behavior in runtime monitoring includes: Real-time variants of temporal logics [47, 54, 27], such as STL over real-valued signals [28]. Metric variants of first-order properties [10] with sliding windows [9], and event-rate independence in the tool Aerial [12]. Real-time capable monitoring in Copilot [56], TeSSLa [50, 23], and asynchronous stream monitoring in Striver [40].

Contributions This thesis provides the following contributions to the field of runtime verification. To enhance the expressivity of specification languages for monitoring, we introduce three extensions to increase specification language expressivity and improve their applicability. The first extension concerns parametric linear temporal logic (PLTL) for runtime monitoring, which had been previously used for model checking and adds parameters for satisfying eventualities. The second extension, Lola 2.0, adds data parameterization to the stream-based specification language Lola, providing the ability to quantify over a data domain. The third extension introduces real-time specifications via real-time sliding windows, which are able to express aggregations on data over time intervals, in RTLola. All these include the identification of fragments with advantageous computational properties. An important property of the monitoring for these specification languages is their resource predictability, which can either be guaranteed from the semantics or derived for the individual specification. Ambiguity is identified as an important property, which refers to the ability of events in the (distant) future to affect the current verdict, parameters, and measures of monitor implementations. In addition to the improvements in expressiveness, we have performed a case study on the application to an autonomous unmanned aerial systems (UAS). This evaluates the applicability of our specification languages for a specific system design, and allows to demonstrate the contribution of runtime monitoring to the system design and development of safety-critical systems. For autonomous UAS, we show that runtime monitoring with the extended specification languages that we propese is suitable to establish the safe operation of the UAS and safeguard the flight environment.

More concretely, this thesis provides the following technical contributions:

1. We establish a lower bound for the space complexity of the online monitoring problem for PLTL.
2. We introduce deterministic PLTL and unambiguous PLTL as syntactic and semantic fragments, respectively.
3. We provide online monitoring algorithms for the two fragments with logarithmic space complexity in the length of the trace.
4. We give an experimental evaluation on circuit examples for a prototypical Java implementation of the monitoring algorithm.
5. We introduce Lola 2.0, which adds data parameterization to the stream-based specification language Lola.
6. We adapt the notion of efficient monitorability to Lola 2.0.
7. We provides an online monitoring algorithm for efficiently monitorable Lola 2.0.
8. We evaluate the enhanced expressivity by expressing network monitoring properties for security violations in Lola 2.0 and provide an experimental evaluation of the monitoring algorithm on network traces.
9. We introduce RTLola by adding real-time sliding windows to Lola 2.0.
10. We define a static analysis for the memory usage of RTLola specifications.
11. We provide an online monitoring algorithm for RTLola.
12. We evaluate an implementation of the online monitoring algorithm for data analysis tasks.
13. We perform a case study in the application environment of autonomous unmanned aircraft systems.
14. We introduce a domain-specific variant of Lola for the UAS application.
15. We provide representative practical specifications in Lola and RTLola for the functional safety of the aircraft.
16. We evaluate the online monitoring of the specifications with experiments on a hardware-in-the-loop (HIL) testbed on real flight hardware.

Structure

- In Chapter 2, we study the monitoring problem for parametric linear temporal logic, which extends linear temporal logic with parametric step bounds on the temporal operators. While the full logic has a larger than logarithmic space complexity in the length of the trace, we introduce deterministic PLTL and unambiguous PLTL as syntactic and semantic fragments of the logic. Both fragments have only logarithmic space complexity in the length of the trace and enjoy the unique measure property, where monitoring a PLTL specification on a single trace always yields a unique result for the timing parameters.
- Chapter 3 provides the necessary background for the stream-based monitoring paradigm by recapitulating the monitoring language Lola together with its monitoring algorithm and analysis. In this chapter we describe basic properties and fragments of the specification language.
- Chapter 4 introduces data parameterization to the synchronous stream-based specification language Lola. Here, we introduce data parameters to output streams, which allows us to slice the incoming input data and treat individual slices on their own timeline and control the lifetime of the individual slices. This extension enables the application of the resulting language, called Lola 2.0, for network monitoring to detect violations of security policies via declarative, stream-based specifications. For practical applications, the efficiently-monitorable fragment of Lola 2.0 is defined to be able to ensure bounded memory usage of the monitor.
- Next, in Chapter 5 we extend the previously synchronous timing model of Lola to include references to real-time intervals, we call the resulting language RTLola. In RTLola, the monitor realization is split into an input-synchronous, event-triggered part and a fixed-rate part, which runs independent of the input event stream. The monitor outputs are evaluated at this specifiable monitor frequency. To recover a bounded memory guarantee despite a possibly unlimited input event rate, *efficient aggregation functions* are defined to access input streams via real-time sliding windows. In general, memory guarantees cannot be established, we introduce a static analysis of RTLola specifications to determine these memory requirements.
- Finally, in Chapter 6 we demonstrate the applicability of stream-based runtime monitoring in a real-world case study: unmanned aerial systems with

complex autonomous decision making. In cooperation with the Institute of Flight Systems at German Aerospace Center (DLR), we have introduced online and offline monitoring to improve system design and development within the DLR ARTIS fleet of unmanned aircraft. We report about the experiences during specification elicitation, instrumentation, and offline and online experiments, as well as their impact on further specification mechanism design. Further, we characterize the potential impact of runtime monitoring for this regulated domain.

The presented contributions in this thesis have been published in the following papers:

1. Chapter 2: Peter Faymonville, Bernd Finkbeiner, and Doron Peled.
   Monitoring Parametric Temporal Logic.
   15th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2014). [33]

2. Chapter 4: Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Hazem Torfah.
   A Stream-based Specification Language for Network Monitoring.
   16th International Conference on Runtime Verification (RV 2016). [34]

3. Chapter 5: Peter Faymonville, Bernd Finkbeiner, Maximilian Schwenger, and Hazem Torfah.
   Real-time Stream-based Monitoring.
   arXiv:1711.03829. [35]

4. Chapter 6: Florian-Michael Adolf, Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Christoph Torens.
   Stream Runtime Monitoring on UAS.
   17th International Conference on Runtime Verification (RV 2017). [1]

5. Chapter 6: Christoph Torens, Florian-Michael Adolf, Peter Faymonville, and Sebastian Schirmer.
   Towards Intelligent System Health Management using Runtime Monitoring.
   AIAA Information Systems-AIAA Infotech @ Aerospace, AIAA SciTech Forum, (AIAA 2017-0419). [74]

2 Monitoring Parametric Temporal Logics

Temporal logics such as LTL are popular as a specification mechanism in runtime monitoring in the runtime verification community following their success in hardware model checking. An important property of LTL-based monitoring is *trace length independence*, which establishes that any LTL formula can be monitored online on a trace with constant memory in the length of the trace.

In this chapter, we describe the monitoring problem for an extension of LTL equipped with step-bounded parametric temporal operators. This extension gives additional expressibility and enables us to measure performance properties of the system, such as response times to external events. Although LTL is a rather restrictive specification mechanism for monitoring, the issues described in this chapter foreshadow the challenges of Chapter 4, which deals with a much more expressive stream-based specification language. The content of this chapter is based on [33] and covers online and offline monitoring algorithms as well as a complexity analysis with a lower bound proof on the memory requirements in terms of the trace length of any online algorithm for the full logic.

Due to the importance of space requirements for monitoring algorithms, especially in hardware realizations, we then introduce two restricted fragments of PLTL, which both have the property of being able to monitor a trace with logarithmic memory in the length of the trace.

Parametric LTL (PLTL) was originally introduced in [2, 3] as a logic for model measuring, i.e. to establish quantitative guarantees for system models given as Kripke structures. The model measuring problem is defined as determining for a Kripke structure the parameter valuations for which the formula holds on all paths.

2.1 PLTL Syntax and Semantics

Given a set of atomic propositions *AP* with typical element *a*, the syntax of LTL is given by the following grammar:

$$\varphi_{\mathsf{LTL}} ::= a \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \bigcirc\varphi \mid \Diamond\varphi \mid \Box\varphi \mid \varphi\,\mathcal{U}\,\varphi \mid \varphi\,\mathcal{R}\,\varphi$$

For a set of parameters X with typical element x, PLTL extends LTL with the following parametric operators:

$$\varphi ::= \varphi_{\mathsf{LTL}} \mid \Diamond_{\leq x} \varphi \mid \Box_{\leq x} \varphi$$

A valuation α for the set of parameters X is a function $\alpha : X \mapsto \mathbb{N} \cup \{\infty\}$, mapping each parameter x to a natural number or infinity.

As an example, the PLTL formula $\Diamond_{\leq x} a$ expresses that the atomic proposition a has to hold within $\alpha(x)$ steps.

An arbiter example, where access requests (r_1, r_2) should be eventually granted (g_1, g_2) within x,y steps, but not at the same time, can be specified in PLTL as following:

$$\varphi = \Box(r_1 \to \Diamond_{\leq x} g_1) \land \Box(r_2 \to \Diamond_{\leq y} g_2) \land \Box\neg(g_1 \land g_2).$$

The PLTL semantics [2, 3] were originally defined on infinite traces. Monitoring algorithms always deal with finite portions of an potentially infinite trace, since monitors either read a finite trace from storage in case of offline monitoring, or a finite history of an ongoing run of a system in the case of online monitoring. The semantics of LTL are usually defined for infinite traces, and therefore need to be adapted to the case of finite traces. There exists a large body of work covering the different finite trace semantics for LTL, an overview is given in [16]. Here, we follow the truncated path semantics, originally introduced in [29]. These semantics take a positive view on the system, treating unfulfilled obligations at the end position not as negative evidence towards the satisfaction of the formula on a trace, as long as they can still be satisfied.

A finite trace σ of length $|\sigma|$ provides an interpretation of the events $\Sigma = 2^{AP}$, where an atomic proposition is present in the event if the proposition currently holds, and the trace can thus be defined as the mapping $\sigma : \{0, \dots, |\sigma| - 1\} \mapsto \Sigma$. We denote the event at position k by $\sigma(k)$.

For a PLTL formula φ over the set of atomic propositions AP, a corresponding valuation α, a finite trace σ, and a trace position k, we define the following semantics for PLTL recursively over the structure of the formula:

- $(\sigma, k, \alpha) \models a$, iff $a \in \sigma[k]$.
- $(\sigma, k, \alpha) \models \neg\varphi$, iff $(\sigma, k, \alpha) \not\models \varphi$.
- $(\sigma, k, \alpha) \models \varphi_1 \land \varphi_2$, iff $(\sigma, k, \alpha) \models \varphi_1$ and $(\sigma, k, \alpha) \models \varphi_2$.
- $(\sigma, k, \alpha) \models \varphi_1 \lor \varphi_2$, iff $(\sigma, k, \alpha) \models \varphi_1$ or $(\sigma, k, \alpha) \models \varphi_2$.

- $(\sigma, k, \alpha) \vDash \bigcirc\varphi$, iff $k+1 < |\sigma|$ and $(\sigma, k+1, \alpha) \vDash \varphi$.
- $(\sigma, k, \alpha) \vDash \Diamond\varphi$, iff $\exists i.\, k \le i < |\sigma|$, where $(\sigma, i, \alpha) \vDash \varphi$.
- $(\sigma, k, \alpha) \vDash \Box\varphi$, iff $\forall i.\, k \le i < |\sigma|$ we have $(\sigma, i, \alpha) \vDash \varphi$.
- $(\sigma, k, \alpha) \vDash \varphi_1 \mathcal{U} \varphi_2$, iff $\exists i.\, k \le i < |\sigma|$, where $(\sigma, i, \alpha) \vDash \varphi_2$ and $\forall j.\, k \le j < i$, we have $(\sigma, j, \alpha) \vDash \varphi_1$.
- $(\sigma, k, \alpha) \vDash \varphi_1 \mathcal{R} \varphi_2$, iff either
  $\exists i.\, k \le i < |\sigma|$, where $(\sigma, i, \alpha) \vDash \varphi_1$ and $\forall j.\, k \le j \le i$, we have $(\sigma, j, \alpha) \vDash \varphi_2$,
  or $\forall j.\, k \le j < |\sigma|$ we have $(\sigma, j, \alpha) \vDash \varphi_2$.
- $(\sigma, k, \alpha) \vDash \Diamond_{\le x}\varphi$, iff $\exists i.\, k \le i \le \alpha(x)$ and $k + i < |\sigma|$, where $(\sigma, k+i, \alpha) \vDash \varphi$, or we have that $k + \alpha(x) \ge |\sigma|$.
- $(\sigma, k, \alpha) \vDash \Box_{\le x}\varphi$, iff $k + \alpha(x) < |\sigma|$ and $\forall i.\, k \le i \le \alpha(x)$ we have $(\sigma, k+i, \alpha) \vDash \varphi$.

Logical constants can be defined as following: **true** $:= a \vee \neg a$, **false** $:= a \wedge \neg a$.

Arbiter Trace Example As an example, consider the following finite trace over the alphabet $AP = \{r_1, r_2, g_1, g_2\}$:

position	0	1	2	3	4	5	6	7	8
stream	$\{r_1, r_2\}$	$\{r_1, r_2\}$	$\{g_1, r_2\}$	$\{r_1, g_2\}$	$\{r_1\}$	$\{r_1\}$	$\{r_1\}$	$\{r_1\}$	$\{g_1\}$

According to our previously defined semantics, the formula $\varphi = \Box(r_1 \rightarrow \Diamond_{\le x} g_1) \wedge \Box(r_2 \rightarrow \Diamond_{\le y} g_2) \wedge \Box\neg(g_1 \wedge g_2)$ is satisfied on this trace for $\alpha(x) = 5$ and $\alpha(y) = 3$.

Monotonicity Following the semantics of the parametric operators, we observe that parametric operators appear in two ways, which we call polarities: Parameterized eventualities, such as $\Diamond_{\le x}\varphi$, are *upward-closed*, i.e. if $\Diamond_{\le x}\varphi$ holds for some $\alpha(x) = j$, then it also holds for all $\alpha(x) = k$ with $k > j$. Conversely, parameterized always-operators are *downward-closed*: if $\Box_{\le x}\varphi$ holds for some $\alpha(x) = j$, then it also holds for all $\alpha(x) = k$ where $k < j$, i.e. for all smaller values.

Existence of a valuation α To check whether there exists any valuation α for a given trace σ, we can check a single valuation $\hat{\alpha}$ by setting the parametric operators to their weakest requirements: for $\Box_{\le x}\varphi$ we set $\hat{\alpha}(x) = 0$ and for $\Diamond_{\le y}\varphi$ to $\hat{\alpha}(y) = |\sigma|$.

LTL Abstraction The LTL abstraction $[\varphi]$ of a PLTL formula φ is the LTL formula which is satisfied if and only if there exists some parameter valuation under which φ is satisfied. It is obtained via the following syntactic rewrite rules:

- $[\Diamond_{\leq x} \psi] = \Diamond[\psi]$
- $[\Box_{\leq x} \psi] = [\psi]$
- $[p] = p$
- $[\neg p] = \neg p$
- $[\psi_1 \wedge \psi_2] = [\psi_1] \wedge [\psi_2]$
- $[\psi_1 \vee \psi_2] = [\psi_1] \vee [\psi_2]$
- $[\bigcirc \psi] = \bigcirc[\psi]$
- $[\Diamond \psi] = \Diamond[\psi]$
- $[\Box \psi] = \Box[\psi]$
- $[\psi_1 \mathcal{U} \psi_2] = [\psi_1]\mathcal{U}[\psi_2]$
- $[\psi_1 \mathcal{R} \psi_2] = [\psi_1]\mathcal{R}[\psi_2]$

Optimality Assuming a formula with only a single parameter, occuring once in the formula, the monotonicity properties of the operators directly give rise to a definition of optimality: For upward-closed parametric operators, we report the *minimal* value which can be guaranteed on the trace, since it represents the strongest guarantee which holds. For downward-closed parametric operators, we conversely report the *maximal* value. If the specification contains more than a single parameter, these optimization goals do not necessarily yield a single, optimal solution: Consider the trace $\sigma = [\emptyset, \{p\}]$ of length 2. For the PLTL-formula $\varphi = \Diamond_{\leq x} \Diamond_{\leq y} p$, there are two minimal incomparable valuations: $\alpha : x \mapsto 1, y \mapsto 0$ and $\alpha' : x \mapsto 0, y \mapsto 1$, depending on when the descent to the satisfaction of the subformula $\Diamond_{\leq y} p$ happens. The set $\{\alpha, \alpha'\}$ forms an antichain for the ground set $\mathbb{N}^2$ and the natural partial order $\leq$ on integer vectors.

To restore a total (i.e. linear) order, we introduce a total order on the parameters in X, called *priority order* and denoted as $\gg$, where we say that $x \gg y$ if the parameter x has priority over y. The $max(X)$ w.r.t. $\gg$ denotes the highest priority parameter. This order in turn induces a total order on valuations, denoted as $\sqsupset$, where $\alpha \sqsupset \alpha'$ if, for $x = max(X)$, we have:

- $\alpha(x) < \alpha'(x)$ and x is associated to an upward-closed parametric operator, or
- $\alpha(x) > \alpha'(x)$ and x is associated to a downward-closed parametric operator, or
- $\alpha(x) = \alpha'(x)$, and $\alpha \setminus \{x\} \sqsupset \alpha' \setminus \{x\}$.

For a PLTL formula φ and a trace σ, we call the optimal valuation α^* – with respect to $\sqsupset$ – the *measure* of φ on σ.

Operator context Observe that parametric operators in the scope of always, eventually, until, and release operators measure more than once and combine their measurements due to the conjunctive or disjunctive nature of their parent operator: As an example, the formula $\Box\Diamond_{\leq x}\, p$ triggers a measurement in every position of the trace, and the measure of the formula is the conjunctive combination of the individual measurements of $\Diamond_{\leq x}\, p$, which themselves are downward closed. Therefore, the measure will be the maximum over all measurements due to the conjunctive nature of the $\Box$-operator, and the value will be the longest $\neg p$-sequence on the trace. If we consider the formula $\Diamond\Diamond_{\leq x}\, p$, however, the disjunctive nature of the $\Diamond$-operator allows us to select the minimum over the measurements starting in every position of the trace, yielding – if the formula is satisfied at all on a trace – always the value 0, since we can shift the start of the measurement to the first p-position. While formulas of this type are syntactically permissible in full PLTL, they are obviously not very meaningful in practice.

Finite trace semantics and measurements Because of the truncated path semantics, we ignore open measurements which look beyond the end of the trace. A formula like $\Box\Diamond_{\leq x}\, p$ would only hold on traces which end in p-states and not report a measure otherwise. The measure for x will take into account all evidence observed on the trace, but the last open eventuality will only be taken into account insofar it contributes to a new, larger minimum value for x.

Negation-normal form To simplify the presentation of our algorithms, we assume that PLTL formulas are given in negation-normal form, where negations occur only next to atomic propositions. As in LTL, general PLTL formulas can be rewritten to negation-normal form with a small set of rewrite rules, which push negations to the inner subformulas and remove double negations. For PLTL, we add the rewrite rule $\neg\Box_{\leq x}\, \varphi \mapsto \Diamond_{\leq x}\, \neg\varphi$ and $\neg\Diamond_{\leq x}\, \varphi \mapsto \Box_{\leq x}\, \neg\varphi$. This transformation increases the size of the formula at most linearly.

2.2 Offline monitoring

First, we give an algorithm to perform offline monitoring and measuring of a PLTL formula on a finite trace. The algorithm is best suited for the offline case, where the trace is stored on external storage before analysis, since it needs access to the trace positions in reverse chronological order. To apply the same algorithm in an online setting, one would need to store the trace and re-run the algorithm for every incoming event, which will become infeasible quickly. The algorithm is a straightforward extension of the one for LTL with statistical measures introduced in [37].

We first give a method to efficiently check a trace against a formula and a concrete valuation (given as an input to the algorithm), which is in effect a formula without parameters. Then, we present how to check a formula with a single parameter. Due to the given parameter order $\sqsupset$, we can then use this procedure to start from the highest-priority parameter, use binary search to determine an optimal value for the parameter while setting all other parameters to their weakest values. This optimal value can then be added as a constant to the valuation, and we can repeat the procedure for to the next-highest priority parameter. The final result will be the measure of the formula.

Checking formulas with constant parameters To check a formula with only constant parameters on a finite trace, we perform a backward traversal of the trace σ. The backward traversal has the usual advantage that the algorithm already knows how the future plays out, and thus does not have to track all possible nondeterministic choices, as a forward traversal of a trace must. This approach allows an algorithm whose runtime is linear in the length of the formula and in constant time per event.

We check the satisfaction of the formula φ on the trace by maintaining – for every subformula ψ of φ – a Boolean variable b_ψ, which indicates whether a subformula holds in the currently considered trace position. To compute b_ψ, we need to access the value of b_ψ in the previously processed position, which we denote by b'_ψ, to implement the expansion laws of the temporal operators. For every parametric subformula ψ, we will additionally maintain a counter c_ψ, which we will use to ensure that the constant parameters are correct for the given trace. In the case of a parametric eventuality $\Diamond_{\leq j} \psi_g$, this counter will track the distance to the satisfaction of the goal subformula ψ_g. In the case of a parametric always $\Box_{\leq j} \psi_g$, this counter keeps track of for how long ψ_g holds. In both cases, the corresponding Boolean variable b_ψ will be true as long as the counter c_ψ does not increase/decrease beyond k.

We initialize the b_ψ-variables as following:

- $b_{\bigcirc \psi}$:= **false**
- $b_{\Box \psi}$:= **true**
- $b_{\Diamond \psi}$:= **false**
- $b_{\psi_1 \mathcal{U} \psi_2}$:= **false**
- $b_{\psi_1 \mathcal{R} \psi_2}$:= **true**
- $b_{\Diamond_{\leq k} \psi}$:= **false**
- $b_{\Box_{\leq k} \psi}$:= **true**

All counters c_ψ are initialized to 0.

To process the event $\sigma[k]$, we first rename all variables b_ψ to b'_ψ and the apply the following update rules in a bottom-up fashion:

- $b_a :=$ *if* $a \in \sigma[k]$ *then* **true** *else* **false**
- $b_{\neg\varphi} := \neg b_\varphi$
- $b_{\varphi_1 \wedge \varphi_2} := b_{\varphi_1} \wedge b_{\varphi_2}$
- $b_{\varphi_1 \vee \varphi_2} := b_{\varphi_1} \vee b_{\varphi_2}$
- $b_{\bigcirc\psi} := b'_\psi$
- $b_{\square\psi} := \psi \wedge b'_{\square\psi}$
- $b_{\diamondsuit\psi} := \psi \vee b'_{\diamondsuit\psi}$
- $b_{\psi_1 \mathcal{U} \psi_2} := b_{\varphi_2} \vee (b_{\varphi_1} \wedge b'_{\psi_1 \mathcal{U} \psi_2})$
- $b_{\psi_1 \mathcal{R} \psi_2} := b_{\varphi_2} \wedge (b_{\varphi_1} \vee b'_{\psi_1 \mathcal{R} \psi_2})$

The counters c_ψ are updated in the following way:

- $c_{\diamondsuit_{\leq j}\psi_g} :=$ *if* b_{ψ_g} *then* 0 *else* $c'_{\diamondsuit_{\leq j}\psi_g} + 1$
- $c_{\square_{\leq j}\psi_g} :=$ *if* $\neg b_{\psi_g}$ *then* 0 *else* $c'_{\square_{\leq j}\psi_g} + 1$

Finally, the Boolean variables for the parametric subformulas are determined:

- $b_{\diamondsuit_{\leq j}\psi_g} := c_{\diamondsuit_{\leq j}\psi_g} \leq j$
- $b_{\square_{\leq j}\psi_g} := c_{\square_{\leq j}\psi_g} \geq j$

For every subformula and trace position, we thus update one Boolean variable and at most one integer counter. The running time of our algorithm is therefore linear in trace and specification, in $O(|\varphi| \times |\sigma|)$, and it runs in space $O(|\varphi| + \#_c \times \log|\sigma|)$ due to the integer counters, which are bounded by the trace length and encoded in binary. Every position of the trace has to be read only once from external storage. Moreover, accesses are in a predictable orer.

Measuring formulas with at least one parameter If the formula φ contains exactly one parameter, we perform binary search to obtain the optimal value for the parameter, which is bounded between 0 and $|\sigma|$. For a single parameter, this yields a runtime of $O(|\varphi| \times |\sigma| \times \log|\sigma|)$, as we need a logarithmic number of calls in the length of the trace.

For a larger number of parameters, we start with the highest-priority parameter according to $\sqsupset$ and set all other parameters to their weakest values, since they are of lower priority and the weakest values do not affect the determined measure. Once an optimal value for this parameter has been determined, we fix this value in the valuation and continue the procedure with the next parameter in the parameter order.

To determine the measure of a formula containing n parameters (bounded above by $|\varphi|$), we thus apply the procedure for the single parameter at most $|\varphi|$ times, leading us to the following theorem for offline monitoring:

Theorem 2.1. *Let φ be a PLTL formula, $\sqsupset$ be an order on the parameters of φ, and σ be a finite trace. With direct access to all trace positions, the measure of φ on σ can be computed in space $O(|\varphi| \times \log|\sigma|)$ and time $O(|\varphi|^2 \times |\sigma| \times \log|\sigma|)$.*

2.3 The online monitoring problem - hardness

After we have established an algorithm for the case of offline monitoring, we now turn our focus to the setting of online monitoring. In many applications of runtime verification, monitors run alongside their systems to analyse their behavior on-the-fly, or traces stored on secondary storage are too large to keep in memory or to be accessed multiple times during each analysis, since the monitoring process may be I/O-bound. Therefore, online algorithms and their space requirements are of great importance for monitoring tasks.

For temporal logics such as LTL, online monitoring with only a single access to each event can be performed by constructing an alternating automaton from the formula and maintaining an – in the worst case – exponential number of nondeterministic possibilities to satisfy the formula. The space complexity of this algorithm is constant in terms of the length of trace [37]. In turn, this enables to construct a hardware circuit without any external memory requirement [36]. The size only depends on the specification, which is usually known before a hardware monitor implementation. In terms of trace length, the work to be performed by monitor is constant per event, and does not grow, making timing fully deterministic.

In the case of PLTL, retaining the same space complexity for online monitoring is unrealistic, as for any logic which can measure the length of the trace. It is easy to measure the trace length since the formula $\bigcirc\,\mathbf{false}$ is only true in the last position of the trace. Therefore, we will have at least a logarithmic dependency due to the binary encoding of counters.

For most embedded systems, a logarithmic space dependency is still fine, since the event frequency of the system will usually be fixed at design time and the memory of the system can be appropriately sized.

However, as we will shortly prove, not even a logarithmic-space online monitoring algorithm exists for full PLTL. In later sections, we will give two restricted variants of the logic, which recover this important property.

Theorem 2.2. *There exists no online measuring algorithm for PLTL which uses only logarithmic memory in the length of trace.*

Proof. Suppose such an algorithm exists. We show that even for a formula with only two parameters, any online algorithm is forced to use more than logarithmic space in the length of trace. More concretely, let $\varphi = \Box(a \rightarrow (\Diamond_{\leq x} b \vee \Diamond_{\leq y} c))$ with $x \sqsupset y$. Now, we show that there exists a sequence σ of length $O(n \cdot m)$, which forces the memory of any monitor to store an arbitrary integer set $K = \{k_1, k_2, \dots, k_n\}$ of length n. Without loss of generality, we assume that the individual k_i are smaller than m and that the k_i are in ascending order, i.e. $k_1 < k_2 < \cdots < k_n$. To show the memory content requirement, we construct a continuation trace ρ, which is able to recall any of the k_i as part of the measure of φ of the concatenated trace $\sigma \cdot \rho$. The traces are illustrated in Figure 2.1.

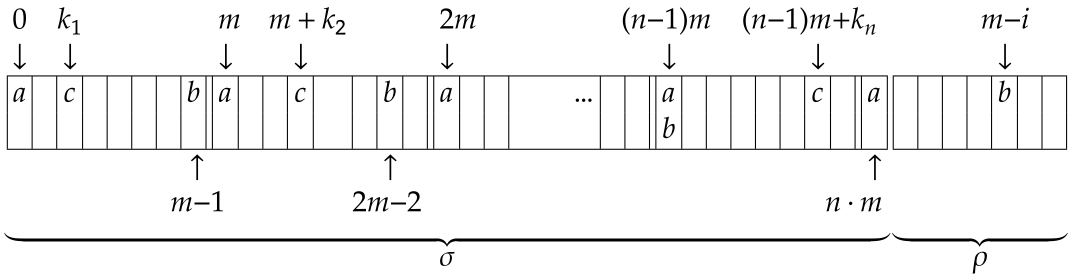


Figure 2.1: Sequences σ and ρ in the proof of Theorem 2.2. Proposition a is **true** every m steps, creating n measuring segments of length m. Proposition b starts at position $m-1$ and moves one step backward in every segment. Proposition c is used to encode the k_i, one per segment. The suffix ρ contains a single proposition b at the position $m-i$ to force the recall of value k_i as part of the measure of $\sigma \cdot \rho$.

The sequence σ is constructed as following: Every segment of length m starts with a position on which proposition a holds. Proposition b occurs first at position $m-1$ and moves one position closer to the beginning of the trace each segment, until it occurs in the same position as a in the last segment. Proposition c is true exactly with distance k_i, where i is the segment number. The trace σ ends with proposition a holding at position $n \cdot m$.

To recall the value of k_i via the measure from the monitor, we construct as follows: Proposition b holds at position $m-i$, and a and c never hold in ρ.

At the start of every segment, the a-position triggers a new measurement, which needs to be satisfied either by the left-hand-side $\Diamond_{\leq x} b$ of the disjunction or by the right-hand-side $\Diamond_{\leq y} c$ of the disjunction. In the continuation ρ, this is only possible with the left-hand side, since no c appears in ρ. This automatically fixes a measure for the higher-priority variable x, since the optimal achievable value for x is $m - i$.

To find the correct measure of y, we only need to consider the first i segments of σ, since for the remaining trace $\sigma \cdot \rho$, the left-hand-side of the disjunction holds, as the distances from a to b are smaller than $m - i$ as we progress through the trace. Now, since the k_i have been encoded into σ in ascending order, we need to make sure that the first i segments satisfy the right-hand-side of the implication, as they cannot satisfy the left-hand-side of the implication, since the a-b distances are larger than the measure of x. Therefore, the measure of y needs to be the maximum of the individual measurements of the first i segments, and the largest a-c distance happens to be exactly the value k_i.

Thus, any monitor has to store the full set K at the last position of trace σ, since we can recall any value k_i by a proper extension with ρ. With only logarithmic memory, it is impossible to store the set K: A binary encoding of K needs at least a memory of $O(n \cdot \log(m))$, i.e. linear space. Therefore, any online monitoring algorithm for PLTL needs more than logarithmic space. This concludes the proof of Theorem 2.2. $\square$

The formula $\varphi = \Box(a \rightarrow (\Diamond_{\leq x} b \vee \Diamond_{\leq y} c))$ with $x \sqsupset y$ used in the proof of Theorem 2.2 purposefully uses two features of PLTL to show the hardness result: the disjunction between the two measuring operators and the optimality requirement on the measure in combination with the parameter order. Both features together disallow the summarization (for example to only store the currently reached optimal value or value combination) of the measuring parts of the formula along the trace. For an online monitor, the disjunctions may create in every step a new measurement to be started from this point onward, and the parameter order with the optimality requirement requires the monitor to store measured values for one parameter according to the context of all other parameters, which may be recalled later because a higher-priority parameter reaches its optimal value in a certain context.

We now present two restrictions of PLTL with a logarithmic space monitoring procedure:

- *Deterministic PLTL*, a syntactic restriction in analogy to deterministic LTL [53], and

- *Unambigous PLTL*, a semantic restriction, which restricts the measuring but retains the full expressivity of LTL.

2.4 Measuring Automata

We use an automaton construction to allow us to implement the semantics of the logic. To check and measure PLTL formulae on a trace, we use extended finite-state automata, which maintain on a run – in addition to a current state – a fixed number of integer variables. These integer variables will be used to store the necessary information on the measurements along the trace.

Formally, a deterministic measuring automaton A is described by the tuple $A = (\Sigma, \Omega, Q, q_0, \Theta, \delta, \gamma, F, \omega)$ where Σ is the input alphabet, Ω the output domain, Q the state space, q_o the initial state, $\Theta : X \mapsto \{0, \infty\}$ an initial assignment, $\delta : Q \times \Sigma \mapsto Q \cup \{\bot\}$ describes the transition function, an update function $\gamma : Q \times \Sigma \mapsto (X \mapsto \mathbb{N}) \mapsto (X \mapsto \mathbb{N})$, a set of final states $F \subseteq Q$ and an output function $\omega : F \times (X \mapsto \mathbb{N}) \mapsto \Omega$.

A *run* of a measuring automaton A on a trace $\sigma = \sigma_0, \sigma_1, ... \sigma_{n-1} \in \Sigma^*$ of length n is a sequence of configurations $(s_0, \eta_0), (s_1, \eta_1) ... (s_n, \eta_n)$, where the ith configuration is a pair of state s_i and valuation η_i, which adhere to the following conditions:

- $s_0 = q_0$,
- $\eta_0 = \Theta$,
- $\forall i < n - 1.\ s_{i+1} = \delta(s_i, \sigma_i)$,
- $\forall i < n - 1.\ \eta_{i+1} = \gamma(s_i, \sigma_i)(\eta_i)$,
- $s_n \in F$.

The *result* of the run is determined by $\omega(s_n, \eta_n)$. Since A – more concretely δ and γ – are deterministic, we have the property that for every trace σ, A either has a unique run or no run at all. If A has a run on σ, we say that A accepts σ with result $\omega(s_n, \eta_n)$.

Defining updates on valuations with γ We restrict the possible functions γ by specifying them in terms of the following operations for a variable $x \in X$:

- Reset: $x := 0$,
- Increment: $x := x + 1$,
- Maximum: $x := \max(x, y)$, where $y \in X$,
- Minimum: $x := \min(x, y)$, where $y \in X$.

Due to these restrictions, on an input trace σ of length n, the values of the variables in X are either ∞ or bounded by n, since only one increment can be performed per trace event. Therefore, the values of variable $x \in X$ along a run can be encoded in binary in logarithmic space in the length of n.

Lemma 2.3. *The configuration of a measuring automaton can be represented in logarithmic space in the length of the input.*

2.5 Deterministic PLTL

The definition of deterministic PLTL was inspired by [53], that introduced deterministic LTL, which has a linear translation to universal Büchi automata, was defined. Our fragment PLTL$^{\text{det}}$ is defined by a syntactic restriction, which ensures the determinicity of the full transition relation of the automaton: the propositional and temporal, but also the measuring parts, i.e. the counters of the measuring automaton.

In deterministic PLTL, we remove disjunctive features of the logic by determinizing temporal operators like until and eventually and by restricting disjunctions. These features are determinized by using conjunctions with atomic propositions, which are immediately available from the inputs and resolve the remaining nondeterminism.

Formulas of deterministic PLTL are generated from the following syntax:

$$\begin{aligned}\varphi ::= \top \mid \bot \mid a \mid \neg a \mid \varphi \wedge \varphi \mid (a \wedge \varphi) \vee (\neg a \wedge \varphi) \mid \\ \bigcirc \varphi \mid \Diamond a \mid \Box \varphi \mid (a \wedge \varphi)\mathcal{U}(\neg a \wedge \varphi) \mid \Diamond_{\leq x} a \mid \Box_{\leq x} \varphi\end{aligned}$$

The semantics of deterministic PLTL stay the same as for PLTL.
One example formula in this fragment is the arbiter specification:

$$\varphi = \Box(r_1 \rightarrow \Diamond_{\leq x_1} g_1) \wedge \Box(r_2 \rightarrow \Diamond_{\leq x_2} g_2) \wedge \Box\neg(g_1 \wedge g_2),$$

which can be rewritten to:

$$\begin{aligned}\varphi = &\Box((\neg r_1 \wedge \top) \vee (r_1 \wedge \Diamond_{\leq x_1} g_1)) \\ &\wedge \Box((\neg r_2 \wedge \top) \vee (r_2 \wedge \Diamond_{\leq x_2} g_2)) \\ &\wedge \Box(((\neg g_1 \wedge \top) \vee (g_1 \wedge \neg g_2)) \wedge ((\neg g_2 \wedge \top) \vee (g_2 \wedge \neg g_1)))\end{aligned}$$

to fall into the syntactic fragment.

2.5.1 From Deterministic PLTL to Measuring Automata

A measuring automaton for a deterministic PLTL formula φ runs on the input alphabet $\Sigma = 2^{AP}$ and produces as an output the mapping $\Omega : X \mapsto \mathbb{N}$.

Similar to the standard closure-based automaton construction for LTL [22], we construct the state space of our measuring automata for deterministic PLTL from subsets of the *closure* of the formula (called *atoms*) and add counter variables.

The closure of a PLTL formula φ, denoted by $cl(\varphi)$, is the set of PLTL formulas that includes all subformulas of φ and the negations of the non-parametric subformulas of φ.

The states of the measuring automaton used to monitor and measure a PLTL^{det} formula φ are based on the atoms of $cl(\varphi)$. The current atom of the measuring automaton represents the state of the formula after processing a prefix of the trace.

An *atom* of a PLTL formula φ is a subset of formulas of $cl(\varphi)$, which adheres to the following properties:

- **Consistency w.r.t. propositional logic**: A subset $A \subseteq \text{cl}(\varphi)$ is consistent with respect to propositional logic if the following conditions hold: $\psi_1 \wedge \psi_2 \in A$ iff $\psi_1 \in A$ and $\psi_2 \in A$, $\psi \in A$ implies $\neg\psi \notin A$, and $\top \in \text{cl}(\varphi)$ implies $\top \in A$.

- **Local consistency w.r.t. the until operator**: A subset $A \subseteq \text{cl}(\varphi)$ is locally consistent with respect to the until operator if for all $\psi_1 \mathcal{U} \psi_2 \in \text{cl}(\varphi)$ the following conditions hold: $\psi_2 \in A$ implies $\psi_1 \mathcal{U} \psi_2 \in A$, $\psi_2 \notin A$ and $\psi_1 \mathcal{U} \psi_2 \in A$ implies $\psi_1 \in A$.

- **Local consistency w.r.t. the release operator**: A subset $A \subseteq \text{cl}(\varphi)$ is locally consistent with respect to the release operator if for all $\psi_1 \mathcal{R} \psi_2 \in \text{cl}(\varphi)$ the following conditions hold: $\psi_1 \in A$ and $\psi_2 \in A$ implies $\psi_1 \mathcal{R} \psi_2 \in A$, and $\psi_1 \mathcal{R} \psi_2 \in A$ implies $\psi_2 \in A$.

- **Local consistency w.r.t. the globally operator**: A subset $A \subseteq \text{cl}(\varphi)$ is locally consistent with respect to the globally operator if for all $\Box \psi \in \text{cl}(\varphi)$, we have that if $\Box \psi \in A$, then $\psi \in A$.

- **Local consistency w.r.t. the parametric globally operator**: A subset $A \subseteq \text{cl}(\varphi)$ is locally consistent with respect to the parametric globally operator if for all $\Box_{\leq x} \psi \in \text{cl}(\varphi)$, we have that if $\Box_{\leq x} \psi \in A$, then $\psi \in A$.

- **Maximality**: A subset $A \subseteq \mathrm{cl}(\varphi)$ is maximal if for all non-parametric subformulas $\psi \in cl$, we have that either $\psi \in A$ or $\neg\psi \in A$.

The set of atoms of a PLTL formula φ is denoted by At_φ.

To implement the semantics of the temporal operators of PLTL, we define a successor relation on atoms.

Let $\rightarrow \subseteq \mathrm{At}_\varphi \times 2^{AP} \times \mathrm{At}_\varphi$ denote the successor relation between the atoms of φ. For each $t \in \mathrm{At}_\varphi$, $e \in 2^{AP}$, $t' \in \mathrm{At}_\varphi$, we have that $(t, e, t') \in \rightarrow$, also denoted as $t \xrightarrow{e} t'$, if t' is the smallest set such that the following conditions hold:

- $t' \cap AP = e$.
- If $\bigcirc \psi \in t$, then $\psi \in t'$.
- If $\square \psi \in t$, then $\square \psi \in t'$.
- If $\psi_1 \mathcal{U} \psi_2 \in t$, then either $\psi_2 \in t$ or $\psi_1 \in t$ and $\psi_1 \mathcal{U} \psi_2 \in t'$.
- If $\diamondsuit \psi \in t$, then $\psi \in t$ or $\diamondsuit \psi \in t'$.
- If $\diamondsuit_{\leq x} \psi \in t$, then $\psi \in t$ or $\diamondsuit_{\leq x} \psi \in t'$.
- If $\square_{\leq x} \psi \in t$, then $\psi \in t$ and $\psi \notin t'$ or $\square_{\leq x} \psi \in t'$.

Note that the parameters x are kept as symbols in the successor relation, as the counters are handled separately.

In the case of $\mathrm{PLTL}^{\mathsf{det}}$, this successor relation yields a deterministic transition function for the automaton, since the incoming event e fully determines the next atom t'. All possible disjunctions in these conditions are resolved, since the target formulas of eventually-operators and until-formulas are either directly atomic propositions or are protected by atomic propositions immediately available from e.

Lemma 2.4. *For every atom $t \in At_\varphi$ and every event $e \in 2^{AP}$ of a $PLTL^{\mathsf{det}}$-formula φ, there is at most one atom $t' \in At_\varphi$ such that $t \xrightarrow{e} t'$.*

The transition function δ of the measuring automaton can then be defined according to the relation $\rightarrow$:

- For the designated initial state q_0 (the only state which is not an atom, because no events have been seen so far), the immediate successor $\delta(q_0, e)$ is the unique atom which contains φ and is consistent with e, i.e. the atom $t \in \mathrm{At}_\varphi$ where $\varphi \in t$ and $t \cap AP = e$ or $\perp$ if no such atom exists.

- For all other states $t \in \mathrm{At}_\varphi$, the next state $\delta(t,e)$ is either the unique atom $t' \in \mathrm{At}_\varphi$ for $t \xrightarrow{e} t'$, or $\perp$ if no such atom exists.

The update function γ of the measuring automaton keeps track of two counter variables n_x and m_x for each parameter $x \in X$. The variable n_x is a counter, which keeps track of the current value of x, i.e. the number of steps since which the subformula associated to x has been present in the trace of atoms. The purpose of the second variable m_x is to record the extremal value of the possibly multiple measurements of n_x, since the temporal operators of the formula may introduce the parametric subformula multiple times into the current atom, for example for the formula $\square((\neg r \wedge \top) \vee (r \wedge \diamondsuit_{\leq x} g))$. In this example formula, n_x will keep track of the number of steps until g is true since the last r, and m_x will record the maximal r-g distance along the trace. For formulas with parametric operators of other polarity, such as $\square(q \wedge \square_{\leq x} p)$, m_x will record the minimal number of steps until p holds after a q appears.

The initialization of the counter variables via Θ also depends on the polarity of parametric operators:

- For upward-closed operators, where m_x is used to keep track of the maximal (completed) measurement of n_x seen so far, we initialize $\Theta(n_x) = 0$ and $\Theta(m_x) = 0$, i.e. the weakest guarantee.
- For downward-closed operators, where m_x keeps track of the minimal (completed) measurement of n_x seen so far, we initialize $\Theta(n_x) = 0$ and $\Theta(m_x) = \infty$, again the weakest guarantee.

To properly define the update function $\gamma(s,e)$, which maps a state s and an input event e to a mapping between successive valuations, it is important to distinguish situations where a parametric subformula is freshly *generated*, i.e. a new measurement is started, from situations where a parametric subformula was already present, i.e. a previously started measurement is continued.

A formula $\psi \in t$ is *generated* in a pair of atoms $t, t' \in \mathrm{At}_\varphi$, denoted by the predicate $\mathrm{generated}(\psi, t, t')$, if one of the following conditions is met:

- ψ is a direct subformula[1] of some other formula in t', or
- $\bigcirc \psi \in t$.

[1] A direct subformula of a formula is a subformula for which a single application of the syntax grammar yields the original formula.

To also cover the designated initial state q_0 with the definition and to simplify notation, we set generated(ψ, q_0, t') to **true** for all t', ψ.

We give the definition for $\gamma(s,e)$ per parametric subformula and associated counters.

In the case of upward-closed parametric subformulas, we take the maximal value of the individual measurements as the strongest guarantee for the formula. For every upward-closed parametric subformula $\Diamond_{\leq x} \psi \in \delta(s,e)$:

$$\gamma(s,e) = \begin{cases} m'_x := m_x,\ n'_x := 0 & \text{if generated}(\Diamond_{\leq x} \psi, s, \delta(s,e)) \\ m'_x := \max(m_x, n_x),\ n'_x := 0 & \text{if } \psi \in t' \wedge \neg\text{generated}(\Diamond_{\leq x} \psi, s, \delta(s,e)) \\ m'_x := m_x,\ n'_x := n_x + 1 & \text{if } \psi \notin t'. \end{cases}$$

In the case of downward-closed parametric subformulas, we restart counting for the strongest guarantee whenever we re-enter the parametric subformula into the atom, since we in the end need to obtain the minimum of all measurements. For every downward-closed parametric subformula $\Box_{\leq x} \psi \in \delta(s,e)$:

$$\gamma(s,e) = \begin{cases} m'_x := m_x, n'_x := 0 & \text{if } \psi \in t', \text{generated}(\Box_{\leq x} \psi, s, \delta(s,e)) \\ m'_x := m_x, n'_x := n_x + 1 & \text{if } \psi \in t', \neg\text{generated}(\Box_{\leq x} \psi, s, \delta(s,e)) \\ m'_x := \min(m_x, n_x), n'_x := 0 & \text{if } \psi \notin t'. \end{cases}$$

If the parametric subformulas of a parameter x do not occur in an atom, we keep their counter variables unchanged: $m'_x := m_x$ and $n'_x := n_x$.

The final states F of the measuring automaton are exactly the atoms without unfulfilled obligations (and the designated initial state q_0): $t \in F$ iff for all $\psi_1 \mathcal{U} \psi_2 \in t$ also $\psi_2 \in t$, for all $\Diamond \psi \in t$ also $\psi \in t$, and there is no $\bigcirc \psi \in t$.

For such states $t \in F$, $\omega(t, \eta)$ returns the variable value of m_x for upward-closed parameters if $\Diamond_{\leq x} \psi \notin t$ and $\max(m_x, n_x)$ for $\Diamond_{\leq x} \psi \in t$, and for downward-closed parameters it returns the value of m_x if $\Box_{\leq x} \psi \notin t$ and $\min(m_x, n_x)$ if $\Box_{\leq x} \psi \in t$, to take into account the currently active counter evidence.

2.5.2 Correctness

We split the proof of the correctness of the measure automaton construction into two parts:

Lemma 2.5. *If there exists a run $\pi = (t_0, \eta_0), (t_1, \eta_1), \ldots (t_n, \eta_n)$ of A_φ on the trace $\sigma = e_0 e_1 \ldots e_{n-1} \in (2^{AP})^*$ with result r, then $(\sigma, 0, r) \models \varphi$.*

To prove Lemma 2.5, we first prove that the atom sequence $t_0, t_1, \ldots t_n$ is in correspondence with the semantics.

Lemma 2.6. *For all subformulas $\psi \in cl(\varphi)$ and for all positions i, if $\psi \in t_i$, then $(\sigma, i-1, r) \models \psi$.*

The proof of Lemma 2.6 proceeds by induction on the length of the trace, progressing backwards from the last position.

Proof. We prove, by induction on i, starting in the last atom t_n, the slightly stronger claim that, for all positions i and all non-parametric subformulas ψ, $\psi \in t_i$ iff $(\sigma, i-1, r) \models \psi$, and for all parametric subformulas ψ, if $\psi \in t_i$ then $(\sigma, i-1, r) \models \psi$.
Base case $k = n$: Since π is accepting, t_n is final.
By structural induction on ψ.
For all subformulas ψ of φ, $\psi \in t_i \Rightarrow (\sigma, i-1, r) \models \psi$.

- Let $\psi = \mathbf{true}$. $(\sigma, n-1, r) \models \mathbf{true}$.
- Let $\psi = p$. $\exists\, t_{n-1} \xrightarrow{s_{n-1}} t_n$. By definition of $\longrightarrow$, $p \in t_n$, and $p \in s_{n-1}$. Therefore, $(\sigma, n-1, r) \models p$.
- Let $\psi = \neg\mu$. $\neg\mu \in t_n$. $\mu \notin t_n$. Therefore, $(\sigma, n-1, r) \not\models \mu$.
- Let $\psi = \mu \wedge \nu$. Since t_n is an atom, $\mu \in t_n$ and $\nu \in t_n$. Therefore, $(\sigma, n-1, r) \models \mu$, $(\sigma, n-1, r) \models \nu$ and thus $(\sigma, n-1, r) \models \mu \wedge \nu$.
- Let $\psi = (p \wedge \mu) \vee (\neg p \wedge \nu)$. Since t_n is an atom, either $p \wedge \mu \in t_n$ or $\neg p \wedge \nu \in t_n$. Therefore, either $(\sigma, n-1, r) \models p \wedge \mu$ or $(\sigma, n-1, r) \models \neg p \wedge \nu$ and thus $(\sigma, n-1, r) \models (p \wedge \mu) \vee (\neg p \wedge \nu)$.
- Let $\psi = \bigcirc \mu$. Since t_n is final, there is no such formula.
- Let $\psi = (p \wedge \mu)\mathcal{U}(\neg p \wedge \nu)$. Since t_n is final, we have $\neg p \wedge \nu \in t_n$. Therefore, $(\sigma, n-1, r) \models p \wedge \nu$ and $(\sigma, n-1, r) \models (p \wedge \mu)\mathcal{U}(\neg p \wedge \nu)$.
- Let $\psi = \Diamond_{\leq x} \mu$. Since t_n is final, we have $\mu \in t_n$. Therefore $(\sigma, n-1, r) \models \mu$ and $(\sigma, n-1, r) \Diamond_{\leq x} \models \mu$.
- Let $\psi = \Box_{\leq x} \mu$. Since t_n must be locally consistent with respect to the globally operator, we have $\mu \in t_n$ and $(\sigma, n-1, r) \models \mu$. Thus, $(\sigma, n-1, r) \Box_{\leq x} \models \mu$.

Inductive case $k+1 \Rightarrow k$: By structural induction on ψ.

- Let $\psi = \textbf{true}$. $(\sigma, k-1, r) \models \textbf{true}$.
- Let $\psi = p$. $\exists t_{k-1} \xrightarrow{s_k} t_k$. By definition of $\longrightarrow$, $p \in t_k, s_k$. Therefore, $(\sigma, k-1, r) \models p$.
- Let $\psi = \neg\mu$. $\neg\mu \in t_k$. $\mu \notin t_k$. Therefore, $(\sigma, k-1, r) \not\models \mu$.
- Let $\psi = \mu \wedge \nu$. Since t_k is an atom, $\mu \in t_k$ and $\nu \in t_k$. Therefore, $(\sigma, k-1, r) \models \mu$, $(\sigma, k-1, r) \models \nu$ and thus $(\sigma, k-1, r) \models \mu \wedge \nu$.
- Let $\psi = (p \wedge \mu) \vee (\neg p \wedge \nu)$. Since t_k is an atom, either $p \wedge \mu \in t_k$ or $\neg p \wedge \nu \in t_k$. Therefore, either $(\sigma, k-1, r) \models p \wedge \mu$ or $(\sigma, k-1, r) \models \neg p \wedge \nu$ and thus $(\sigma, k-1, r) \models (p \wedge \mu) \vee (\neg p \wedge \nu)$.
- Let $\psi = \bigcirc \mu$. $\exists t_k \xrightarrow{s_k} t_{k+1}$. By definition of $\longrightarrow$ and IH, $\mu \in t_{k+1}$, therefore $(\sigma, k, r) \models \mu$ and $(\sigma, k-1, r) \models \bigcirc \mu$.
- Let $\psi = (p \wedge \mu)\mathcal{U}(\neg p \wedge \nu)$. By definition of $\longrightarrow$ and local consistency of the atom t_k, we have either $\neg p \wedge \nu \in t_k$ or $(p \wedge \mu) \in t_k$ and $(p \wedge \mu)\mathcal{U}(\neg p \wedge \nu) \in t_{k+1}$. Therefore, either $(\sigma, k-1, r) \models p \wedge \mu$ and (by IH) $(\sigma, k, r) \models (p \wedge \mu)\mathcal{U}(\neg p \wedge \nu)$ or $(\sigma, k-1, r) \models \neg p \wedge \nu$. Thus, either way $(\sigma, k-1, r) \models (p \wedge \mu)\mathcal{U}(\neg p \wedge \nu)$.
- Let $\psi = \Diamond_{\leq x} \mu$. We either have $\psi \in t_{k+1}$, with some $n_x = \eta_{k+1}(n_x)$, or $\mu \in t_k$. In the latter case, we trivially have $(\sigma, k-1, r) \models \Diamond_{\leq x} \mu$. In the former case, we have that $\mu \in t_{k+j+1}$ for some distance j. By IH, we have $(\sigma, k, r) \models \Diamond_{\leq x} \mu$, thus we also have $(\sigma, k, r') \models \Diamond_{\leq x} \mu$ for some r′.
- Let $\psi = \Box_{\leq x} \mu$. Since t_{k+1} must be locally consistent with respect to the globally operator, we have $\mu \in t_{k+1}$, $\mu \in t_k$ and $(\sigma, k, r) \models \mu$ as well as $(\sigma, k-1, r) \models \mu$. By IH, $(\sigma, k, r)\Box_{\leq x} \models \mu$, and thus $(\sigma, k-1, r')\Box_{\leq x} \models \mu$ for some r′.

This concludes the proof of Lemma 2.6. □

It remains to show the remaining part of Lemma 2.5, which is that the result of the run is the measure of φ.

Lemma 2.7. *For all positions i and atoms t_i, and for every eventuality $\Diamond_{\leq x} \mu$ in t_i, the distance $d_i(x)$ to the next atom containing μ adheres to $d_i(x) \leq r(x) - \eta_i(n_x)$, and for every parameterized globally operator $\Box_{\leq x} \mu$ in t_i, the distance $d_i(x)$ to the next atom containing $\neg\mu$ adheres to $d_i(x) \geq r(x) - \eta_i(n_x)$.*

We show, inductively, that for a subformula $\Diamond_{\leq x} \psi$ in atom t_i, the distance to the next atom with $[\psi]$ is at most $r(x) - \eta_i(n_x)$ steps; and, likewise, that for a subformula $\Box_{\leq x} \psi$ in atom t_i, the distance to the next atom with $\neg[\psi]$ is at least $r(x) - \eta_i(n_x)$ steps.

Proof. We prove that the final valuation η_n which determines the result r, agrees with the valuation r such that $(\sigma, 0, r) \vDash \varphi$, by induction on trace σ. More precisely, we prove that for every position i, for every eventuality $\Diamond_{\leq x} \mu$ in atom t_i, the distance $d_i(x)$ to the next atom containing μ adheres to $d_i(x) \leq r(x) - \eta_i(n_x)$ and for every parameterized globally operator $\Box_{\leq x} \mu$ in t_i, the distance $d_i(x)$ to the next atom containing $\neg\mu$ adheres to $d_i(x) \geq r(x) - \eta_i(n_x)$.
Base case $i = 0$:
We split the cases based on the parametric operators:

- Let $\psi = \Diamond_{\leq x} \mu$. We have $\eta_0(m_x) = 0$ and $\eta_0(n_x) = 0$ by initialization with θ. We prove by contradiction that $r(x) = \omega(\eta_n(m_x))$ contains the maximum η-value of n_x along the trace: If there were a position k, where $n_x > \eta_n(m_x)$, then k must have been followed by a position $j > k$ where $\mu \notin t_j$. By definition of γ, the newly stored maximum in $\eta_j(m_x)$ must be greater than $\eta_n(m_x)$. The position j must occur before the end of the trace, since we do not allow eventualities in final atoms. Thus, we have that $d(x) \leq r(x)$.

- Let $\psi = \Box_{\leq x} \mu$. We have $\eta_0(m_x) = \infty$ and $\eta_0(n_x) = 0$ by initialization with θ. We prove by contradiction that $r(x) = \omega(\eta_n(m_x))$ contains the minimum η-value of n_x along the positions k of the trace where $\neg\mu \in t_k$: If there were a position j, where $\neg\mu \in t_j$ and $n_x < \eta_n(m_x)$, then by definition of γ, the newly stored minimum in $\eta_j(m_x)$ must be smaller than $\eta_n(m_x)$. Since the minimum is also applied by ω on the last position of the run, we have that $d(x) \geq r(x)$.

Inductive case $i \Rightarrow i + 1$:
We split the cases based on the parametric operators:

- Let $\psi = \Diamond_{\leq x} \mu$. If $\mu \in t_{i+1}$, then $d_{i+1}(x) = 0$ and $d_{i+1}(x) \leq r(x) - \eta_{i+1}(n_x)$, since $r(x)$ contains the maximum of all n_x-values. If $\mu \notin t_{i+1}$, then we have that $d_{i+1}(x) = d_i(x) - 1$ and by definition of γ, $\eta_{i+1}(n_x) = \eta_i(n_x) + 1$. Thus, together with the induction hypothesis, the inequality still holds.

- Let $\psi = \Box_{\leq x} \mu$. If $\neg\mu \in t_{i+1}$, then $d_{i+1}(x) = 0$ and $d_{i+1}(x) \geq r(x) - \eta_{i+1}(n_x)$, since $r(x)$ contains the minimum of all n_x-values in $\neg\mu$-atoms. If $\mu \in t_{i+1}$, then we have that $d_{i+1}(x) = d_i(x) - 1$ and by definition of γ, $\eta i + 1(n_x) = \eta i(n_x) + 1$. Thus, together with the induction hypothesis, the inequality still holds.

This concludes the proof of Lemma 2.7. □

Thus, for $r(x) = \omega(t_n, \eta_n)(x)$, we have $(\sigma, 0, r) \vDash \varphi$.

Lemma 2.8. *For a trace $\sigma = e_0 e_1 \dots e_{n-1} \in (2^{AP})^*$, if $(\sigma, 0, r) \vDash \varphi$, then there exists a run $\pi = (t_0, \eta_0), (t_1, \eta_1), \dots (t_n, \eta_n)$ of A_φ on σ with result r', where $r \sqsupseteq r'$.*

For the reverse direction, stated as Lemma 2.8, we first construct the sequence of atoms corresponding to the given trace. Based on the semantics, we show inductively that the subformulas in the atoms hold over the respective suffixes.

Lemma 2.9. *For a trace $\sigma = e_0 e_1 \dots e_{n-1}$ and a deterministic PLTL formula φ, if $(\sigma, 0, r) \vDash \varphi$, then there exists an atom sequence $t_0 \dots t_n$ such that t_0 is the unique atom that contains φ and is consistent with e_0, $t_i \xrightarrow{s_i} t_{i+1}$ for all $i = 0 \dots n-2$, and for every subformulas ψ and position $i = 0 \dots n-1$, if $\psi \in t_i$, $(\sigma, i-1, r) \vDash \psi$.*

We complete the atom sequence of Lemma 2.9 into a complete run by computing the values of n_x and m_x for each parameter x and each trace position according to the definition of the automaton.

Proof. There exists a sequence $u_1, u_2, \dots u_n$ with $\forall i. u_i \subseteq$ non-parametric formulas of φ with the property that for all positions i and for all subformulas ψ, $\psi \in u_i \Leftrightarrow (\sigma, i-1, r) \vDash \psi$ for some r.
We proceed by induction on σ, constructing the sequence $u_1, u_2, \dots u_n$ backwards.
For all non-parametric subformulas:
Base case $k = n$: By structural induction on ψ:

- Let $\psi = \mathbf{true}$. $\mathbf{true} \in u_n$.
- Let $\psi = p$. If $p \in s_{n-1}$, then $p \in u_n$.
- Let $\psi = \neg\mu$. If $(\sigma, n-1, r) \nvDash \mu$, $\neg\mu \in u_n$.
- Let $\psi = \mu \wedge \nu$. If $\mu \in u_n$ and $\nu \in u_n$ then $\mu \wedge \nu \in u_n$.
- Let $\psi = (p \wedge \mu) \vee (\neg p \wedge \nu)$. If $p \in u_n$ and $p \wedge \mu \in u_n$ or $\neg p \in u_n$ and $\neg p \wedge \nu \in u_n$ then $(p \wedge \mu) \vee (\neg p \wedge \nu) \in u_n$.
- Let $\psi = \bigcirc\mu$. $\bigcirc\mu \in u_n$.
- Let $\psi = (p \wedge \mu)\mathcal{U}(\neg p \wedge \nu)$. If $(\neg p \wedge \nu) \in u_n$, then $(p \wedge \mu)\mathcal{U}(\neg p \wedge \nu) \in u_n$.

Inductive case $k+1 \Rightarrow k$: Induction on ψ:

- Let $\psi = \mathbf{true}$, and $\mathbf{true} \in u_k$.
- Let $\psi = p$. If $p \in s_{k-1}$, then $p \in u_k$.
- Let $\psi = \neg\mu$. If $(\sigma, k-1, r) \nvDash \mu$, $\neg\mu \in u_k$.
- Let $\psi = \mu \wedge \nu$. If $\mu \in u_k$ and $\nu \in u_k$ then $\mu \wedge \nu \in u_k$.
- Let $\psi = (p \wedge \mu) \vee (\neg p \wedge \nu)$. If $p \in u_k$ and $p \wedge \mu \in u_k$ or $\neg p \in u_k$ and $\neg p \wedge \nu \in u_k$ then $(p \wedge \mu) \vee (\neg p \wedge \nu) \in u_k$.
- Let $\psi = \bigcirc \mu$. If $\mu \in u_{k+1}$, then $\bigcirc \mu \in u_k$.
- Let $\psi = (p \wedge \mu)\mathcal{U}(\neg p \wedge \nu)$. If $(\neg p \wedge \nu) \in u_k$ or $(p \wedge \mu) \in u_{k+1}$ and $(p \wedge \mu)\mathcal{U}(\neg p \wedge \nu) \in u_{k+1}$, then $(p \wedge \mu)\mathcal{U}(\neg p \wedge \nu) \in u_k$.

Then there exists a sequence $\pi = t_0, t_1, \dots t_n$, such that for all positions i, we have that $u_i \subseteq t_i \subseteq \mathrm{cl}(\varphi)$ and for all subformulas ψ, $\psi \in t_i \Rightarrow (\sigma, i-1, r)$.
We proceed by induction on σ, constructing the accepted sequence π.
Base case $k = 0$: q_0.
Base case $k = 1$: By structural induction on ψ.
Since $u_1 \subseteq t_1$, only the parametric operators remain.

- Let $\psi = \Diamond_{\leq x} \mu$. If $\Diamond_{\leq x} \mu$ is generated by a subformula of u_1, $\Diamond_{\leq x} \mu$ is in t_1.
- Let $\psi = \Box_{\leq x} \mu$. If $\Box_{\leq x} \mu$ is generated by a subformula of u_1, $\Box_{\leq x} \mu$ is in t_1.

Inductive case $k \Rightarrow k+1$: By structural induction on ψ. Since $u_{k+1} \subseteq t_{k+1}$, only the parametric operators remain.

- Let $\psi = \Diamond_{\leq x} \mu$. If $\psi \in t_k$ and $\mu \notin t_k$, $\psi \in t_{k+1}$.
- Let $\psi = \Box_{\leq x} \mu$. If $\psi \in t_k$ and $\mu \notin t_{k+1}$, then $\psi \in t_{k+1}$.

This concludes the proof of Lemma 2.9. □

Lemma 2.10. *For a trace $\sigma = e_0 e_1, \dots e_{n-1}$ and a deterministic PLTL formula φ, if $(\sigma, 0, r) \vDash \varphi$, then there exists a run of A_φ, $\pi = (t_0, \eta_0), (t_1, \eta_1) \dots (t_n, \eta_n)$ where $t_i \xrightarrow{e_i} t_{i+1}$ and for all subformulas and positions i, if $\psi \in t_i$, $(\sigma, i-1, r) \vDash \psi$, with result $\omega(e_n, \eta_n) \sqsubseteq r$.*

Proof. To prove Lemma 2.10, that the result of the run is at least as good as r, we show, inductively, (1) that for each subformula $\Diamond_{\leq x} \psi$ and each trace position i, $\eta_i(n_x)$ is less than or equal to the difference of $r(x)$ and the distance of the closest atom that contains ψ; and (2) that for each subformula $\Box_{\leq x} \psi$ and each trace position i, the sum of $\eta_i(n_x)$ and the distance of the closest atom that contains $\neg[\psi]$ is greater than or equal to $r(x)$. Since m_x maintains for $\Diamond_{\leq x} \psi$ and for $\Box_{\leq x} \psi$, the maximum and minimum measure, respectively, the Lemma 2.10 follows. □

Theorem 2.11. *For every PLTL$^{\mathsf{det}}$-formula φ, there exists a deterministic measuring automaton A_φ with a linear number of states in $|\varphi|$ and a linear number of variables X in $|\varphi|$, such that for every sequence $\sigma \in (2^{AP})^*$, σ is accepted by A_φ with result r iff r is the measure of φ on σ.*

Proof. Since A_φ only has a at most one run on a given trace due to determinicity and therefore a unique result, if a run exists, Lemma 2.5 and Lemma 2.8 imply that the result of A_φ on a trace is the measure of φ on the given trace. □

Lemma 2.12. *The configuration of a measuring automaton can be represented in logarithmic space in the length of the input sequence.*

From Theorem 2.11 and Lemma 2.12 (which follows directly from the construction) it follows that the space required by the online monitor is linear in the size of the specification and logarithmic in the length of the trace.

Corollary 2.13. *A PLTL^{det} formula φ can be measured in linear space in the size of φ and logarithmic space in the length of the trace.*

2.6 Unambiguous PLTL

In PLTL$^{\mathsf{det}}$, we have traded the full expressivity of LTL against a simpler parametric monitoring and measuring problem, where monitoring a single trace always lead to a unique result: the measure of the trace. In this section, we recover the full expressivity of LTL, but instead only determinize the measuring parts of the formula. This leads to an exponential construction of the monitoring automaton in the size of the formula, but keeps the logarithmic dependency on the trace length.

As the size of the formula is usually known at design-time of a system, this exponential component in the complexity does not necessarily lead to a problem in practice, as the formula could be refined in a more deterministic manner while expressing a strengthened property.

While PLTL$^{\mathsf{det}}$ is a syntactic restriction on the permitted formulas, unambiguous PLTL is a semantic restriction on the formulas. The semantics of disjunctive features of the logic are modified to remove precisely the expensive nature of measuring under uncertainty with respect to the satisfaction of complex future temporal dependencies.

Syntax and Semantics of unambiguous PLTL The syntax of unambiguous PLTL is the same as PLTL as given in Section 2.1. The semantics of temporal and logical operators with disjunctive features is modified as follows:

- $(\sigma, k, \alpha) \vDash \varphi_1 \vee \varphi_2$, iff $(\sigma, k, \alpha) \vDash \varphi_1$ or $(\sigma, k, \alpha) \vDash \neg[\varphi_1] \wedge \varphi_2$.
- $(\sigma, k, \alpha) \vDash \varphi_1 \mathcal{U} \varphi_2$, iff $\exists i. k \leq i < |\sigma|$, where $(\sigma, i, \alpha) \vDash \varphi_2$ and $\forall j. k \leq j < i$, we have $(\sigma, j, \alpha) \vDash \varphi_1 \wedge \neg[\varphi_2]$.
- $(\sigma, k, \alpha) \vDash \varphi_1 \mathcal{R} \varphi_2$, iff
  either $\exists i. k \leq i < |\sigma|$, where $(\sigma, i, \alpha) \vDash \varphi_1 \wedge \varphi_2$ and $\forall j. k \leq j \leq i$, we have $(\sigma, j, \alpha) \vDash \neg[\varphi_1] \wedge \varphi_2$,
  or $\forall j. k \leq j < |\sigma|$ we have $(\sigma, j, \alpha) \vDash \neg[\varphi_1] \wedge \varphi_2$.
- $(\sigma, k, \alpha) \vDash \Diamond_{\leq x} \varphi$, iff
  $\exists i. k \leq i \leq \alpha(x)$ and $k + i < |\sigma|$, where $(\sigma, k + i, \alpha) \vDash \varphi$ and $\forall j. k \leq j < i$, we have $(\sigma, j, \alpha) \vDash \neg[\varphi]$,
  or we have that $k + \alpha(x) \geq |\sigma|$ and $\forall j. k \leq j < |\sigma|$, we have $(\sigma, j, \alpha) \vDash \neg[\varphi]$.

These changes ensure that we never have the situation where a measurement needs to be combined disjunctively from both sides of a logical disjunction, and that the temporal operators have to match on the first opportunity.

Note that this change removes commutativity of disjunction, if there is a parametric subformula on its right-hand side. Also note that for formulas in the syntax of PLTL$^{\mathsf{det}}$, both semantics agree, guaranteeing that unambiguous PLTL is a strict generalization of PLTL$^{\mathsf{det}}$.

Example Consider the trace $\sigma = \{a\}\{a, b\}\varnothing$ and the PLTL formula $\varphi = a\mathcal{U}\Diamond_{\leq x} b$. According to the original semantics in Section 2.1, both for the valuation $\alpha : \{x \mapsto 0\}$ and the valuation $\alpha' : \{x \mapsto 1\}$, we have that $(\sigma, 0, \alpha) \vDash \varphi$ and $(\sigma, 0, \alpha') \vDash \varphi$, respectively. This is due to the two possibilities to satisfy the goal of the until formula by satisfying $\Diamond_{\leq x} b$. Under the unambiguous semantics, we immediately use the first position where the subformula is satisfied, so we only have $(\sigma, 0, \alpha') \vDash \varphi$ with measure $x \mapsto 1$, because the LTL abstraction $[\Diamond_{\leq x} b] = \Diamond b$ is satisfied at position 0, and the requirement would be that $\neg[\Diamond_{\leq x} b]$ is satisfied.

2.6.1 From Unambiguous PLTL to Measuring Automata

The disjuncts of unambiguous PLTL are not guarded by atomic propositions, but instead are guarded by full temporal subformulas. The truth values of these formulas may - in the worst case - only be resolved at the end of the trace, so we have to extend the construction of the automaton to keep track of all possible runs resulting in a positive result.

Still, the nondeterminism in the successor relation of the measuring automaton only depends on the behavior of the non-parametric subformulas, as the following lemmata show.

Lemma 2.14. *Let t, t' be two atoms in a state s of the measuring automaton reached after reading some trace $\sigma = e_0 e_1 e_2 \dots e_n$, such that there exists a sequence $t_1 t_2 \dots t_{n+1}$ of atoms with $t_i \in s_i$ for $i = 1 \dots n+1$ and $t_i \xrightarrow{e_i} t_{i+1}$ for $i = 1 \dots n$ such that $t_{n+1} = t$, and a sequence $t'_1 t'_2 \dots t'_{n+1}$ of atoms with $t'_i \in s_i$ for $i = 1 \dots n+1$ and $t'_i \xrightarrow{e_i} t'_{i+1}$ for $i = 1 \dots n$ such that $t'_{n+1} = t'$. If t_i and t'_i agree on the non-parametric formulas for all $i = 1 \dots n+1$, then $t = t'$.*

Proof. By induction on the length of σ. By contradiction, let i be the first position where a parametric subformula ψ occurs in t_i but not in t'_i, and let ψ be the largest (in some total order that extends formula length) such subformula.

- if ψ occurs as the direct subformula of a disjunction or until operator in t_i and t'_i, then this contradicts unambiguity in disjunction or until;
- if ψ occurs as the direct subformula of a conjunction or negation, this contradicts the consistency requirement.

Hence, no such ψ can exist. □

Lemma 2.15. *Let t_1, t_2 be two atoms in a state of the measuring automaton reached after reading some trace σ, and let t' be an atom in the state reached after reading the additional event e, such that $t_1 \xrightarrow{e} t'$ and $t_2 \xrightarrow{e} t'$. Then $t_1 = t_2$.*

Proof. We first show the result for t_1 and t_2. Then, repeating the argument on the predecessor atoms, the corresponding atoms on the sequences of atoms leading to t_1 and t_2, agree on the non-parametric subformulas. By Lemma 2.14, it then follows that t_1 and t_2 also agree on the parametric formulas.

Suppose there exist $t_1, t_2 \in \mathrm{At}_\varphi$ with $t_1 \xrightarrow{e} t'$, $t_2 \xrightarrow{e} t'$, $t_1 \cap \mathrm{AP} = t_2 \cap \mathrm{AP} = e_1$, and there exists a non-parametric subformula $\psi \in \mathrm{cl}(\varphi)$ such that $\psi \in t_1 \setminus t_2$. Let ψ be the smallest (in some total order that extends formula length) such formula in t_1.

- $\psi \in \mathrm{AP}$: ψ cannot be an atomic proposition, because $t_1 \cap \mathrm{AP} = t_2 \cap \mathrm{AP}$;
- $\psi = \mu \wedge \eta$, $\psi = \mu \vee \eta$, or $\psi = \neg\mu$: ψ cannot be a conjunction, disjunction, or negation because of the assumption that ψ is the smallest subformula where t_1 and t_2 disagree;
- if ψ is a (non-parametric) temporal formula, then, because of maximality, $\neg\psi \in t_2$;
  - if $\psi = \bigcirc\eta$, then $\eta \in t'$ because of $t_1 \xrightarrow{e} t'$, and $\neg\eta \in t'$, because $\bigcirc\neg\psi \in t_2$ and $t_2 \xrightarrow{e} t'$. This contradicts the consistency of t'.
  - if $\psi = \mu\mathcal{U}\eta$, then there are two cases: (1) $\eta \in t_1$; since ψ is the smallest formula in t_1 where t_1 and t_2 disagree, we have that $\eta \in t_2$; since, by maximality, also $\neg\psi \in t_2$, this contradicts the consistency of t_2. (2) $\eta \notin t_q$, therefore by maximality, $\neg\eta \in t_1$, and, again because ψ is the smallest formula in t_1 where t_1 and t_2 disagree, $\neg\eta \in t_2$. Because of $t_1 \xrightarrow{e} t'$ we thus have ψ in t', and by $t_2 \xrightarrow{e} t'$, we have that $\neg\psi \in t'$. This contradicts the consistency of t'.
  - if $\psi = \mu\mathcal{R}\eta$, then there are two cases: (1) $\mu \in t_1$; since μ is the smallest formula in t_1 where t_1 and t_2 disagree, we have that $\mu \in t_2$; since, by maximality, also $\neg\psi \in t_2$, this contradicts the consistency of t_2. (2) $\mu \notin t_1$, therefore by maximality, $\neg\mu \in t_1$, and, again because ψ is the smallest formula in t_1 where t_1 and t_2 disagree, $\neg\mu \in t_2$. Because of $t_1 \xrightarrow{e} t'$ we thus have ψ in t', and by $t_2 \xrightarrow{e} t'$, we have that $\neg\psi \in t'$. This contradicts the consistency of t'.

□

Lemma 2.16. *Every final state $f \in F$ reached by the measuring automaton on some trace contains exactly one atom, and this atom has no unfulfilled obligations.*

Proof. Assume that there exist two different atoms $t_1, t_2 \in f, t_1 \neq t_2$. We first show that t_1 and t_2 agree on the non-parametric formulas. In analogy to the proof of Lemma 2.15 it then follows that the corresponding atoms on the sequences of atoms

leading to t_1 and t_2 agree on the non-parametric subformulas. By Lemma 2.14, it then follows that t_1 and t_2 also agree on the parametric formulas.
By contradiction, let ψ be the smallest (in some total order that extends formula length) non-parametric formula that is in t_1 but not in t_2.

- $\psi \in \mathrm{AP}$: ψ cannot be an atomic proposition, because the atoms of a state agree on the atomic propositions.
- $\psi = \mu \wedge \eta$, $\psi = \mu \vee \eta$, or $\psi = \neg\mu$: ψ cannot be a conjunction, disjunction, or negation because of the assumption that ψ is the smallest subformula where t_1 and t_2 disagree;
- $\psi = \bigcirc \eta$ is not allowed for final states;
- if $\psi = \mu\mathcal{U}\eta$, then $\eta \in t_1$ and, since ψ is the smallest formula where t_1 and t_2 disagree, also $\eta \in t_2$; hence, $\psi \in t_2$;
- if $\psi = \mu\mathcal{R}\eta$, then $\mu \in t_1$ and, since ψ is the smallest formula where t_1 and t_2 disagree, also $\mu \in t_2$; hence, $\psi \in t_2$;

Hence, no such formula ψ can exist. □

Soundness.

Theorem 2.17. *For every PLTL formula φ there exists a measuring automaton $A_\varphi = (\Sigma, \Omega, Q, q_0, X, \theta, \delta, \gamma, F, \omega)$ with an exponential number of states Q in $|\varphi|$ and a linear number of variables X in $|\varphi|$ such that for every sequence $\sigma \in (2^{AP})^*$, σ is accepted by A_φ with result r iff r is the measure of φ on σ under the unambiguous semantics.*

Proof. The correctness proof of the construction of A_φ in Theorem 2.17 follows the structure of the proof of Theorem 2.11 (the corresponding construction for deterministic PLTL). The key difference is in the proof of Lemma 2.9, where we claim that for every trace σ and formula φ, if $(\sigma, 0, r) \models \varphi$, then there exists an atom sequence that satisfies the successor relation.

For deterministic PLTL, this sequence can be constructed in a simple induction, progressing from the first position forwards, because the semantics are deterministic, i.e., the subformulas of the successor atom are uniquely determined by the present atom and the next event.

For unambiguous PLTL, the parametric subformulas are chosen based on the truth value of the non-parametric subformulas. We therefore construct the sequence of atoms in two steps. In the first step, we compute, progressing backwards from the final position, precisely the set of non-parametric formulas that are satisfied in each position. In the second step, we add, progressing forwards from the initial position, the parametric subformulas according to the (now deterministic) semantics of unambiguous PLTL.

From Theorem 2.17 and Lemma 2.12 it follows that the space required by the online monitor is exponential in the size of the specification and logarithmic in the length of the trace. □

Corollary 2.18. *Under the unambiguous semantics, a PLTL formula φ can be measured in exponential space in the size of the specification φ and logarithmic space in the length of the trace.*

2.7 Monitoring Algorithm

For online monitoring, we first construct the measuring automaton for the given PLTL formula. Then, we process the incoming events starting in the initial states, following the transition relation and eliminating sets of states which contain unsatisfiable sets of formulas.

To represent the current state of the automaton, the algorithm maintains in case of deterministic PLTL only a single, universally interpreted atom. In case of unambiguous PLTL, it maintains a set of universally interpreted atoms. The counters, which are kept per atom, are represented in a binary encoding.

For deterministic PLTL, the online monitoring algorithm (Algorithm 1) needs space linear in the specification φ, due to the maintenance of the current atom and the counters, and logarithmic in the length of the trace σ. The time complexity is linear in σ, and polynomial in φ, if the transition relation is computed on-the-fly.

The complexity of the online monitoring algorithm for unambiguous PLTL (Algorithm 2) is now mainly dominated by the specification, since it needs to maintain a universally interpreted set of atoms. For space complexity, it is exponential in formula φ (due to the need for explicitly maintaining a large number of counters), and again logarithmic in the length of the trace σ due to the size of the counters. With regard to time complexity, the algorithm is still constant time per event in respect to σ, and exponential in φ for the on-the-fly calculation of δ and γ.

Algorithm 1 Online Monitoring Algorithm for deterministic PLTL

```
Input: specification: PLTL^det $\varphi$, deterministic measuring automaton $A_\varphi$,
trace $\sigma$ with incoming events $e \in 2^{AP}$
Output: (preliminary) verdict $v$, measure $r$
1:  $currAtom \leftarrow q_0$                                                  ▷ initial state
2:  $currMeas \leftarrow \eta_0, \eta_0 = \{\Theta(n_x) \mid x \in X\} \cup \{\Theta(m_x) \mid x \in X\}$   ▷ initial counter values
3:  $pv \leftarrow$ true                                                       ▷ initial verdict
4:  for all new incoming events $e$ do
5:      $nextAtom \leftarrow \delta(currAtom, e)$                              ▷ evaluate transition relation
6:      $nextMeas \leftarrow \gamma(currMeas, currAtom, e)$                    ▷ update counters
7:      $currAtom \leftarrow nextAtom$
8:      $currMeas \leftarrow nextMeas$
9:      if $currAtom \in F$ then                                               ▷ evaluate preliminary verdict
10:         $pv \leftarrow$ true
11:     else
12:         $pv \leftarrow$ false
13:     end if
14: end for
15: $v \leftarrow pv$                                                          ▷ set final verdict
16: $r \leftarrow \omega(currAtom, currMeas)$                                   ▷ final evaluation of counters
```

2.8 Experiments

The algorithms from the previous section have been implemented in Java for both deterministic and unambiguous PLTL, together with an implementation for the offline algorithm described in section 2.2. For evaluation purposes, a Boolean abstraction of simulated circuit traces was used. The experiments were executed on an 2.6 GhZ Intel Core i7 processor with 8GB memory. The traces were generated as circuit simulation runs and stored on a solid-state disk drive. For the online monitor, they were given eventwise to the implementation.

Bus arbiter For this benchmark, traces from a bus arbiter for a shared resource serving three clients were used. The parameters are used to measure the waiting times of the clients with the following PLTL formula:

$$\Box(r_0 \rightarrow \Diamond_{\leq x_0} g_0) \wedge \Box(r_1 \rightarrow \Diamond_{\leq x_1} g_1) \wedge \Box(r_2 \rightarrow \Diamond_{\leq x_2} g_2)$$

Memory controller The second benchmark is an implementation of a memory controller module. The controller exports a bus interface to a memory module and reads and writes memory cell contents onto the bus. Here, we measured the re-

Algorithm 2 Online Monitoring Algorithm for unambiguous PLTL

```
Input: specification: unambiguous PLTL formula $\varphi$, measuring automaton $A_\varphi$,
trace $\sigma$ with incoming events $e \in 2^{AP}$
Output: (preliminary) verdict $v$, measure $r$
1: $currAtomList \leftarrow [q_0]$                                        ▷ initial state
2: $currMeasList \leftarrow [\eta_0], \eta_0 = \{\Theta(n_x) \mid x \in X\} \cup \{\Theta(m_x) \mid x \in X\}$    ▷ initial counter values
3: $pv \leftarrow$ true                                                    ▷ initial verdict
4: for all new incoming events $e$ do
5:     $nextAtomList \leftarrow []$
6:     $nextMeasList \leftarrow []$
7:     for $(t, \eta) \in zip(currAtomList, cMeasList)$ do
8:         $nextAtomList \leftarrow nextAtomList.append(\delta(t, e))$        ▷ $\delta$ is not deterministic
9:         $nextMeasList \leftarrow nextMeasList.append(\gamma(\eta, t, e))$
10:                                          ▷ $\gamma$ lifted to lists of counter valuations
11:     end for
12:     $currAtomList \leftarrow nextAtomList$
13:     $currMeasList \leftarrow nextMeasList$
14:     if $\exists\, currAtom \in currAtomList \wedge currAtom \cap F \neq \emptyset$ then
15:                                                    ▷ evaluate preliminary verdict
16:         $pv \leftarrow$ true
17:     else
18:         $pv \leftarrow$ false
19:     end if
20: end for
21: $v \leftarrow pv$                                                       ▷ set final verdict
22: $r \leftarrow \omega(currAtomList, currMeasList)$                  ▷ final evaluation of counters
```

tention period of a memory cell over a trace. The PLTL specification consisted of a single $\mathcal{U}_{\leq x}$ formula.

The performance differences highlighted in Table 2.8 are mainly a result of the different I/O behavior of the algorithms. Since the offline algorithm has to perform multiple passes over the trace, it reads the input file multiple times from disc. Since the online algorithm just maintains the reachable set of atoms at any given point in time, it does not have to explore the full state space of the corresponding measuring automata.

	bus arbiter		memory controller	
trace length	offline	online	offline	online
10k events	2027 ms	74 ms	444 ms	84 ms
100k events	6679 ms	263 ms	752 ms	246 ms
1M events	63711 ms	1484 ms	6275 ms	727 ms
10M events	180281 ms	13642 ms	65209 ms	15606 ms

Table 2.1: Runtime (in ms) for the offline and online monitoring algorithms on both benchmarks.

3 Stream Monitoring

This chapter provides the necessary background and important notions for stream-based monitoring with Lola as described in [24]. As an addition, a PLTL embedding is described in Section 3.4.

Classic monitoring approaches, as presented in Chapter 2, work on a Boolean abstraction of the system trace, mainly due to the historical roots of the underlying logics, and their use in hardware model checking. There, the Boolean abstraction ensures an overall finite state behavior of the system model, and thus makes model checking feasible because large, possibly infinite value domains of variables are abstracted away. In runtime monitoring, we are however not necessarily limited to Boolean abstractions. Since we are observing the runtime behavior as a single trace, it is easy to extract the current value of variables in the system under observation. Therefore, the barrier to use richer logics with expressive value domains is not immediately prohibitive.

The specification language Lola [24] was originally introduced for monitoring synchronous circuits, and it was inspired by developments in synchronous programming languages [41] like Esterel [18] and Lustre [20]. Its features include support for data types such as integers, floating point numbers and strings, as well as functions and future and past lookups of values. The underlying computation model is that of a stream transducer, which computes from a set of (typed) input streams a set of (typed) output streams. The verdict domain of such a monitor can be much richer than a Boolean result, as Boolean output streams can be used to mimic classic temporal logic monitoring, but output streams with numeric types allow the computation of aggregations and statistics and trigger external alerts based on the computed information.

For stream-based monitoring, the computational model of the monitor is more complex. Instead of computing a single output for a full trace as for a logic, the stream specification defines a transducer, which transforms a set of (typed) input streams to a set of (typed) output streams.

Since both input and output streams are typed, with types not necessarily restricted to Boolean, the output streams of a stream-based monitor can be interpreted as verdicts, which contain all preliminary and the final verdict.

As an example, while PLTL monitoring could be used to compute partial statistics of traces, such as the maximal (or minimal) waiting time of arbiter clients, LOLA allows to express general statistics, such as the *average* waiting time of a client, by giving direct access to the counters that a PLTL monitor keeps implicit, as shown in Listing 1.

```
input bool request
input bool grant
output int wait := if !request[-1,true] & request { 0 }
        else { wait[-1,0]+1 }
output int num_grants := if grant & wait > 0 { num_grants[-1,0] + 1 }
        else { num_grants[-1,0] }
output int sum_wait := if grant & wait > 0 { sum_wait[-1,0] + wait }
        else { sum_wait[-1,0] }
output double avg := sum_wait / num_grants
```

Listing 1: A LOLA specification tracking the average waiting time

One distinctive feature of LOLA in comparison to rule-based monitoring methods such as Eagle [6] and RuleR [7], another type of specification formalism with more expressivity than temporal logics, is the decoupling of the specification language (the declaration), from the monitoring algorithm (the evaluation). This allows for multiple backends: the same specification can be monitored using a hardware circuit, compiled into a high-level programming language or just fed to an interpreter.

The expressivity of LOLA is not just a result of the richer type system, as a formal complexity and expressivity analysis for the case of just Boolean types shows in [19], but also due to its computational model.

As illustrated by Listing 1, one major difference of stream-based specifications in comparison to temporal logics is the explicit representation of intermediate results, which allows compositional specifications. The output stream `avg` uses the intermediate results computed in `num_grants` and `sum_wait` to compute the final statistic.

3.1 Classic LOLA

As a simple example, a specification which counts the number of times an input proposition `in` is **true** can be expressed in LOLA as follows:

```
input bool in
output int out := if in { out[-1,0] + 1 } else { out[-1,0] }
```

Listing 2: A simple Lola specification

In the first line, we explicitly declare the input stream (`input`), its type (`bool`) and declare its name (`in`). On the second line, we define the output stream `out` of type `int` and give a stream expression to define its value. Within the expression, we use a conditional to test whether the stream `in` is **true** in the current position, and then use the stream access operator `out[-1,0]` to access the value of `out` in the previous position (-1) of the stream and – depending on the value of `in` – either increment or leave the value of `out` as before.

To illustrate the evaluation of this specification, consider the following example trace:

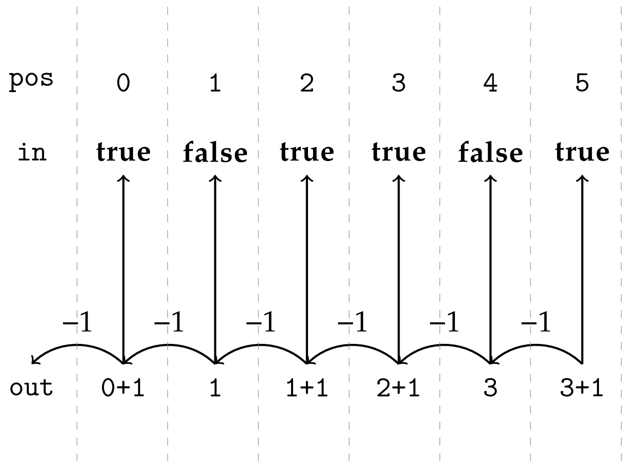


Figure 3.1: Example input trace and evaluation of the output stream `out`

In order to evaluate `out` at position 0, we first evaluate the conditional by looking at `in` at position 0. Since it has value **true**, we evaluate the positive case, and determine the value of `out[-1,0]` at position 0. Since the stream offset looks beyond the beginning of the stream, it evaluates to the out of bounds value, 0. Therefore, the value of `out` is 1 in the first position. The other positions are evaluated accordingly, except for the past offsets to `out`, which now evaluate to the value of `out` in the previous position.

3.1.1 Syntax

A Lola specification is declared as a system of equations, which is expressed as a set of typed stream variables and their corresponding stream expressions. We will denote stream types by capital letters (T), stream variables by lowercase letters (t), and expressions by lowercase letters (e) or (φ) for Boolean expressions. Three types of streams exist: *input* streams, *output* streams, and *trigger* streams.

$$\begin{aligned}
\textbf{input}\ & T_1\ t_1 \\
& \vdots \\
\textbf{input}\ & T_m\ t_m \\
\textbf{output}\ & T_{m+1}\ s_1 := e_1(t_1, \dots, t_m, s_1, \dots, s_n) \\
& \vdots \\
\textbf{output}\ & T_{m+n}\ s_n := e_n(t_1, \dots, t_m, s_1, \dots, s_n) \\
\textbf{trigger}\ & \varphi_1, \dots, \varphi_k
\end{aligned}$$

Note that only output and trigger streams have corresponding stream expressions. The stream expressions $e_i(t_1, \dots, t_m, s_1, \dots, s_n)$ are defined over the set of independent (input) stream variables $t_1, \dots, t_m$ and dependent (output) stream variables $s_1, \dots, s_n$. All stream variables are typed with their associated types T_i. Trigger expressions $\varphi_1, \dots, \varphi_k$ are by definition typed as Boolean.

Stream Expressions A stream expression $e(t_1, \dots, t_m, s_1, \dots, s_n)$ of type T is defined recursively from the following elements:

- *Constants*: For a given constant c of type T, $e = c$ is an atomic stream expression of type T.
- *Stream Variables*: For a stream variable s of type T, $e = s$ is an atomic stream expression of type T.
- *Functions*: For a k-ary function $f : T_1 \times \cdots \times T_k \to T$ and stream expressions $e_1, \dots, e_k$ of matching types $T_1, \dots, T_k$, $e = f(e_1, \dots, e_k)$ is a stream expression of type T.
- *Conditionals*: For a stream expression b of type `bool`, stream expressions e_1 and e_2 of type T, $e = \textbf{ite}(b, e_1, e_2)$ is a stream expression of type T.
- *Stream Offsets*: For a stream variable s with corresponding type T, a default

value d of type T, and a stream offset i of type `int`, $e = s[i,d]$ is a stream expression of type T.

The type system supports basic types such as `bool`, `int`, `double`, and `string`. Basic logical connectives and arithmetic operators are used with their usual semantics and can easily be defined within the syntax as function expressions.

3.1.2 Semantics

The semantics of a Lola specification on a specific input trace are defined via its corresponding *evaluation model*. Note that in general, a Lola specification and a trace may have no corresponding evaluation model, one evaluation model, or even many evaluation models.

Let Φ be a Lola specification with input stream variables $t_1, \dots t_m$ of types $T_1, \dots, T_m$ and output stream variables $s_1, \dots, s_n$ of types $T_{m+1}, \dots, T_{m+n}$. Let N denote the length of the input traces, whose values are denoted by finite subtraces τ_i for $1 \leq i \leq m$. The value of input stream i at position j is denoted by $\tau_i(j)$, where $0 \leq j < N$.

An *evaluation model* of Φ on τ is a tuple $\Gamma = \langle \tau_1 \dots \tau_m, \sigma_{m+1} \dots \sigma_{m+n} \rangle$ of typed streams of length N, such that for all streams s_i, their stream expressions e_i, and all positions j, the values of σ_i match the evaluation function $\mathrm{val}(e_i)(j)$, defined recursively over the structure of e_i as follows:

- $\mathrm{val}(c)(j) = c$ (constant expressions)
- $\mathrm{val}(t_h)(j) = \tau_h(j)$ (stream variables)
- $\mathrm{val}(f(e_1, \dots, e_h))(j) = f(\mathrm{val}(e_1)(j), \dots, \mathrm{val}(e_h)(j))$ (function application)
- $\mathrm{val}(\mathbf{ite}(b, e_1, e_2))(j) = \begin{cases} \mathrm{val}(e_1)(j) & \text{if } \mathrm{val}(b)(j) = \mathbf{true} \\ \mathrm{val}(e_2)(j) & \text{else} \end{cases}$ (conditional expression)
- $\mathrm{val}(s_h[k,d])(j) = \begin{cases} \mathrm{val}(s_h)(j+k) & \text{if } 0 \leq j+k < N \\ d & \text{if otherwise} \end{cases}$ (stream offsets)

3.1.3 Properties

We now recapitulate important semantic and syntactic properties of Lola specifications.

Well-defined specifications A Lola specification is *well-defined* [24], if for any set of appropriately-typed same-length input streams, it has a single, unique evaluation model. One example specification which does not have the property is the following:

```
output bool a := !a
```

In this specification, stream a accesses itself with an offset of 0 and negates the result. It does not have an evaluation model. A second example specification which has more than a single evaluation model, is the following:

```
output int c := c
```

The evaluation models for this specification allow every possible integer value for c in every position. While for these specifications the problems are easy to spot, they can easily be made more complex:

```
output int a := b
output int b := a
output int c := d[1,0]
output int d := c[-1,0]
```

Listing 3: A Lola specification with circular references

Observe that the well-definedness property of Lola specifications is a *semantic* property of the specification. In the worst case, we would need to explore all possible evaluation models of a specification, which is expensive to check for pure Boolean streams as the problem is in EXPTIME and PSPACE-hard [19], and undecidable in general due to the rich type model for streams.

Well-formed specifications Since we need an efficient way to determine whether a Lola specification has a unique evaluation model, the notion of *well-formed* specifications was introduced in [24]. Well-formedness represents a more restrictive, but syntactic criterion, based on the dependency graph of a Lola specification. It over-approximates well-definedness: Every well-formed specification is well-defined, but not vice versa.

The *dependency graph* of a Lola specification is a weighted and directed multi-graph, where the vertices represent stream variables, the edges represent stream accesses within stream expressions, and the edge weights represent temporal offsets. Given a Lola specification Φ, its dependency graph is defined as follows:

$G_\Phi = \langle V, E\rangle$, where $V = \{s_1, \ldots, s_n, t_1, \ldots t_m\}$. For the edge set E, we add an edge $s_i \xrightarrow{w} s_j$ to E whenever there is an access in the stream expression e_i to s_j with offset w.

The dependency graph of the specification in Listing 3 is shown in Figure 3.2.

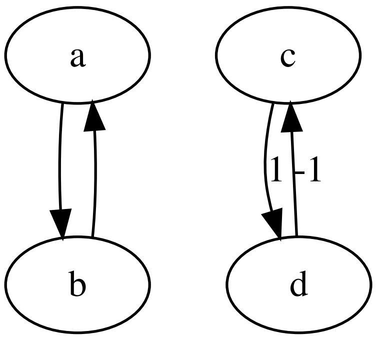


Figure 3.2: Dependency graph of the specification in Listing 3.

A *cycle* in the graph G_Φ is a sequence $v_0 \xrightarrow{e_0, w_0} v_1 \ldots v_{k-1} \xrightarrow{e_k, w_k} v_k$, such that for all i, $v_i \in V$, $e_i = v_i \xrightarrow{w_i} v_{i+1} \in E$, and $v_0 = v_k$. The *weight* of a cycle is the sum of its weights w. If $w = 0$, we call the corresponding cycle zero-weight.

A Lola specification is *well-formed* iff its dependency graph does not contain zero-weight cycles. As can be seen in the example of 3.2, the cycle formed by the streams *c-d-c* has zero-weight, therefore the specification is not well-formed.

3.2 Efficiently monitorable specifications

The possible use of future dependencies in stream access expressions within Lola may lead to high memory requirements for an online monitor.

```
input bool in
output bool eventually := in | eventually[+1,false]
```

Listing 4: A Lola specification checking the eventually-operator with unbounded future.

An example specification can be seen in Listing 4, where the usual semantic definition of the LTL-operator *eventually* was directly translated to Lola. Here, the memory requirement for an online monitor grows linearly with the input trace, if

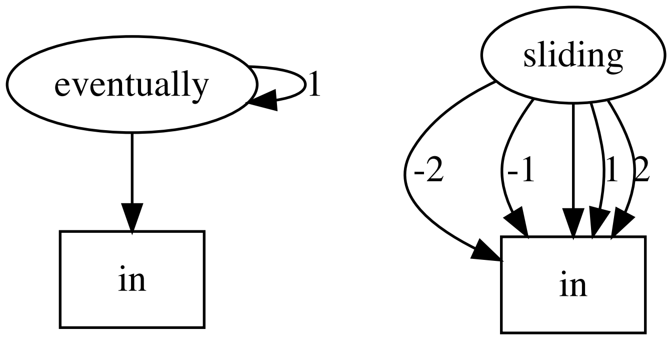


Figure 3.3: Dependency graphs of the specifications in Listing 4 (left) and Listing 5 (right).

the stream `in` only becomes **true** in the last position of the trace, since it needs to maintain the full stack of unresolved stream expressions at every position before, and can only back-propagate the result once it arrives at the last position.

The main issue is that Lola in general allows *unbounded* dependencies on the future value of streams. From the point of view of memory requirements of an online monitor, however, not all use of future dependencies leads to a problem.

```
input int in
output double sliding =
                (in[-2,0] + in[-1,0] + in + in[+1,0] + in[+2,0]) / 5.0
```

Listing 5: A Lola specification computing a five-step sliding average.

The example specification for a sliding average described in Listing 5 contains future references, but an online monitoring algorithm only has to keep five values or partially evaluated stream expressions in its memory.

The difference between the two specifications can be visualized from their dependency graphs, seen in Figure 3.3. While the left dependency graph and specification contains a positive cycle on the `eventually`-stream and therefore looks unbounded into the future, the right dependency graph and specification does not contain a cycle and only has bounded lookahead into the the future.

This gives us the following characterization of efficient monitorability: A Lola specification is *efficiently monitorable* iff its dependency graph does not contain any positive-weight cycles. As we will see in the online monitoring algorithm in the next section, this property of the specification is essential to bound the memory resources of the monitor.

For the eventually-specification of Listing 4, we can state the property in a different way as a LOLA specification without future lookups, as described in Listing 6.

```
input bool in
output bool eventually := eventually[-1,false] | in
```

Listing 6: A LOLA specification checking the eventually-operator in a past version.

Note that the specifications produce different outputs, as the first variant will set the output stream (after delaying until the first position where `in` becomes **true**) to **true** in every position, whereas the second (past) variant will continually output **false** until the first position where `in` becomes **true**.

3.3 Online Monitoring Algorithm

The online monitoring algorithm maintains an equation store with two sets of equations R (resolved) and U (unresolved). The equation store with R and U contains instantiated stream expressions for every position. U contains partially resolved stream expressions, where not all stream accesses have been resolved yet. Once a stream expression has been fully evaluated, it is moved to the set R, which contains only constant expressions (i.e. values). Both sets are indexed by stream variable and position. The lookup-table GC for garbage collection holds the offset vectors which determine when values are cleared from set R. An example equation store with R and U is illustrated in Figure 3.4.

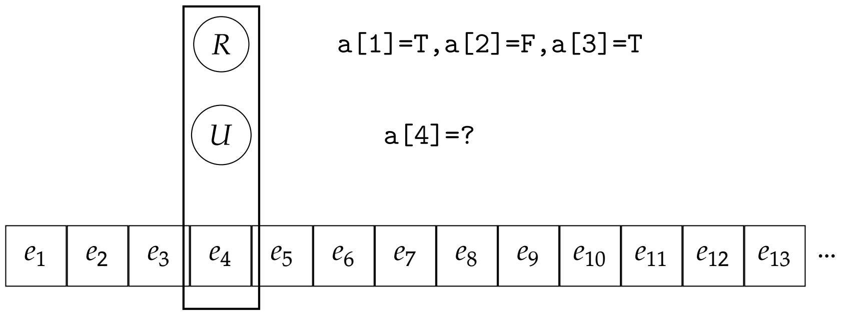

R
a[1]=T,a[2]=F,a[3]=T
U
a[4]=?
e_1 e_2 e_3 e_4 e_5 e_6 e_7 e_8 e_9 e_{10} e_{11} e_{12} e_{13} ...

Figure 3.4: Illustration of the equation store for classic LOLA. The equations with future accesses to stream `a` are not yet resolved for position 4. Within the current monitor step, `a[4]` can be resolved and moved to the set R.

Algorithm 3 Online Monitoring Algorithm for Lola

```
    Input: Incoming Event $E$
    Output: List of Triggers $T$
 1: pos ← pos + 1                                   ▷ Increment current position
 2: for all $i \in E$ do                            ▷ Add input events to R
 3:     $R \leftarrow R \cup \{i\}$
 4: end for
 5: while fixpoint $U$ do
 6:     for all $e_j \in U$ do
 7:         simplify $e_j$                          ▷ Evaluate expression
 8:         if $e_j$ is constant then
 9:             $U \leftarrow U \setminus \{e_j\}$
10:             $R \leftarrow R \cup \{e_j\}$       ▷ Add to resolved equations
11:             check if $e_j$ activates a trigger; add to $T$
12:         end if
13:     end for
14: end while
15: for all $e \in R$ do                            ▷ Perform garbage collection
16:     if GC(e) > pos then
17:         $R \leftarrow R \setminus \{e\}$        ▷ Compare to Offset
18:     end if
19: end for
20: return active triggers $T$
```

The performance of the algorithm is mainly determined by the order of expression evaluation in line 6, which can be optimized before runtime by analysis of the specification.

3.3.1 Time and Memory Requirements

For Algorithm 3, the time complexity is polynomial in the size of the specification and linear in the trace. The space requirement of the algorithm in terms of the specification is again polynomial. For the memory dependency on the trace, we can derive a precise upper bound only for efficiently monitorable specifications, which equals the maximal size of the sets R and U, which in turn is determined by the maximal GC-distance of any stream times the size of the Lola specification. This memory bound only depends on the specification, therefore the online monitoring problem for Lola for efficiently monitorable specifications is constant modulo datatypes[1] in the length of the trace .

[1] Assuming constant storage requirements for the stored stream values in their respective datatypes.

3.4 Embedding PLTL in Lola

To embed a deterministic or unambiguous PLTL formula as a Lola specification, we can utilize the structure of the corresponding measuring automaton and implement its transition relation and counter updates using Lola streams. For a $\mathrm{PLTL}^{\mathsf{det}}$ formula, it is sufficient to represent the current state of the automaton by maintaining a representation of a single atom via its subformulas. For unambiguous PLTL, the construction needs to be lifted to sets of atoms.

For a $\mathrm{PLTL}^{\mathsf{det}}$ formula φ over 2^{AP} with corresponding deterministic measuring automaton $A_\varphi = (\Sigma, \Omega, Q, q_0, \Theta, \delta, \gamma, F, \omega)$, we introduce the following Lola streams:

- Per $p \in AP$: a Boolean input stream,
- Per subformula of $\mathrm{cl}(\varphi)$: a Boolean output stream,
- Per parametric subformula of $\mathrm{cl}(\varphi)$: two integer output streams to maintain counters n_x and m_x, and a Boolean output stream to implement the generated-predicate.

The transition relation δ can then be implemented via the stream expressions of the Boolean output streams of the subformulas. The update function γ can be directly realized through the stream expressions of the integer output streams.

The verdict v is the last value of the Boolean output stream of the top-level formula φ itself. The measure r can be evaluated identically to ω by using the last values of the integer output streams for the counters.

The partial statistics and datatypes supported by Lola are sufficient to implement the counters and counter updates of the measuring automaton, as only increments, case distinctions and minimum and maximum operations are needed.

Example We will translate the $\mathrm{PLTL}^{\mathsf{det}}$-formula $\varphi = \Box(\neg r \wedge \top) \vee (r \wedge \Diamond_{\leq x} g)$ to a Lola specification. The closure of φ contains the following elements: $\mathrm{cl}(\varphi) = \{r, g, \neg r, \Diamond_{\leq x} g, r \wedge \Diamond_{\leq x} g, (\neg r \wedge \top) \vee (r \wedge \Diamond_{\leq x} g), \varphi\}$. The Lola specification is shown in Listing 7.

Note that the size and number of streams of the corresponding Lola specification for a $\mathrm{PLTL}^{\mathsf{det}}$ formula is linear in the size of the formula. For unambiguous PLTL, we have to implement the full exponential construction for the atoms, since we need to maintain a set of counters per atom.

Limits The presented classic variant of Lola offers no direct support for parameterization. Still, it is expressive enough to express parametric logics such as PLTL, as

```
input bool r
input bool g

output bool neg_r := ! r
output bool pfinally_g := pfinally_g[-1,false] | g
output bool r_and_pfinally_g := r & pfinally_g
output bool neg_r_or_r_and_pfinally_g := neg_r | r_and_pfinally_g
output bool phi := phi[-1,true] & neg_r_or_r_and_pfinally_g

output bool generated_pfinally_g := neg_r[-1,false] & r
output int n_x := ite(generated_pfinally_g, 0,
                      ite(g, 0, n_x[-1,0]+1))
output int m_x := ite(g, max(n_x,m_x[-1,0]), m_x[-1,inf])
```

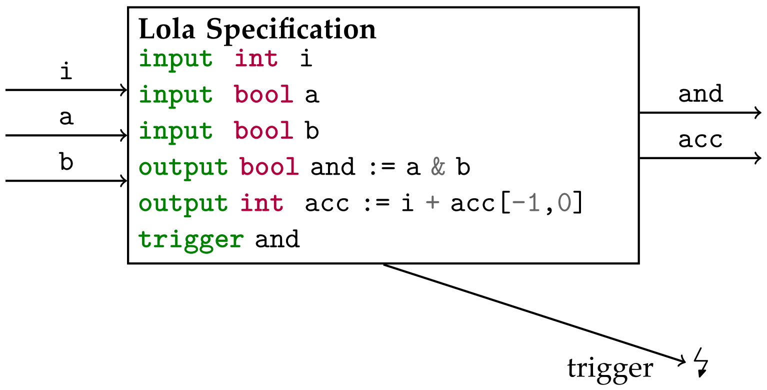
Listing 7: A Lola specification implementing a monitor for the PLTL$^{\mathsf{det}}$-formula φ.

we have demonstrated in this chapter. With regards to parameterization, the main drawback of the monitoring approach of classic Lola is that everything parametric needs to be explicitly encoded into the specification, which leads to a significant increase of the size of the specification, and that there is no language support to specify the behavior of substreams of the input data in a direct manner.

4 Stream Monitoring with Parametric Data

A fundamental challenge in runtime monitoring and in the development of monitoring specifications is the handling of data in the event stream of a system. Allowing local specifications of substreams, automatic reconstruction of their context, and efficiently handling the local monitors in the online monitoring algorithm are the key issues handled within the extension of LOLA to handle parametric data that we present in this chapter.

Within the field of runtime monitoring, there have been multiple approaches to deal with parametric data in specifications. One such approach in the context of rule-based specification languages was in the systems EAGLE, described in [42, 6], and RULER, described in [7]. In [4], Alur et. al. proposed a quantitative extension of regular expressions, QRE, to handle the specification of parametric data in event streams. For automata-based approaches, quantified event automata were introduced in [5] and contain a similar type of quantification over data.



```
input  int  i
input  bool a
input  bool b
output bool and := a & b
output int  acc := i + acc[-1,0]
trigger and
```

Figure 4.1: Model of classic LOLA Stream Monitoring

To demonstrate the introduction of parameters and data into a specification language like LOLA, we first give an educational example. Suppose we would like to monitor an authentication log of a system with many users. The log will record successful and unsuccessful authentication attempts. Due to mistyped passwords and other user mistakes, we may regularly see unsuccessful attempts in the log, which are usually followed by a password reset or a successful authentication attempt for that particular user. Across all users, for any given time interval, we may see a significant number of failed authentication attempts, which makes it hard to specify a generic alert threshold without considering the individual users. Still, detecting failed authentication attempts for individual users is useful to stop directed password-guessing attacks.

A common mitigation to prevent guessing attacks is to perform rate-limiting on the number of authentication attempts which an individual user may perform successively. A common property associated to this mitigation would be to trigger a lockdown for a user account with more than three consecutive failed login attempts.

If there is an a-priori known, finite number of users in our system, we can monitor the property with a logic-based approach: We simply instantiate an appropriately large set of atomic propositions ($\text{failed}_{user1}, \text{success}_{user1}, \ldots$), and then express the property as following in LTL for a user u:

$$\varphi_u = \neg \Diamond(\text{failed}_u \wedge \bigcirc(\neg\text{success}_u \mathcal{U}(\text{failed}_u \wedge \bigcirc(\neg\text{success}_u \mathcal{U} \text{failed}_u))))$$

This approach has the following drawbacks:

- the system log output needs to be pre-processed to give Boolean valuations to the indexed atomic propositions,
- the number of users needs to be bounded at design-time of the specification,
- the encoding results in a large number of atomic propositions,
- the resulting specification is large and grows proportionally to to the number of users,
- it does not scale to infinite domains such as strings (for user identifiers).

A similar approach would be possible in the classic version of Lola, where one could introduce one output stream per possible user identifier. The problem, however, is that one needs to explicitly reconstruct the context per user to obtain the current number of failed login attempts, as can be seen in Listing 4.1, where the outer conditional needs to check whether the current input event is relevant to the stream, and can only then test the inner conditional for the intended property. The context is implemented by the outer conditional, which on the negative case just carries the previous value over.

```
input   bool loginSuccess
input   int  uid
output  int  attempts_1 := ite(uid = 1,
                           ite(loginSuccess, 0, attempts_1[-1,0] + 1),
                               attempts_1[-1,0])
trigger attempts_1 > 3

output  int  attempts_2 := ite(uid = 2,
                           ite(loginSuccess, 0, attempts_2[-1,0] + 1),
                               attempts_2[-1,0])
trigger attempts_2 > 3
…
```

Listing 4.1: Login attempts mitigation unrolled as a classic Lola specification.

An example input trace is displayed in Figure 4.2. Here, we highlight two relevant slices of the input trace for the user identifiers `uid` with values 1 and 2 are highlighted. This visualization emphasizes the need to treat each user on its individual *slice* and its individual *time scale*.

```
input  bool   loginSuccess
input  string uid
output int    attempts<string user>
     invoke: uid
     extend: uid=user
     := if loginSuccess then 0 else attempts(user)[-1,0]+1
output bool   bruteforce<string user>
     invoke: uid
     extend: uid=user
     := attempts(user) > 3
trigger any(bruteforce)
```

Listing 4.2: Login attempts mitigation as a Lola 2.0 specification.

Both features have native language support in Lola 2.0, as demonstrated in Listing 4.2: The data parameterization of the output stream `attempts` with the input stream `uid` creates a data slice via the inline `invoke` stream `uid`. Whenever the monitor sees a fresh value for `uid` in the input, it *invokes* a new output stream instance for the stream `attempts` with the parameter bound to the new fresh `uid`

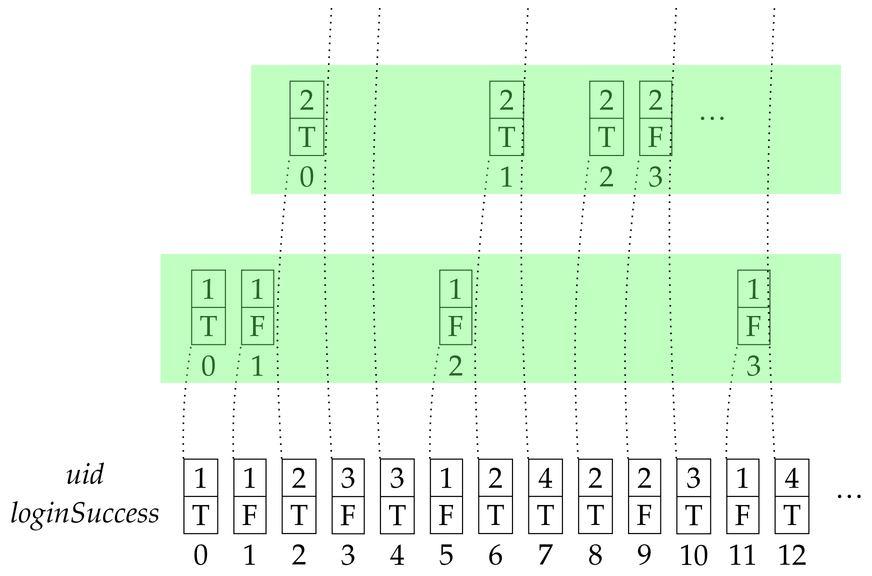
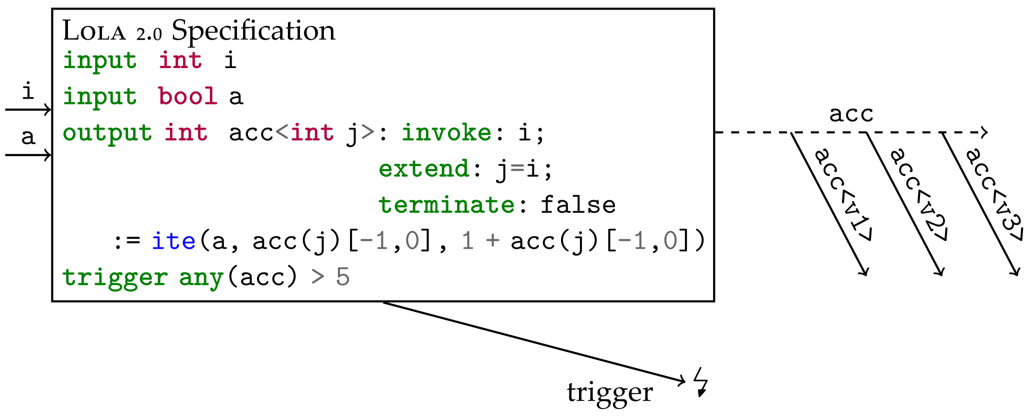


Figure 4.2: Visualization of an login attempts example trace. The trace has two input streams, the user identifier `uid` as an integer and the Boolean login indication `loginSuccess`.

value. The local time scale of each data slice is then constructed via the inline `extend` stream predicate (`uid = user`), which *extends* all those output stream instances for which the stream results in the value **true**. Note that stream offset expressions such as `attempts(user)[-1,0]` are now interpreted on the local time scale of the stream instance, as controlled by the extend stream. Finally, the trigger expression `any(bruteforce)` tests whether any of the `bruteforce` output streams has value **true**. Optionally, an auxiliary `terminate` stream can be specified, to define when to *terminate* an output stream instance, as a reset mechanism. If a stream instance has been terminated, it can be freshly invoked again later via a new invoke.

4.1 Parameterized Stream Monitoring

The primary extension to the work in [24] is the notion of *template output streams*. The parameters and respective types of the template output streams are syntactically described after the stream variable name. Then these parameters can be accessed by name in the stream expression. The binding of the parameters in the stream instances is performed via its auxiliary `invoke`, the semantics of the stream offset references is controlled via its auxiliary (Boolean) `extend` stream. The auxiliary (Boolean) `terminate` stream is used to allow the termination of a stream instance, which may be used to re-bind with the same parameters in a later position and to control memory usage in the monitor.



```
input  int  i
input  bool a
output int  acc<int j>: invoke: i;
                        extend: j=i;
                        terminate: false
   := ite(a, acc(j)[-1,0], 1 + acc(j)[-1,0])
trigger any(acc) > 5
```

Figure 4.3: Model of LOLA 2.0 Stream Monitoring.

4.1.1 Syntax

A (classic) LOLA specification, as described in Chapter 3, is defined as a system of equations, which is expressed as a set of typed stream variables and their corresponding stream expressions. For the syntax of LOLA specifications, we refer to subsection 3.1.1.

Parametric Output Streams Output stream *templates* are output streams equipped with a set of parameters. These parameters which may be instantiated to output stream *instances*. Formally, an output stream template is defined as:

$$\begin{aligned}
&\textbf{output}\ T\ s\langle p_1 : T_1, \dots, p_l : T_l\rangle \\
&\qquad\textbf{invoke:}\ s_{inv} \\
&\qquad\textbf{extend:}\ s_{ext} \\
&\qquad\textbf{terminate:}\ s_{ter} \\
&\qquad := e(t_1, \dots, t_m, s_1, \dots, s_n, p_1, \dots, p_l)
\end{aligned}$$

Observe that $\langle p_1 : T_1, \dots, p_l : T_n\rangle$ denotes the *declaration* of the parameters $p_1, \dots, p_l$, while $(p_1, \dots, p_l)$ is a *using occurrence* of the parameters bound in the stream declaration. As syntactic sugar, the auxiliary streams may be declared inline, as seen in

Figure 4.3.

The auxiliary streams s_{inv}, s_{ext}, s_{ter} control the lifetime and the clock of the associated stream instances. For a stream equation template without parameters, we omit the empty parameter set $\langle\rangle$. Specifiying the invoke, extend, and terminate streams of the stream equation template is optional. The default stream expression for invoke is the empty tuple (), the default stream expression for extend is `true`, and the default stream expression for terminate is `false`, which allows downward-compatibility with LOLA, as it matches the behavior of classic LOLA streams, which are never terminated and extended in every position.

Stream Expressions A stream expression $e(t_1, \dots, t_m, s_1, \dots, s_n, p_1, \dots, p_n)$ is defined recursively from the following elements:

- For a parameter p of type T, $e = p$ is an atomic stream expression of type T.
- For a constant c of type T, $e = c$ is an atomic stream expression of type T.
- For a k-ary function $f : T_1 \times \dots \times T_k \to T$ and stream expressions $e_1, \dots, e_k$ of matching types $T_1, \dots, T_k$, $e = f(e_1, \dots, e_k)$ is a stream expression of type T.
- For a stream expression b of type `bool`, stream expressions e_1 and e_2 of type T, $e = \mathbf{ite}(b, e_1, e_2)$ is a stream expression of type T.
- For a stream variable s with associated type T, parameters $p_1, \dots, p_n$, a constant d of type T, and a constant i of type `int`, $e = s(p_1, \dots, p_n)[i, d]$ is a stream expression of type T.
- For a template stream variable s and an aggregation operator Op of type T, $e = Op(s)$ is a stream expression of type T. Examples of aggregation operators are `any` of type `bool`, and `count` of type `int`.

The type system supports basic types such as `bool`, `int`, `double`, and `string`, as well as tuples of the basic types such as `(bool,int)`.

Classic LOLA specifications can be expressed with stream templates by specifying an empty set of parameters and using the aforementioned defaults for the auxiliary stream $s_{inv}, s_{ext}, s_{ter}$. This ensures that syntactically, every classic LOLA specification is also a LOLA 2.0 specification.

4.1.2 Semantics

The semantics of a LOLA 2.0 specification are defined via its corresponding evaluation model on a given trace. Note that without appropriate restrictions, a LOLA specification in general may have either no evaluation model, a single evaluation model, or many evaluation models.

Stream Templates, Instances, and Auxiliary Streams A stream template introduces a stream template variable s of type T with declared parameters $p_1, \dots, p_l$ of types $T_1, \dots, T_l$. For a given parameter valuation $\alpha = (v_1, \dots, v_l)$ of matching types $T_1, \dots, T_l$, the instance of s under α is defined as:

$$s(v_1, \dots, v_l) = e(t_1, \dots, t_m, s_1, \dots, s_n, p_1, \dots, p_n)[p_1/v_1, \dots, p_l/v_l].$$

The expression s_{inv} is the *invocation* stream variable of s and has type $T_1 \times \cdots \times T_l$. Intuitively, if s_{inv} evaluates to a fresh value $(v_1, \dots, v_l)$, a new instance $s\langle v_1, \dots, v_l\rangle$ is invoked.

The expression s_{ext} is the *extension* stream variable of s and has type $T_1 \times \cdots \times T_l \rightarrow$ `bool`. If s is invoked with $\alpha = (v_1, \dots, v_l)$, an extension stream s^{α}_{ext} is invoked with the corresponding parameter valuation. Intuitively, whenever s^{α}_{ext} evaluates to **true**, then $s\langle v_1, \dots, v_l\rangle$ is extended in this position.

The expression s_{ter} is the *termination* stream variable of s and has type $T_1 \times \cdots \times T_l \rightarrow$ `bool`. If s is invoked with $\alpha = (v_1, \dots, v_l)$, a termination stream s^{α}_{ter} is invoked with the corresponding parameter valuation. Intuitively, whenever s^{α}_{ter} evaluates to **true**, then $s\langle v_1, \dots, v_l\rangle$ is terminated.

If a stream instance is terminated and extended at the same time, it does not have a value at that position.

Formal Semantics Let Φ be a Lola 2.0 specification with independent (input) stream variables $t_1, \dots t_m$ of type $T_1, \dots, T_m$ and dependent (output) template stream variables $s_1, \dots, s_n$ of type $T_{m+1}, \dots, T_{m+n}$. Let N denote the length of the input trace, whose values are denoted by finite subtraces τ_i for $1 \leq i \leq m$. The value of input stream i at position j is denoted by $\tau_i(j)$.

An *evaluation model* of Φ on the input streams τ is a set Γ of stream instances σ of length N with types $T_1, \dots, T_{m+n}$. We extend each type T_i with the symbol $\perp$, to indicate stream positions where a stream instance has not been invoked yet, is not extended, or has already terminated (and has not been invoked again). We use s^{α}_i to denote the stream instance of template stream s_i with parameter valuation α and σ^{α}_i to refer to the corresponding stream instance in Γ.

We impose the following conditions on evaluation models in order to ensure that the set of stream instances in Γ contains all necessary instances and the values in the stream instances σ are evaluated accordingly:

- A designated constant stream σ_0 is in Γ and produces the empty tuple () in every position $1 \ldots N$. This stream serves as an invocation stream for all streams without parameters.
- For each template stream variable s_i, the stream instances are determined by its associated invocation stream s_{inv} (we omit i from the subscript where the reference is clear). If Γ contains some stream σ_{inv}, then Γ must also contain a stream for every instance of s_i invoked by σ_{inv} at some position; i.e. for all $j < N$ where $\sigma_{inv}(j) = \alpha$, $\sigma_i^\alpha \in \Gamma$.[1]

To define the lifetime of stream instances, we use the predicate $alive(s_i, (v_1, \ldots, v_l), j)$, which evaluates to **true** at a position j if there is an earlier position $j' \leq j$ where the instance of s_i with parameter valuation $\alpha = (v_1, \ldots, v_l)$ was invoked, i.e. $\sigma_{inv}(j') = \alpha$, the stream was not terminated in the meantime, i.e. for all j'', $j' \leq j'' \leq j$, $\sigma^\alpha_{ter}(j'') = \textbf{false}$.

For each stream that is *alive* in a position and extended in the same position, its value is determined by evaluating the corresponding stream expression.

For each stream instance $\sigma_i^\alpha \in \Gamma$ with some parameter valuation $\alpha = (v_1, \ldots, v_l)$ of template stream s_i,

$$\sigma_i^\alpha(j) = \begin{cases} \mathrm{val}(e_i[p_1/v_1, \ldots, p_l/v_l])(j) & \text{if } alive(s_i, (v_1, \ldots, v_l), j) \text{ and } \sigma^\alpha_{ext}(j) = \textbf{true} \\ \bot & \text{otherwise} \end{cases}$$

where the evaluation function $\mathrm{val}(e_i[p_1/v_1, \ldots, p_l/v_l])(j)$ is defined recursively over e_i as following:

- constant expressions:
  $\mathrm{val}(c)(j) = c$
- input streams:
  $\mathrm{val}(t_h)(j) = \tau_h(j)$ for $1 \leq h \leq m$
- function application:
  $\mathrm{val}(f(e_1, \ldots, e_h))(j) = f(\mathrm{val}(e_1)(j), \ldots, \mathrm{val}(e_h)(j))$
- conditional expression:
  $$\mathrm{val}(\textbf{ite}(b, e_1, e_2))(j) = \begin{cases} \mathrm{val}(e_1)(j) & \text{if } \mathrm{val}(b)(j) = \textbf{true} \\ \mathrm{val}(e_2)(j) & \text{else} \end{cases}$$

[1] Note that in general, s_{inv} may itself also be parameterized (the only exception being the designated constant stream σ_0), which is omitted here to simplify the presentation.

- stream access (current position):
$$\mathrm{val}(s_h(p_1,\dots,p_n)[0,d])(j) = \begin{cases} \sigma_h^{(p_1,\dots,p_n)}(j) & \text{if } \mathit{alive}(s_h,(p_1,\dots,p_n),j) \\ d & \text{otherwise} \end{cases}$$

- stream access with offset:
$\mathrm{val}(s_h(p_1,\dots,p_n)[k,d])(j) =$
$$\begin{cases} d & \text{if } j \geq N \text{ or } j < 0 \text{ or } \neg \mathit{alive}(s_h,\alpha',j) \\ \mathrm{val}(s_h(\alpha')[k-1,d])(j+1) & \text{if } k > 0, \sigma_{ext}^{\alpha'}(j) = \mathbf{true}, \mathit{alive}(s_h,\alpha',j) \\ \mathrm{val}(s_h(\alpha')[k+1,d])(j-1) & \text{if } k < 0, \sigma_{ext}^{\alpha'}(j) = \mathbf{true}, \mathit{alive}(s_h,\alpha',j) \\ \mathrm{val}(s_h(\alpha')[k,d])(j+1) & \text{if } k > 0, \sigma_{ext}^{\alpha'}(j) = \mathbf{false}, \mathit{alive}(s_h,\alpha',j) \\ \mathrm{val}(s_h(\alpha')[k,d])(j-1) & \text{if } k < 0, \sigma_{ext}^{\alpha'}(j) = \mathbf{false}, \mathit{alive}(s_h,\alpha',j) \end{cases}$$
for $\alpha' = (p_1,\dots,p_n)$.

For stream accesses to instances with parameters, note that the parameters need to have matching types for the substitution, but they may be swapped or replaced by constant values.

The auxiliary extend stream to every stream instance serves as a local clock to the stream instance, as illustrated in Figure 4.2 (with $\perp$ positions omitted).

4.1.3 Fragments and Properties

Dependency Graph To facilitate syntactic checks and properties on Lola 2.0 specifications, we extend the notion of the *dependency graph* of the specification. The dependency graph for our login attempts specification is depicted in Figure 4.4. Intuitively, the vertices of the graph represent streams, and the edges represent stream accesses from the corresponding stream expressions. There are additional stream accesses for the auxiliary invoke, extend, and terminate streams of the template stream variables.

Formally, a dependency graph of a Lola 2.0 specification Φ with input stream variables $t_1 \dots t_m$, output template stream variables $s_1 \dots s_n$, is a directed multi-graph $G_\Phi = (V, E)$ defined as:

- $V = \{t_1,\dots,t_m\} \cup \{s_1,\dots,s_n\}$

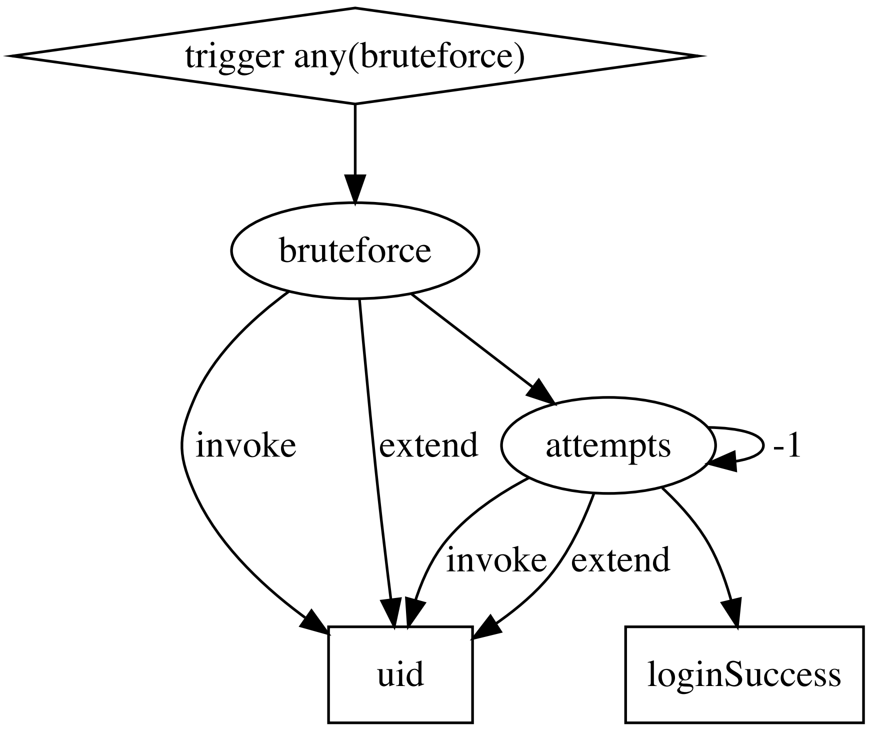


Figure 4.4: Dependency Graph of Login Specification shown in Figure 4.2. Trigger streams are illustrated as diamond shapes, input streams have rectangular shape, and output streams are round-shaped. Every edge corresponds to a stream access, and if the offset is unequal to 0, the edge is labeled with it.

- $E : V \times \{+, -, 0\} \times V$,

$$\begin{aligned} E = &\{(v, +, v') \mid v' \in \text{accesses}(v) \text{ with positive offset}\} \\ &\cup\{(v, -, v') \mid v' \in \text{accesses}(v) \text{ with negative offset}\} \\ &\cup\{(v, 0, v') \mid v' \in \text{accesses}(v) \text{ without offset}\} \end{aligned}$$

  where accesses(v) is the set of stream accesses in the stream expression and auxiliary streams of v.

Thus, for every stream access to v' in the stream expression of v with offset k, the multi-graph contains an edge from v to v' annotated with the sign $a \in \{+, -, 0\}$ of the offset k or 0. A *cycle* in a directed multi-graph $G(V, E)$ is a finite sequence of vertices $v_0, \dots, v_n$ such that $v_0 = v_n$, $\forall i. v_i \in V$, and $\forall i. \exists a_i. (v_i, a_i, v_{i+1}) \in E$.

Well-formed specifications A LOLA 2.0 specification Φ is *well-formed* iff its dependency graph G_Φ does not contain a cycle with the following properties:

- all annotations a_i are 0, or
- there exists at least one + and at least one − annotation.

The stronger requirements for well-formedness of LOLA 2.0-specifications are due to the possibly different pace of extend-clocks in the specification.

Efficiently monitorable specifications For classic LOLA, recall that a LOLA specification Φ is *efficiently monitorable*, whenever every value of every stream is resolved within a finite number of steps and it does not contain unbounded lookups into the future. This definition excludes specifications such as the following:

```
output bool a = a[1,true]
```

In order to determine the value of stream a at the first position in an online monitor, we have to wait for the input stream to terminate, because the stream access looks one step into the future, and the default value of the stream access only evaluates to **true** in the last position of the trace. The monitor only then may propagate this value backwards step-by-step to evaluate the unresolved stream expressions of the earlier positions.

This specification property can again be determined by inspecting the cycles of its dependency graph; If the dependency graph does not contain positive cycles, the specification is guaranteed to be efficiently monitorable.

Compared to the classic definition, LOLA 2.0 potentially aggravates the problem: if streams with unbounded future lookup are used as invocation streams, we not only have to back-propagate a finite number of values, but we may have to retroactively instantiate new stream instances and extend different sets of stream instances along the way, which may effectively lead to a full monitor reset with respect to e.g. count trigger expressions. Even bounded future lookups to streams which are not extended in every step are a potential problem, because the next extension of the stream may be delayed indefinitely.

Therefore, we significantly extend the definition of efficient monitorability for LOLA 2.0 specifications and disallow future dependencies within all auxiliary streams and their dependencies. While bounded future lookups were acceptable in LOLA, the aforementioned issues force a stricter definition, since full monitor resets would lead to unpredictable online monitor execution time per incoming event. Also, this greatly simplifies the monitoring algorithm, as the set of stream instances which needs to be evaluated can be determined with the available past and current values of other stream instances.

4.2 Fixpoint-based online monitoring algorithm

For our *online* monitoring algorithm, we restrict the trace to being available one event at a time, and we have no a-priori bound on the length of the trace. Output stream instances and triggers shall be evaluated as soon as possible.

The main data structure of our online monitoring algorithm is an *equation store*, which keeps track of the state of the evaluation of stream instances and their expressions. The algorithm maintains the following sets:

- a set of currently *active* stream instances S,
- a set of *resolved* equations R,
- a set of *unresolved* equations U,
- and a set of offset vectors for *garbage collection* of R called GC.

Intuitively, for every step of the monitor, the set S determines which stream instance expressions are to be added initially to the set U. Then, until sets S, R, and U become stable, the monitor simplifies equations in U, moving them to R if they are fully evaluated, checks for new invocations, extensions, and terminations of stream instances with effect on S (and potentially U), and finally removes resolved equations from R whenever the set GC determines they are no longer needed.

More formally, the algorithm performs the following steps per position (starting at position $j = 0$):

1. For each input stream t_i, add its value $\tau_i(j)$ to R.
2. Add the designated default invocation stream value $\sigma_0(j) = ()$ to R.
3. Initialize the set $S(\_, j)$ of active stream instances: For all output stream templates s_i and valuations α such that $\alpha \in S(s_i, j-1)$ (α was previously active), if $\sigma^{\alpha}_{\text{ter}}(j) = \mathbf{false}$, add α to $S(s_i, j)$.

Then repeat the following steps until a fixpoint on the sets S and R is reached:

1. Evaluate and simplify equations in U as much as possible. Once they are evaluated to a constant value, add them to R.
2. Check for new invocations, extensions and terminations by the additions to R.
3. If for any output stream template s_i the corresponding invocation stream yields a new valuation, i.e. $\sigma^{\alpha}_{i,\text{inv}}(j) = \beta$ is added to R, then $S(s_i, j) = S(s_i, j) \cup \beta$ and we add $\sigma^{\beta}_{i}(j) = e^{\beta}(j)$ to U.
4. If for any output stream template s_i, the corresponding extension stream evaluates to **true**, i.e. $\sigma^{\alpha}_{i,\text{ext}}(j) = \textbf{true}$ is added to R, then add $\sigma^{\alpha}_{i}(j) = e^{\alpha}(j)$ to U.

For the unresolved equations stored in U, we apply the following evaluation steps:

- Function application: e.g. $0 + x \to x$, $\textbf{true} \wedge \textbf{false} \to \textbf{false}$, …
- Rewriting conditional expressions: $\textbf{ite}(\textbf{true}, e_1, e_2) \to e_1$, $\textbf{ite}(\textbf{false}, e_1, e_2) \to e_2$
- Resolve stream accesses (to stream s_i at current position): If $\sigma^{\alpha}_{i}(j) = c$ in R, then substitute $\sigma^{\alpha}_{i}(j)$ by c in every right hand side U.
- Resolve stream accesses with offsets: If some $\sigma_{l,\alpha}(j) = e_l$ in U contains a subexpression $\sigma_{i,\alpha}(j)[k, d]$, $\sigma^{\alpha}_{\text{ext}}(j) = c$ is in R with either $c = \textbf{true}$ or $c = \textbf{false}$, and $\sigma^{\alpha}_{\text{ter}}(j) = \textbf{false}$ then

$$\sigma^{\alpha}_{i}(j)[k,d] \to \begin{cases} \sigma^{\alpha}_{i}(j) & \text{if } k = 0, \sigma^{\alpha}_{\text{ext}}(j) = \textbf{true} \\ \sigma^{\alpha}_{i}(j+1)[k-1,d] & \text{if } k > 0, \sigma^{\alpha}_{\text{ext}}(j) = \textbf{true} \\ \sigma^{\alpha}_{i}(j-1)[k+1,d] & \text{if } k < 0, \sigma^{\alpha}_{\text{ext}}(j) = \textbf{true} \\ \sigma^{\alpha}_{i}(j+1)[k,d] & \text{if } k > 0, \sigma^{\alpha}_{\text{ext}}(j) = \textbf{false} \\ \sigma^{\alpha}_{i}(j-1)[k,d] & \text{if } k < 0, j > 0, \sigma^{\alpha}_{\text{ext}}(j) = \textbf{false} \\ d & \text{otherwise} \end{cases}$$

- Resolve stream accesses to inactive streams: If some $\sigma^{\alpha}_{l}(j) = e_l$ in U contains a subexpression $\sigma^{\alpha}_{i}(j)[k, d]$, and $\alpha \notin S(s_i, j)$, then $\sigma^{\alpha}_{i}(j)[k, d] \to d$.

After each event has been processed, we utilize the set GC to remove entries from the set R which are no longer needed. For each template output stream, we therefore initially calculate its cutoff vector, which determines for how many steps resolved equations remain in R. The initial value per stream can again be determined by the dependency graph of the specification: for each template output stream, the maximal negative-weight path over all dependent streams is the longest possible usage of the determined value. Due to the potentially slow moving pace of individual stream instances, we keep in GC a vector per stream instance, which on invocation is set to the initial vector value 0, and then incremented every time the stream instance is extended. Once the stream instance vector reaches the cutoff vector value for its template output stream, we can safely remove the instance equation from R. The stream instance vectors have to maintained dynamically along the stream per stream instance and cannot be predicted before.

Once a stream instance has terminated either via its auxiliary termination stream or because the input stream has ended, we replace all open offset expressions looking beyond the final position with the default value and rerun the fixpoint steps once more.

4.2.1 Memory Requirements

An efficiently-monitorable classic Lola specification needs polynomial space in the size of the specification Φ and constant space in the length of the trace σ in the case of online monitoring. The space dependency on the trace again is given modulo space required for the datatypes and values.

For a Lola 2.0 specification, the same is not true in general: If an output stream has for example a parameter of type string, an unbounded number of stream instances may exist. Thus, the amount of memory needed along the trace grows linear with the length of the trace.

In practical applications, however, a bound on the number of instances can be established as part of the design considerations of a monitor and can be monitored at runtime. The event rate of the system can also be established reasonably at design time. Then, one could either pre-allocate memory up to the instance bound and use sufficiently large memory for numeric types or incrementally use the memory along the trace. Either way, a safe estimate can be found at design time and the overall memory requirement can be bounded for a specific Lola 2.0 specification.

This memory requirement is especially important for hardware realizations with or without external memory.

4.3 Efficient implementation

Depending on the use case for LOLA 2.0, different memory vs. computation time trade-offs can be made in the implementation of the algorithm of the last section. The performance of the online monitor depends mainly on the size of the set S of maintained active stream instances. If this set is small, the main part of the work will be spent in the evaluation and simplification of stream equations in the set U, in step 1 of the fixpoint. These evaluations can be cached or pre-computed if necessary, especially with respect to the impact of Boolean inputs on further evaluation steps. Specifications may also allow to determine a linearization on the evaluation order without the need for a fixpoint, if the dependency graph forms a tree.

If the set S is large, the performance will be dominated by the steps 2 – 4 of the fixpoint. If the monitor has to keep a large set of instances, but only a small number of equations will be touched in the evaluation process, then efficient lookup structures from parameter valuations to the corresponding stream instances and the efficient evaluation of trigger expressions on parametric output streams are key to the performance. The set of dependencies of input streams can again be pre-computed and does not need to be evaluated on-the-fly in every step.

4.4 Experiments

We have implemented LOLA 2.0 in a prototype command-line tool in C with support for inverse index structures from inputs to stream instances. The target application was network monitoring, where a (passive) network monitor observes edge traffic and reports suspicious IP addresses to a separate firewall component.

Web application Fingerprinting The LOLA 2.0 specification shown in Listing 8 detects web application fingerprinting (i.e. the identification of server-side software and software version) attacks via an indirect method. It relies on the observation that such fingerprinting attacks often yield a large number of server-side responses with HTTP status codes 404 (“not found”) or 405 (“method not allowed”), since they essentially blindly probe the request URL, as illustrated in Figure 4.5. While some HTTP 4xx responses are perfectly normal due to mis-typed URLs, outdated links, or programming errors, we detect whether a single host issues a large number of requests resulting in such response codes. Therefore, the counters of malformed requests are parameterized by destination (since we detect responses in the outgoing traffic) IP address. We also parameterize by source IP address, to account for

```
input string Protocol, Response, Src, Dest
output (string, string) badHttpRequestInvoke:
  ext: Protocol="HTTP" & Response="Bad Request" | "Not Found"
    := (Src, Dest)
output bool badHttpRequestExtend<src, dst>:
  inv: badHttpRequestInv
    := src=Src & dst=Dest & Response = "Bad Request" | "Not Found"
output bool webAppFingerprintingTerminate<src,dst>:
  inv: badHttpRequestInv
    := src=Src & dst=Dest & Response = "OK"
output int webAppFingerprinting<src, dst>:
  inv: badHttpRequestInvoke;
  ext: badHttpRequestExtend;
  ter: webAppFingerprintingTerminate
    := webApplicationFingerprinting(src, dst)[-1,0] + 1
trigger any(webAppFingerprinting > threshold)
```

Listing 8: Web Application Fingerprinting Specification

multiple HTTP servers on the network.

The specification first matches on the HTTP protocol and the relevant statuses, and then starts the appropriate counting stream instances. The stream instances are terminated and the counters therefore reset, if the client manages to form a normal request with response code 200 (“OK”).

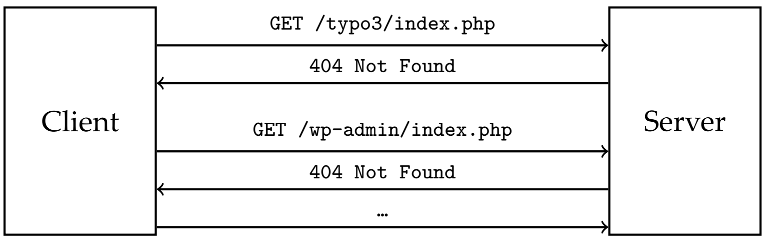


Figure 4.5: Illustration of the network traffic for web application fingerprinting.

TCP SYN Scans The specification shown in Listing 9 detects TCP SYN Scan attacks. In those attacks, an attacker on the network tries to induce resource exhaustion and ultimately denial of service (DoS) on a server by opening a larger number of connections to it and keeping them open. This traffic is illustrated in Figure 4.6. Since the root case of the problem is the necessary memory allocations in the TCP stack of the server operating system, one may want to block misbehaving hosts from flooding the server further. Therefore, a network monitor on the edge of network may detect such hosts and alleviate the memory pressure by blocking the source IP address on a firewall.

Such attacks are usually detected by inspecting network trace files with tools like Snort [21]. Since those tools are not able to slice the data based on source and destination address of packets and their properties are usually stateless, this results in many false positives, since sometimes connections drop to many users at once because of congestion and other problems en route. Our solution is to detect such TCP SYN scans by behavioural detection: we keep a statistic on open connections per source and destination IP address pair and detect whether in a short time period, we see many open TCP handshakes between individual pairs.

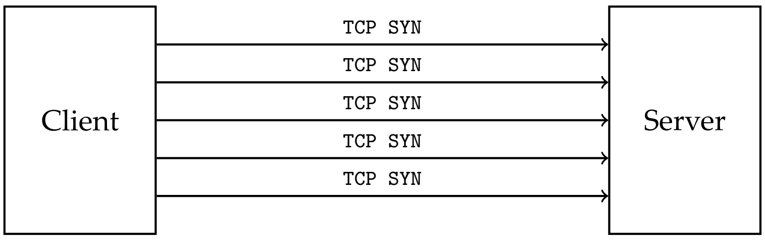


Figure 4.6: Illustration of the network traffic for TCP SYN Scan attacks.

In the specification from Listing 9, this detection is implemented in two phases: in the first phase, realized via the parametric output stream `waitForAck`, we detect single incomplete TCP Handshakes, which set the SYN bit and never set the ACK bit. These are terminated once the ACK bit is set.

In the second phase, we count further open SYNs from the same host pair via the parametric output stream `tcpSynScan`. Then, we set two triggers: one *count* trigger, which triggers when the number of concurrent incomplete handshakes reaches a certain threshold over all host pairs, and one *any* trigger, which triggers when a specific host pair has more than threshold open connections.

The multi-phase approach in this specification helps the specification designer, no need to establish single formula describing the behavior.

The trace files for the experiments with this specification were obtained from the Malware Capture Facility Project[2] and preprocessed to CSV-files via Wireshark[3]. The results are summarized in Table 4.7. All experiments were run on a single quad-core machine with an 3.6GHz Intel Xeon processor with 32 GB RAM. The input trace files were stored on an internal SSD drive.

[2]http://mcfp.weebly.com
[3]http://www.wireshark.org

#Packets	Snort alerts	Time (sec)	# Invocations		Count	Any
			Wait	Scan	Trigger	Trigger
901710	613	2550.31	53654	662	660	11
1710373	472	6279.87	95983	521	519	31
1857753	1699	6786.06	107721	550	548	31
1954427	2428	7160.27	115787	695	693	31
2419607	2036	10347.96	146748	1535	1533	31

Figure 4.7: Experimental results for network monitoring of TCP SYN Flood Attacks. Snort specification for comparison was set to 100 TCP Requests in 60 seconds. Trace files from Malware Capture Facility Project. Runtime: 8-30 minutes, Memory: 20-50MB. Snort runtime: less than 2 minutes.

Analysis and Discussion The runtime results show a slow-down due to large number of stream instances, even though only one of the instances is extended in every event. This could possibly be improved by more efficient data structure for stream instance lookups. For a pure software implementation, it can handle a large amount of open instances.

The bounded instantiation of the stream instances is guaranteed here by the fixed bitwidth of source and destination IP address, and thus special data structure support for IP addresses would further help the experimental performance.

From practical perspective, the amount of trigger alerts would result in significantly less manual inspection of attacks compared to the Snort solution. Data parameterization keeps the specifications succinct and the implementation works for reasonably-sized examples in practice.

```
const int threshold1 := 100
const int threshold2 := 2
const int threshold3 := 500

input string Protocol, Syn, Ack, Source, Destination

output bool  extend_incompleteHandshakeInvoke <>
        :inv: unit
        :ext: true
        :ter: false
        := Protocol="TCP" & Syn="Set" & Ack="Not set"

output (string, string) incompleteHandshakeInvoke <>
        :inv: unit
        :ext: extend_incompleteHandshakeInvoke()[0, false]
        :ter: false
  := (Source, Destination)

output bool incompleteHandshakeTerminate <string src, string dest>
        :inv: incompleteHandshakeInvoke
        :ext: true
        :ter: incompleteHandshakeTerminate(src, dest)[0, false]
  := Source=src & Destination=dest & Syn="Not set" & Ack="Set"

output int waitForAck <string src, string dst>
        :inv: incompleteHandshakeInvoke
        :ext: true
        :ter: incompleteHandshakeTerminate(src, dst)[0, false]
  := waitForAck(src, dst)[-1, 0] + 1


output bool term_ext_tcpSynInvoke <string src, string dst>
        :inv: incompleteHandshakeInvoke
        :ext: true
        :ter: incompleteHandshakeTerminate(src, dst)[0, false]
  := waitForAck(src, dst)[0, 0]  > threshold1

output (string, string) tcpSynInvoke <string src, string dst>
        :inv: incompleteHandshakeInvoke
        :ext: term_ext_tcpSynInvoke(src, dst)[0, false]
        :ter: incompleteHandshakeTerminate(src, dst)[0, false]
  := (src, dst)

output bool tcpSynExtend <string src, string dst>
        :inv: tcpSynInvoke
        :ext: true
        :ter: tcpSynTerminate(src, dst)[0, false]
  := src=Source & dst=Destination & Syn="Set"

output bool tcpSynTerminate <string src, string dst>
        :inv: tcpSynInvoke
        :ext: true
        :ter: tcpSynTerminate(src, dst)[0, false]
  := src=Source & dst=Destination & Syn="Not set" & Ack="Set"

output int tcpSynScan <string src, string dst>
        :inv: tcpSynInvoke
        :ext: tcpSynExtend(src, dst)[0, false]
        :ter: tcpSynTerminate(src, dst)[0, false]
  := tcpSynScan(src, dst)[-1, 0] + 1

trigger count(tcpSynScan) > threshold2
trigger any(tcpSynScan > threshold3)
```

Listing 9: Network Monitoring Specification to detect SYN-Scan attacks.

5 Real-time Stream Monitoring

While support for parametric data was added to Lola in Chapter 4, the underlying computation model was still *synchronous*: the monitor only performs a new round of stream evaluation whenever a new input event is received, and time advances in discrete steps. This is perfectly fine for systems with a reliable timing source for both system and monitor, like hardware circuits, for which Lola was originally developed in [24], and time-triggered embedded systems as demonstrated in [1]. For example, this enables to check if a time limit is exceeded by counting the number of steps.

On the other hand, *asynchronous* systems are systems which are driven by external events, without a central, reliable timing source. The *event rate* at which the system can receive inputs and produce outputs may vary during the execution of the system and monitor, and only an upper bound may be known at design time.

This timing model introduces significant challenges for any monitoring approach:

- For safety-critical embedded applications, the monitor needs a reliable timing source for itself, and needs to run independently of the system to be able to act as a watchdog,
- and notwithstanding the variable input rate, the monitor has to process incoming events with bounded resources.

Here, we introduce our monitoring approach RTLola, originally introduced in [35] and illustrated in Figure 5.1. In RTLola, we meet these challenges as follows:

- We add native support for real-time references and sliding real-time windows with aggregation functions to the syntax and semantics of Lola 2.0,
- we realize an online monitor for the resulting language by combining two timing models, with an input-synchronous part working on the pace of the input event rate and a time-triggered second part,
- we take an output-based view to realize the time-triggered second part of the monitor: a monitor frequency, to be fixed at design time, decouples the eval-

uation of output streams from the input event rate, and only evaluates output streams and sliding windows when at the monitor frequency.

For the important class of efficiently monitorable RTLola specifications with efficient aggregation functions, this output-based view enables a static specification analysis to compute the *bounded* memory requirement for a particular specification. Furthermore, we show that an implementation of RTLola, which uses stream instantiation as well as a real-time sliding windows, is also useful for offline data analysis tasks and can handle realistic input traces and specifications.

For specification languages in runtime monitoring, real-time properties have been a subject of study for a long time, beginning with real-time variants of temporal logics [54]. One of the main approaches [27] uses signal temporal logic (STL) over real-valued signals [28]. Another used a metric variant of first-order temporal properties in [9], [10] and the tool Aerial [12]. Sliding windows for real-time aggregations were also used in [11] and the efficient evaluation of sliding windows with panes was described in [52]. Practical approaches for embedded and low-level hardware systems were introduced in Copilot [56] and TeSSLA [50], respectively. For asynchronous stream runtime verification, Striver [40] presents a low-level solution without parametrization.

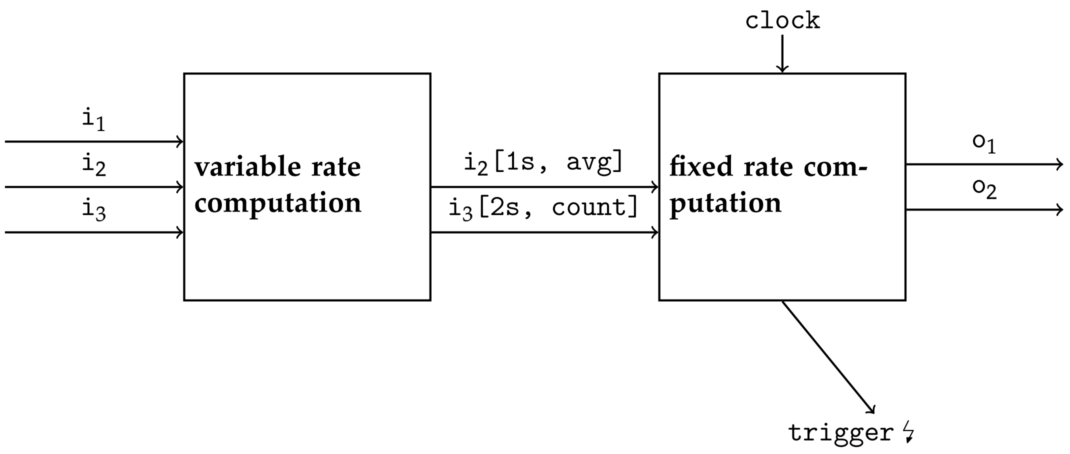


Figure 5.1: Model of Real-time Lola Stream Monitoring. Input Streams i_1,i_2 and i_3 are received by the variable rate (input-triggered) part of the monitor, whereas the trigger and output streams o_1 and o_2 are produced by fixed rate (time-triggered) monitor driven by `clock`. The streams in the time-triggered part access input streams via window expressions, in this case accessing i_2 via a one second average and accessing the number of values of i_3 in a two-second period.

An example for a RTLola-specification can be seen in Listing 5.1. Here we assume an input trace which contains a timestamp and a product identifier. The first output stream `sales_by_prod` splits the input trace into individual streams per product. The second output stream `sales_frequency` then computes a one hour window on those individual product streams and uses the count aggregation function to count the number of sale events. Finally, the `trigger` gives an alert if any product has been sold more than 100 times within the last hour.

```
frequency 0.1 Hz
timeinput double timestamp
input int productID

output bool sales_by_prod<int id>
    invoke: productID
    extend: id = productID
    := true

output int sales_frequency<int id>
    invoke: productID
    := sales_by_prod(id)[1h, count, 0]

trigger any(sales_frequency > 100)
```

Listing 5.1: Identifying frequently sold products on an online shopping platform.

5.1 Monitoring with Time

The two main additions within the syntax of the language are the extension of the stream offset operator and the real-time sliding window operator. Previously, the offset operator $s[k, d]$ allowed only negative and positive integer values for k to indicate the number of discrete steps into the past or future of the accessed stream s. Here, we extend the type of k to also allow real-time offsets with time units, where s is accessed at past offset k with a sample-and-hold semantics. Second, we introduce a new real-time sliding window operator $s[k, \gamma, d]$, where k is a real-time offset, γ an aggregation function, and d again a default value. This operator computes the aggregation function γ over the window of duration k on the stream s.

As illustrated in Figures 5.2 and 5.1, the main feature of RTLola is the decoupling of variable-rate input part of the monitor from the fixed-rate output part. To not un-

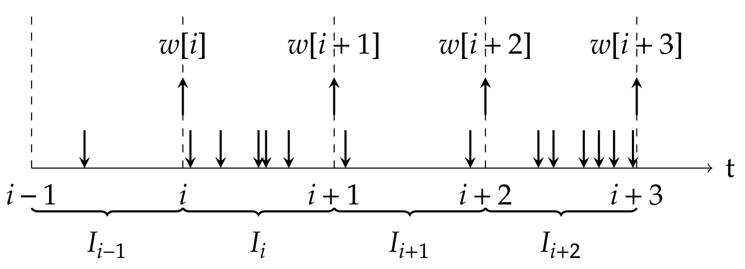


Figure 5.2: Timeline of input stream with incoming events marked as downward arrows. Partitioning of the time line into intervals of equal size assuming a constant rate extension stream for the accessing stream.

necessarily stress computational resources, not every arrival of an input event leads to a full recomputation of all output streams, but at the same time, the monitor shall not skip input events. The key addition to the language which enables this decoupling are the real-time sliding windows[1], which aggregate a variable number of input events with the aggregation function γ over the window size k. Depending on the fixed rate of the accessing stream, this leads to a fixed a number of intervals, where the accessing stream needs to be evaluated. This partitioning of the timeline is depicted in Figure 5.2, where a variable number of input events (downward arrows) is aggregated within the intervals marked by the dashed lines, which also represent the fixed, constant rate extension stream of the accessing stream.

The global time reference is either provided by an input stream with timestamps, or with respect to the local (real-time) clock of the monitor itself.

5.1.1 Syntax

The syntax is closely related to Lola 2.0 as described in Chapter 4, with the aforementioned extensions in the stream equations and clock handling. A specification consists of a system of typed stream equations over typed stream variables. As a first type of stream, input stream definitions are defined as before and only consist of a type T and a chosen stream identifier t, so the i-th input stream is defined as:

$$\textbf{input}\ T_i\ t_i$$

[1]Remark: The computation of the sliding windows in the way they are defined here is coupled to the updates of the input streams and the fixed computation rate of the output stream expression which uses them. A property "every window of 1h contains at least 100 events" is not expressible, since we would need to trigger an evaluation with 1h delay after every input event. This is not a fundamental limitation, as one could simply add a delay operator to the language to schedule such evaluations.

If the input stream contains an external time reference such as a timestamp for an offline monitor, it is indicated by the keyword **timeinput**. As a second type of stream, parameterized output stream templates are syntactically defined as following:

$$\begin{aligned}
&\textbf{output } T_{m+j}\ s_j\ \langle p_1 : T_j^1, \dots, p_k : T_j^k \rangle \\
&\qquad \textbf{invoke} : \mathit{inv}_j \\
&\qquad \textbf{extend} : \mathit{ext}_j \\
&\qquad \textbf{terminate} : \mathit{ter}_j \\
&\qquad := e_j(t_1, \dots, t_m, s_1, \dots, s_n, p_1, \dots, p_k)
\end{aligned}$$

This describes the j-th of n output stream templates. Each of the templates defines a template variable s_j of type T_{m+j}, which depends on the values of the parameters p_1 through p_k. An *instance* of s_j with valuation α of corresponding types $T_j^1 \times \cdots \times T_j^k$ can be defined by the substitution

$$s_j(v_1, \dots, v_k) = e_j(t_1, \dots, t_m, s_1, \dots, s_n, p_1, \dots, p_k)[p_1/v_1, \dots, p_k/v_k].$$

The lifetime of these stream instances is controlled by the template variables inv_j, ext_j, and ter_j, which are the auxiliary streams of s_j and have the following roles:

- The invocation template stream variable inv_j of type $T_j^1 \times \cdots \times T_j^k$ determines the creation of stream instances. Whenever inv_j evaluates to a fresh value $\alpha = (v_1, \dots, v_k)$, a new instance $s_j(\alpha)$ of s_j is invoked.
- The extension template stream variable ext_j of type *bool* is invoked - per stream instance - with the same parameter valuation α whenever a new instance is invoked and controls when the stream instance is extended. An instance $s_j(\alpha)$ is extended whenever its corresponding extension stream $\mathit{ext}_j(\alpha)$ evaluates to **true**. Then, the value of the instance $s_j(\alpha)$ is determined based on the corresponding stream expression $e_j(t_1, \dots, t_m, s_1, \dots, s_n, v_1, \dots, v_k)$.
- Similarly, the termination template stream variable ter_j of type *bool* is also invoked per stream instance with the same parameter valuation α whenever a new stream instance is invoked. An instance $s_j(\alpha)$ is terminated whenever its corresponding termination stream $\mathit{ter}_j(\alpha)$ evaluates to **true**.

If the output stream template does not have any parameters, the empty parameter declaration $\langle\rangle$ may be omitted. Also, if the invocation template stream is omitted, it defaults to the default invocation stream **unit**, which produces the empty tuple () in every position. If the extension template stream is omitted, it is set to the constant stream **true**, which produces a tick in every position, and if the termination template stream is omitted, it is set to the constant stream **false**, respectively.

Timing Model Every individual stream instance $s_j(\alpha)$ is paced by a corresponding local clock defined by its extension stream $ext_j(\alpha)$. These local clocks themselves advance on a global reference clock **gc**, which is extended whenever any new input event arrives for the monitor, and also serves as a clock for the default stream **unit**. Thus, for the monitor, time still evolves in discrete ticks by **gc**, but those ticks are not equidistant on the real-time axis. Instead, they are the finest granularity of observation. The type of **gc** is assumed to be the timestamp of the incoming event, either coming from a `timeinput` time reference or an internal clock signal.

The definition of ext_j may be omitted from the syntactical description of s_j, and the default semantics are that s_j will follow the union (disjunction) of the extensions streams of the output stream instances and input streams which appear in its expression. The extension streams of input streams are evaluated at the pace of **gc** and evaluate to **true** whenever a new value for this input stream is available. The pace of the termination stream ter_j is coupled to the extension stream.

Syntax of stream expressions An RTLola expression $e_j(t_1, \ldots, t_m, s_1, \ldots, s_n, p_1, \ldots, p_k)$ can be constructed recursively from the following sub-expressions:

- For a parameter p with type T, $e = p$ is a stream expression of type T.
- For a constant c of type T, $e = c$ is an atomic stream expression of type T.
- For a k-ary function $f : T_1 \times \cdots \times T_k \to T$ and stream expressions $e_1, \ldots, e_k$ of matching types $T_1, \ldots, T_k$, $e = f(e_1, \ldots, e_k)$ is a stream expression of type T.
- For a stream expression b of type `bool`, stream expressions e_1 and e_2 of type T, $e = \textbf{ite}(b, e_1, e_2)$ is a stream expression of type T.
- For a stream variable s with associated type T, parameters $p_1, \ldots, p_k$, a constant d of type T, and a constant i of type `int`, $e = s(p_1, \ldots, p_k)[i, d]$ is a stream expression of type T.
- For a stream variable s with associated type T, parameters $p_1, \ldots, p_k$, a constant d of type T, and a constant r (in seconds) of type `double`, $e = s(p_1, \ldots, p_k)[r, d]$ is a stream expression of type T.

- For a stream variable s with associated type T, parameters $p_1, \dots, p_k$, a constant d of type T', a constant r (in seconds) of type `double`, and any function γ from the theories with type $T^* \mapsto T'$, $e = s(p_1, \dots, p_k)[r, \gamma, d]$ is a stream expression of type T'.

- For a template stream variable s and an aggregation operator Op of type T, $e = Op(s)$ is a stream expression of type T.

Additionally, *RTLola*-specifications may contain a list of `trigger`-expressions to generate notifications: `trigger` $\varphi_1, \varphi_2, \dots, \varphi_k$, where $\varphi_1, \varphi_2, \dots, \varphi_k$ are arbitrary expressions of Boolean type.

5.1.2 Semantics

The semantics of an RTLola-specification on a given trace are defined again via its corresponding evaluation model. The semantics of the data parameterization correspond to *Lola 2.0*, but the asynchronous input behavior and the real-time (window) accesses are added.

For an RTLola-specification Φ with input streams $t_1, \dots, t_m$ of type $T_1, \dots, T_m$, output stream templates $s_1, \dots, s_n$ of types $T_{m+1}, \dots, T_{m+n}$ with their associated parameters $p_1^{s_i}, \dots, p_{l_i}^{s_i}$ of type $T_1^{s_i}, \dots, T_{l_i}^{s_i}$ (for the output stream template s_i with l_i parameters).

Let $N > 0$ denote the length of the global clock stream of the input trace, whose values are denoted by finite subtraces τ_i for $1 \leq i \leq m$. The value of input stream i at position j is denoted by $\tau_i(j)$. If a value of a stream (instance) is not available in a position, this is denoted by the default value $\perp$, which we add to all types T_i for this purpose.

An *evaluation model* for Φ on τ is a set Γ of streams of length N. An output stream instance of output template stream s_j with parameter valuation $\alpha = v_1, \dots, v_{l_i}$ is denoted by $s_j(\alpha)$, and its corresponding stream in Γ by σ_j^α.

Again, Γ has to be constructed such that all relevant stream instances are included and its stream equations evaluate to the correct values. The elements of Γ are determined as following:

- $\sigma_0^{()} \in \Gamma$, the designated initial stream, which evaluates to the empty tuple () in all positions and serves as the default invocation stream.

- For every output stream template s_j from $s_1, \dots, s_n$, we have to consider its associated invocation stream inv_j. If Γ contains a stream $\sigma_{inv_j}^\alpha$ for some α, then Γ must also contain a stream instance for each fresh value β produced by $\sigma_{inv_j}^\alpha$

at any position, i.e. for all positions $i < N$, if $\sigma^{\alpha}_{inv_j} \neq \bot$, and $\sigma^{\alpha}_{inv_j} = \beta$, then also $\sigma^{\beta}_{j} \in \Gamma$.

The main difference in the evaluation of stream expressions compared to Lola 2.0 is the handling of real-time offsets, real-time windows, and non-existent stream values. For streams, non-existent values accessed by stream offset operators will follow a sample-and-hold semantics.

It remains to define the value of a stream instance $s_j(\alpha)$ when it has been invoked and is currently extended and not terminated. This value is determined by the evaluation of its stream expression, the main mechanism of specification in RTLola. This is captured by the predicate $alive(s_j, \alpha, i)$, which is satisfied for stream instance $s_j(\alpha)$ iff $(\sigma^{\alpha}_{inv_j} = \textbf{true} \wedge \sigma^{\alpha}_{ext_j} = \textbf{true}) \vee (\sigma^{\alpha}_{ext_j} = \textbf{true} \wedge \sigma^{\alpha}_{ter_j} = \textbf{false})$. Each instantiated stream expression e^{α}_j of $e_j(t_1, \dots, t_m, s_1, \dots, s_n, p_1, \dots, p_k)$ has access to the parameter valuation of the current stream instance, as well as access to past and present values of all input streams and output stream instances.

To handle the timing behavior, we define the following helper functions for clocks, timestamps, and positions:

- **gc** refers to the global clock which maps a global position i in the evaluation model to a timestamp.
- A local clock **lc** for discrete ticks, which maps a stream instance s^{α}_j and a global position i to the number of times this instance has been extended since it has been last invoked.
- A function **lc2gc**, which maps the stream instance, global position and a discrete offset back to its global time position.
- For a stream instance s^{α}_j, and a timestamp t, let $\textbf{hold}(s^{\alpha}_j, t)$ return the largest global position where s^{α}_j is extended and has a timestamp equal or smaller than t.
- For a stream instance s^{α}_j, and timestamps t_1 and t_2, let $\textbf{collect}(s^{\alpha}_j, t_1, t_2)$ return the sequence of values of stream instance s^{α}_j which fall between the timestamps t_1 and t_2, i.e. all values of s^{α}_j where $s^{\alpha}_{ext_j}$ is true and where the corresponding **gc**-timestamp is greater or equal than t_1 and less or equal than t_2.

Let $e_j(t_1, \dots, t_m, s_1, \dots, s_n, p_1, \dots, p_l)$ be the stream expression of stream s_j. Then, for each stream instance $\sigma^{\alpha}_j \in \Gamma$ with valuation $\alpha = (v_1, \dots, v_l)$,

$$\sigma_j^\alpha(i) = \begin{cases} \mathrm{val}(e_j[p_1/v_1, \dots, p_l/v_l])(i) & \text{if } \mathit{alive}(s_j, (v_1, \dots, v_l), i) \\ \bot & \text{otherwise} \end{cases}$$

with the following evaluation function $\mathrm{val}(e[p_1/v_1, \dots, p_l/v_l])(i)$:

- constants:
  $\mathrm{val}(c)(i) = c$

- function application:
  $\mathrm{val}(f(e_1, \dots, e_h))(i) = f(\mathrm{val}(e_1)(i), \dots, \mathrm{val}(e_h)(i))$

- conditional expressions:
  $$\mathrm{val}(\mathbf{ite}(b, e_1, e_2))(i) = \begin{cases} \mathrm{val}(e_1)(i) & \text{if } \mathrm{val}(b)(i) = \mathbf{true} \\ \mathrm{val}(e_2)(i) & \text{otherwise} \end{cases}$$

- discrete stream access:
  $$\mathrm{val}(s_j(\alpha)[h, d])(i) = \begin{cases} \sigma_{s_j}^\alpha(\mathbf{lc2gc}(s_j^\alpha, i, h)) & \text{if } 0 \leq \mathbf{lc}(s_j^\alpha, i) + h < N \\ d & \text{otherwise} \end{cases}$$

- real-time stream access:
  $$\mathrm{val}(s_j(\alpha)[r, d])(i) = \begin{cases} \sigma_{s_j}^\alpha(\mathbf{hold}(\sigma_{s_j}^\alpha, \mathbf{gc}(j) + r)) & \text{if } 0 \leq \mathbf{hold}(\sigma_{s_j}^\alpha, \mathbf{gc}(j) + r) < N \\ d & \text{otherwise} \end{cases}$$

- real-time sliding window:
  $$\mathrm{val}(s_j(\alpha)[r, \gamma, d])(i) = \begin{cases} \gamma(\mathbf{collect}(\sigma_{s_j}^\alpha, \mathbf{gc}(j) - r, \mathbf{gc}(j))) & \text{if } (\mathbf{gc}(j) - r) \geq 0 \\ d & \text{if } \mathbf{gc}(j) - r) < 0 \end{cases}$$

5.2 Window Aggregation Functions

For the evaluation of a real-time sliding window expression $s[r, \gamma, d]$ in the context of the stream expression of stream s', the main sources of time and space complexity are the aggregation function $\gamma : T^* \mapsto T'$, the window duration r, and the extension rate of stream s'.

In a generic approach, we would store all input events received within the duration r in a buffer, and whenever s' is extended, recompute γ over the full buffer (containing T^*) to compute the value of the sliding window expression for the further evaluation of the expression of stream s'. For the general case of aggregation functions γ, such as the *median*-function, this is necessary.

In order to ensure a memory guarantee for the monitor, however, storing all possible input events of a variable rate input stream is infeasible. Furthermore, recomputing γ on every extension over the full buffer containing T^* may typically share a lot of operations with the previous computation of γ for s'. For aggregation functions γ with certain properties, this is not necessary, and an optimized paning approach to the sliding window computation similar to [52] can be followed.

First, the window duration r is split into equi-distant time panes $tp_1, \dots, tp_{r \cdot y}$ according to the extension rate y of the stream s', as illustrated in Figure 5.2. For each pane tp, the function γ is used to pre-aggregate the input values of a pane to an intermediate value $\gamma(tp)$; these computations will be performed on the input rate. Then, the evaluation of the sliding window expression will be performed by computing γ over the pre-aggregated values $\gamma(tp)$ of each pane tp. Since the number of panes is fixed, this computation is time-deterministic. Also, this computation on the pre-aggegrates does not need access to the individual input values since they are summarized in $\gamma(tp)$ – therefore the space requirement is also pre-determined and bounded. This is not possible for all types of aggregation functions, only for *efficient* aggregation functions. The important property of γ needed is *associativity*, as it allows us to re-arrange the order of the evaluation of γ. Most aggregation functions, such as *max*, *min*, *sum*, *integral* have the needed property. If the function has also an inverse operation, the effect of a single pane to the value can be reversed, which leads to only two operations needing to be performed in the fixed-rate component: the effect of the outgoing pane needs be reversed, and the pre-aggregated value of the incoming pane needs to be included.

5.3 Memory Analysis

While in Chapter 4, we could ensure a bounded memory requirement for a monitor by bounding the number of instantiations, the variable-rate input poses a separate obstacle for strict memory guarantees.

Our main approach is a constructive one: for practical applications it can be sufficient to define a static analysis for RTLola specifications, which can be used to guide the specification designer towards formalizations of the monitoring properties which do not lead to unbounded memory usage. This approach still allows to accept the memory penalty where it is necessary from an application point of view and allows the full expressiveness of the language. Therefore, we allow to specify a fixed evaluation rate for the output streams with the keyword `frequency`.

Our approach is based on an extended version of the dependency graph (for classic Lola, see Section 3.1.3) of the specification as an analysis method to find unbounded or high memory requirements.

Annotated dependency graph For an RTLola specification Φ, its *annotated dependency graph* $G_\Phi = \langle V, E, \pi, \lambda \rangle$ is a weighted, directed and annotated multi-graph defined as following:

- A set of vertices V representing input streams and stream variables, with $V = \{t_1, \dots, t_m, s_1, \dots, s_n\}$,
- a set of edges E, which defines the stream accesses and thus dependencies between the streams, where $E = \{(s_i, v) \mid v \in \{inv_i, ext_i, ter_i\} \vee e_i \text{ accesses } v\}$,
- a vertex annotation function π, which annotates each vertex with its fixed evaluation rate, or with `var` if it is a variable-rate stream,
- an edge annotation function λ, which labels the stream accesses of E with their corresponding real-time and discrete offsets and sliding window accesses.

As in Section 3.1.3, V and E are derived directly from the syntax of the specification, with the obvious addition of λ. For the vertex annotation function π, we initialize the input stream variables with `var`. If the output streams have a given, fixed rate, we initialize them with this rate. For all other streams v, we define $\pi(v)$ to be the maximal rate of the streams v accesses (where `var` dominates all values).

Computing the memory requirement To compute the memory requirement of Φ, we will use the two annotation functions π and λ to compute the number of values which need to be stored by the online monitoring algorithm. For the classic Lola parts, we can compute this by taking the garbage collection vector offsets, which are directly derived from the maximal (transitive) offset access. For real-time sliding windows where the stream expression of stream s' with rate $\pi(s')$ contains the sliding window access $s[r, \gamma, d]$, we consider the following situations:

- If $\pi(s) = $ `var`, and γ is **not** efficient, then unbounded memory is needed,
- If $\pi(s)$ has frequency y Hz, and γ is **not** efficient, then $y \cdot r$ values have to be stored,
- If $\pi(s) = $ `var`, and γ is efficient, then $\pi(s') \cdot r$ values have to be stored,

- If $\pi(s)$ has frequency y Hz, and γ is efficient, then $min(\pi(s') \cdot r, y \cdot r)$ values have to be stored.

The memory requirement of Φ is the sum of the memory requirements of its output streams, where the requirement for each output stream is multiplied by the bound on its number of stream instances, as for Lola 2.0.

5.4 Online Monitoring Algorithm

We split the presentation of the online monitoring algorithm into two components: the input event-triggered part and the time-triggered part. Both run in concurrent threads and share a part of their data structures, but writes to the data structures are disjoint, since the stream evaluations can be split between the two components.

A difference between RTLola and Lola 2.0 is that not all inputs have to arrive at the same time, so only a subset of the input streams may be extended on an incoming event. For the purpose of the algorithm implementation, the set U of unresolved stream expressions is split into further parts to keep track of the stream instances. In the variable part of the algorithm, we register the incoming input events in their respective stream instances and sliding windows. An instance is compatible to the current event if its parameters match with the values of the event. We also keep track of invocations induced by the current input streams. The list of *efficient* streams in this algorithm is a subset of the output streams which can be directly evaluated without stream accesses with offsets, but with the usual functions, including arithmetic expressions and equality checks. This allows for the pre-evaluation of more expressive triggers and auxiliary streams which run on the input rate.

The fixed-rate part of the algorithm is similar to Lola 2.0, but note that it is triggered by a clock signal, not by an input event. The book-keeping list created in the variable-rate part is used here to initialize the set of stream instances to be extended and terminated. Since every fresh stream instance value may again trigger new invocations, extensions or terminations, we have to run the evaluation in a fixpoint.

Efficent bindings For Algorithm 4, resolving the existential quantifier of Line 14 has a large impact on performance, as the list of stream instances may be large, but only a comparatively small number of them are usually extended with each input event. To quickly identify the relevant stream instances to extend, and since most extension streams are essentially equality checks of input stream values to parameter values, we keep an extra index for these *efficient bindings* to support a

Algorithm 4 Variable-rate part of the online monitoring algorithm for RTLola

```
    Input: Incoming Event $E$, containing $v_i$ for subset of input streams $s_i$,
           Timestamp $ts$
    Output: List of Triggers $T$
 1: for all $(s_i, v_i) \in E$ do                                  ▷ store input events
 2:     extend $s_i$ by fresh value $v_i$
 3:     add $s_i$ to depending window panes with time $ts$
 4: end for
 5: for all $s$ invoked by $E$ do
 6:     $s^\alpha \leftarrow$ invoke $s$ with valuation $\alpha$ compatible with $E$
 7:     if $s$ is efficientStream then
 8:         $efficientStreamInstances \leftarrow efficientStreamInstances \cup \{s^\alpha\}$
 9:     else
10:         $outputStreamInstances \leftarrow outputStreamInstances \cup \{s^\alpha\}$
11:     end if
12: end for
13: for all $s \in efficientStreams$ do
14:     if $\exists\, s^\alpha \in Instances(s)$ compatible with $E$ then
15:         $v \leftarrow value(s^\alpha)$ with $E$                     ▷ extend $T$ if appropriate
16:         extend $s^\alpha$ by $v$
17:         add $v$ to depending window panes with time $ts$
18:     end if
19: end for
20: return active triggers $T$
```

Algorithm 5 Fixed-rate part of the online monitoring algorithm for RTLola

```
    Input: Clock signal at time $ts$, $outputStreamInstances$
    Output: List of Triggers $T$, Outputs $O$
 1: $O \leftarrow \emptyset$
 2:                                          $\triangleright$ Initializations from variable-rate part
 3: $Ext \leftarrow \{s^\alpha \mid s^\alpha \in outputStreamInstances \wedge \texttt{ext}_s^\alpha = \textbf{true}\}$
 4: $Ter \leftarrow \{s^\alpha \mid s^\alpha \in outputStreamInstances \wedge \texttt{ter}_s^\alpha = \textbf{true}\}$
 5: while fixpoint $U$ do
 6:     for all sliding windows $w$ do
 7:         compute value of $w$ from panes with aggregation at time $ts$
 8:     end for
 9:     for all $s^\alpha \in Ext$ do
10:         $v \leftarrow$ (partially) evaluate $s^\alpha$
11:         if $v$ is fully evaluated then
12:             extend $s^\alpha$ by $v$
13:             add $v$ to depending window panes with time $ts$
14:             $O \leftarrow O \cup \{s^\alpha, v\}$
15:             for all $s'^{\alpha'}$ invoked by $s^\alpha$ do          $\triangleright$ $s^\alpha$ may invoke many $s'$
16:                 invoke $s'$ with valuation $\alpha'$ based on $v$
17:             end for
18:             if $v = \textbf{true}$ then          $\triangleright$ $s^\alpha$ may extend or terminate many $s'^{\alpha'}$
19:                 for all $s'^{\alpha'}$ extended by $s^\alpha$ do
20:                     $Ext \leftarrow Ext \cup s'^{\alpha'}$
21:                 end for
22:                 for all $s'^{\alpha'}$ terminated by $s^\alpha$ do
23:                     $Ter \leftarrow Ter \cup s'^{\alpha'}$
24:                 end for
25:             end if
26:         end if
27:     end for
28:     for all $s^\alpha \in Ter$ do
29:         terminate $s^\alpha$
30:     end for
31: end while
32: return active triggers $T$, outputs $O$
```

fast lookup for the existential quantifier.

5.5 Experiments

The online monitoring algorithm for RTLola was implemented in Swift. All experiments were conducted on a quad-core Intel i7 processor with 2,7 GHz and 16GB RAM. The input events were read from the internal SSD and piped to the monitoring process.

5.5.1 Memory Requirement vs. Input Rates

A synthetic input trace of 1 million events length with random data was used to determine the memory and runtime behavior of our implementation. Using rate control on the input pipe to the monitoring process, we simulated different loads on the monitoring system. The input rate ranged from 10k events per second to 100k events per second in steps of 10k events.

Runtime and memory consumption were measured with the standard Unix `time` utility and are reported in seconds and megabytes, respectively. The RTLola specification used for the experiment is shown in Listing 5.2. It has a monitor frequency of 100 Hz and calculates the five-second-sums over three input streams and checks a trigger condition on them. The results of the experiment are shown in Figure 5.3.

```
input double a,b,c
output double sumabc := (a[5s, sum, 0.0] + b[5s, sum, 0.0] + c[5s, sum, 0.0])
trigger any(sumabc < 0.3)
frequency 100 Hz
```

Listing 5.2: Specification used for inputrate performance test.

The memory usage in this experiment was constant over the varying input rate. We can also see that the time does not increase after an event rate of 30k events per second, which indicates the throughput limit for the monitor for this particular specification. The corresponding throughput is approximately 25k events per second.

5.5.2 Data Analysis

To evaluate the performance of the implementation for data analysis tasks with more challenging and complex specifications, we analyzed web shop product re-

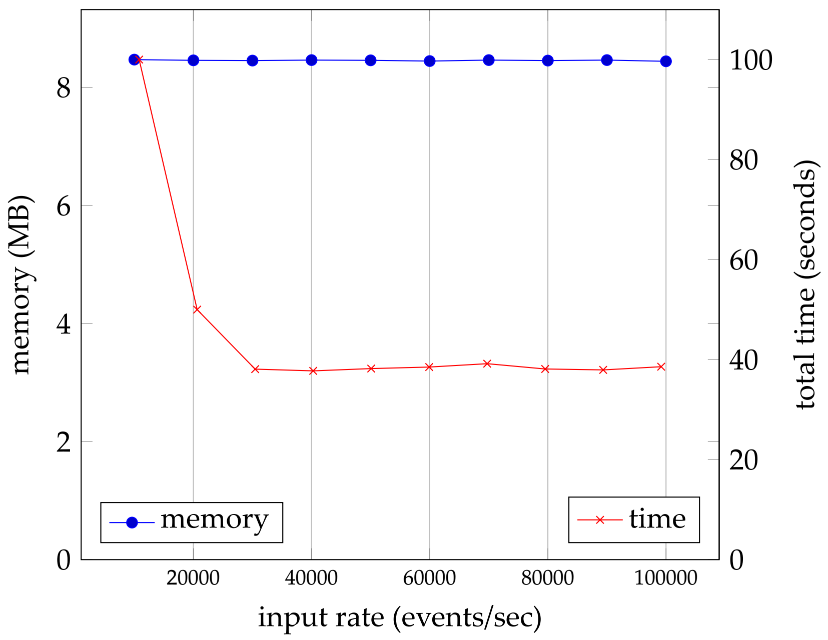


Figure 5.3: Memory usage and computation time vs. input rate for synthetic trace data. Event size is 24 Bytes per event. Overall trace length is 1 million events.

view data [43], which contained the review rating for each product. The specification for our experiment is depicted in Listing 5.3. This specification takes as inputs the identifier of the review (`uid`), the identifier of the product (`pid`), and the 1-through-5-star rating of the review `stars`. The main goal of the specification is to determine "hot" products, which receive a lot of recent reviews that are better than their longer running average.

In our experimental results, shown in Figure 5.4, we recorded runtime (in seconds) and memory usage (in GB) per main product category. The input trace lengths for different product categories went up to 7 million events. The product categories are indicated on the graph on the x-axis, where we show the number of unique product identifiers contained in the dataset. This gives an indication of how many stream instances have to be maintained by the monitor throughout the dataset.

The results for memory usage measurement for the larger sets is influenced by the memory management in the monitor. Still, we can see that also data analysis tasks over large datasets are possible. The maintenance of large sets of stream instances (100k – 500k) is possible, but heavily influences memory usage.

```
frequency 0.1 Hz
input string uid, pid
input double stars
timeinput double ts
output double stars_by_prod <string prod>
        invoke: pid
        extend: pid = prod
        := stars
output double star_avg_l <string prod>
        invoke: pid
        := stars_by_prod(prod)[100 s, average, 0.0]
output double star_avg_s <string prod>
        invoke: pid
        := stars_by_prod(prod)[10 s, average, 0.0]
output double avg_delta <string prod>
        invoke: pid
        := star_avg_l(prod)[0, 0.0] - star_avg_s(prod)[0, 0.0]
trigger any(avg_delta > 1.0)
```

Listing 5.3: Specification to analyse potentially popular products on an online shopping platform. `timeinput` specifies the time reference input.

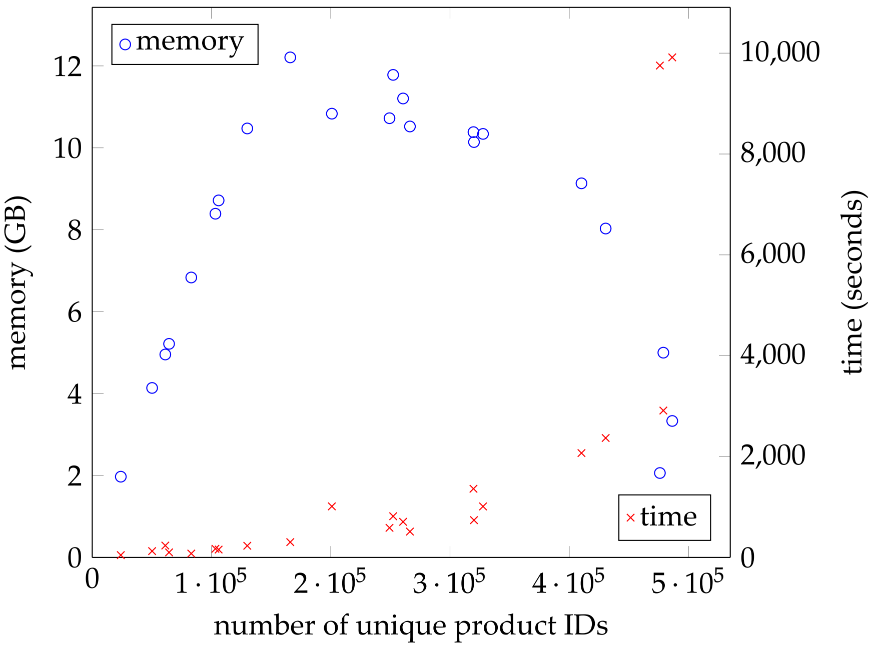


Figure 5.4: Memory usage and computation time vs. the number unique product identifiers per trace file for Amazon review rating data. The computation did not terminate for the two largest categories which include 1.2 and 2.3 million different product due to memory limitations.

6 Case Study: Monitoring in UAS

Unmanned aerial systems (UAS) with autonomous decision-making capabilities are becoming increasingly interesting for civil applications such as logistics and disaster recovery. From these new use cases follow new requirements: integration into classic, already populated civilian airspace, flying over urban territory, using much smaller and potentially cheaper aircraft in comparison to classic piloted aircraft. The safety case for systems in this area will heavily rely on the correctness of the behavior exhibited by the software and hardware components of the flight system in its specific application scenario, since there is no onboard pilot to monitor the system and react. On-ground pilots are limited to visual-line-of-sight (VLOS) applications. Even without human passengers on board, the stability and correctness of the system has potentially harmful consequences for human life on the ground.

A research testbed of autonomous UAS is developed and maintained by the Institute of Flight Systems of the German Aerospace Center (DLR). Within the DLR ARTIS platform (**A**utonomous **R**esearch **T**estbed for **I**ntelligent **S**ystems), a common flight software platform is used for different classes of aircraft, from fixed-wing aircraft to rotary-wing aircraft with take-off weights ranging from 1kg to 150kg. Since ARTIS is primarily a research testbed, only some of the infrastructure is stable (such as ground station, communication links and a basic flight control), and it supports experimentation with different flight controllers, sensor configurations, sensor fusion, optical navigation, path planning and mission management. Therefore, the majority of software components is under constant development.

The systems support high-level autonomy: given a mission plan with a set of GPS waypoints and a high-level task, such as 3D-mapping an area of interest, the system will automatically derive and refine (in an online manner) a flight path and autonomously control the lower-level layers, such as sensors and actuators of the system. The system architecture follows a standard **G**uidance-**N**avigation-**C**ontrol (GNC) pattern, illustrated in Figure 6.2.

In order to bring such autonomous systems closer to use in realistic applications, there are two approaches: (1) classic certification entails the need to demonstrate and certify the correctness of the automated software functions, which control the autonomous decision-making, for all possible operations, and (2) operation-specific certification, which takes the specific operation into account and reduces the effort to certify the potentially complex software functions.

Due to its role as a research platform, the underlying software components are changed frequently, but the interfaces between many safety-critical components in the system are more stable. Therefore, we introduced runtime monitoring with different versions of Lola to the ARTIS platform starting with the data flowing on these component interfaces. This gave a lightweight path to implementation and instrumentation into the existing software platform.

The monitoring approach was also used to support the development process in the platform in the form of a debugging tool: existing flight log files were used to guide the specification development and to identify misbehaving components by flight operators, which removed the need to involve the module developers for the initial analysis. Timing specifications were of particular interest for this task, which identified anomalies in the timing behavior of individual components and allowed to quickly pinpoint the root cause of problems.

This part of the thesis was carried out within an ongoing cooperation between Saarland University and the Institute of Flight Systems, DLR Braunschweig. The chapter is based on [74] and [1].

Related Work Runtime monitoring applied to unmanned aerial systems has mainly been explored in a series of works by Rozier et. al. at NASA and Iowa State [38, 67, 61] in the monitoring approach R2U2. The main differences to the case study described here is in the choice of specification language and monitor integration. The specification language used for R2U2 is based on an extension of linear-time temporal logic with an enriched assertion language for real-valued signals and a top-level statistical analysis using bayesian networks. Their monitor is implemented on an FPGA and integrated via an existing hardware bus and thus only has access to externally visible information, whereas our more tightly coupled software integration gives us access to the internal state of software components.

Other approaches for drone monitoring include the language Copilot, which was used to monitor specific airspeed sensor properties [57] and SAFEGUARD, a monitoring device specific for geo-fencing properties [26]. For the specific property of avoiding ground collisions, Auto GCAS has been developed for military applications [70].

In the context of formal methods for the ARTIS platform, previous work on software verification was described in [72], verification and validation of the mission manager component in [71], and model checking in [73].

Some of the adoption barriers of formal methods for realistic applications have been presented in [25] and especially for formal specification languages in [63].

6.1 DLR ARTIS Research Platform

The UAS research platform ARTIS, maintained by the Institute of Flight Systems at DLR Braunschweig, is focused on the study of system autonomy, security and safety to realize intelligent functions within affordable UAS aircraft. The platform consists of a fleet of UAS with take-off weights from 1kg to 150kg as well as a common software platform, which is parameterized for each individual aircraft and configuration.

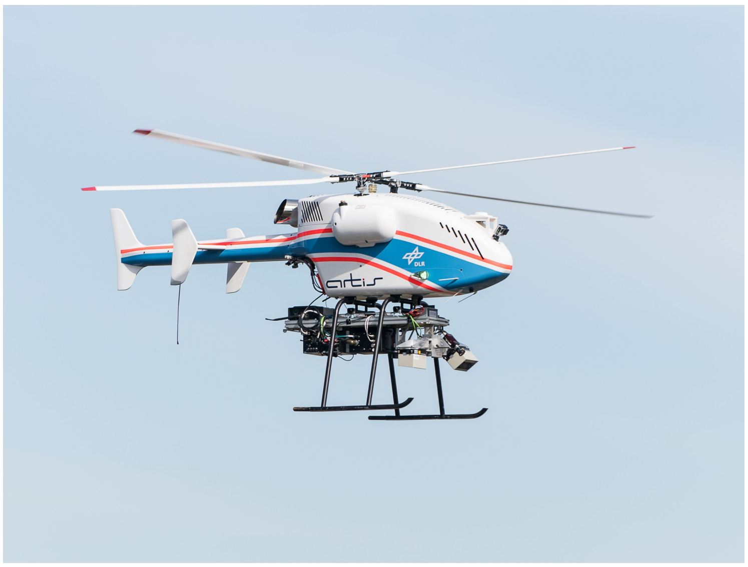


Figure 6.1: SuperARTIS, with an intermeshing rotor, shown with full sensor setup and two flight computers. Take-off weight: 85kg

As previously mentioned, the overall system architecture, as shown in Figure 6.2 is split into three main layers:

- The **flight control** component realizes the **control** layer, the lowest-level (i.e. closest to hardware) layer in the architecture. This component uses a closed-loop controller to command the desired position, height, speed and orientation of the aircraft, e.g. the 6D-pose of the UAS. Its output signals are the actuator control commands, which drive the actuators, such as the motor throttle actuator and rotor blade pitch actuator. The controller is robust against external disturbances such as wind and keeps the aircraft on the given trajectory.

- The **navigation filter** component realizes the middle, **navigation** layer of the architecture. Its main task is to estimate the current state of the aircraft: the position of the aircraft in its environment (localization). As input signals, the component receives sensor readings from a multitude of heterogenous sensors, such as GPS, IMU, altimeter, different camera systems, radar systems or other optical sensors such as LIDAR. Therefore, the primary function of this component is performing *sensor fusion* of these sensor readings into consistent state estimates.

- The **mission manager** component realizes the topmost, **guidance** layer of the system. It performs mission planning and execution. It receives high-level instructions and objectives from a ground control station and keeps track of obstacle and mapping information of the potentially unknown flight area, which is received from a computer vision component. The high-level task of exploring and mapping an unknown area is here broken down into a number of suitable waypoints and path planning is used to generate an optimal flight path. Additionally, it receives the current state estimates from the navigation filter component for online replanning. Then, the generated flight path is used to compute the target 6D-pose for the flight control component.

Each of the layers in the hierarchy represents a different level of system autonomy. ARTIS aircraft have been evaluated in flight tests with respect to closed-loop motion planning in obstacle-rich environments.

During flight, all of the system components collect extensive logging information for debugging, post-flight analysis and evaluation. This log information contains input sensor data, timestamps, and internal states. The data is sent to external storage, and a subset is sent via radio transmission as telemetry data to the ground control station.

The timestamps collected in these log files have a common time base, since the complete software platform runs on the same CPU and RTOS. Also, the system has a *time-triggered* architecture: all software functions are triggered through a base task, where the GNC runs at 50 Hz and all other parts via subsequent synchronous function calls. Therefore, the system has a native synchronous timing model. Since the components are tightly coupled via function calls, there is no hardware bus in the system.

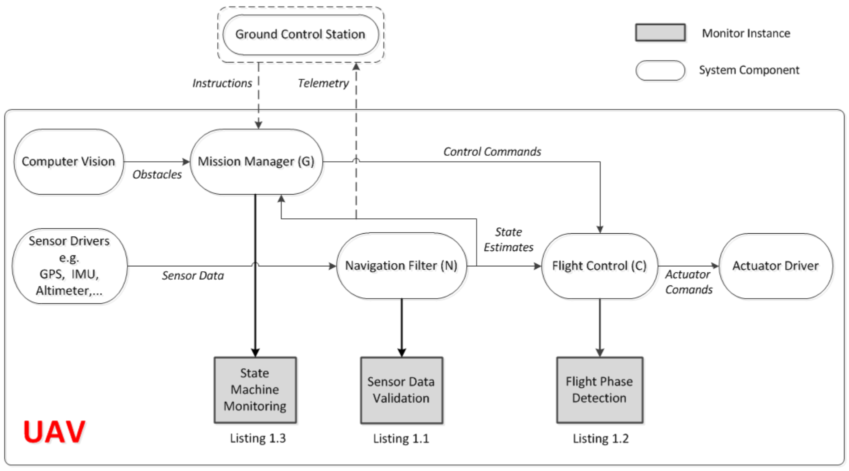


Figure 6.2: Illustration of the Guidance-Navigation-Control (GNC) architecture with its main system components (white round boxes) of the DLR ARTIS software platform with added monitor instances (grey rectangular boxes).

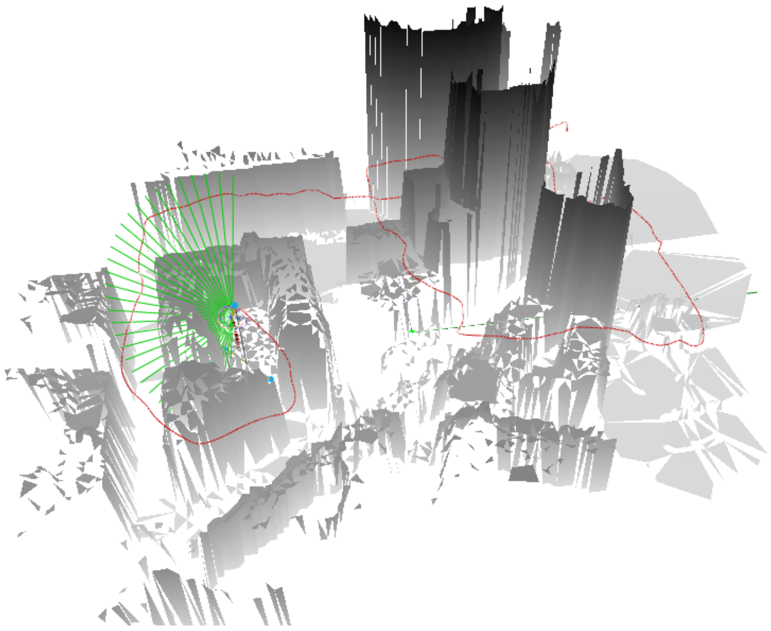


Figure 6.3: A simulation of autonomous exploration and mapping of an unkown area with onboard sensors. The flown path is indicated in red, the simulated sensor distance in green, and mapped obstacles in grey.

Due to the software architecture, time overruns in software functions along the effect chain can result in variance in the timestamps in the log files of the compontens as compared to their ideal frequency (timing jitter). This behavior can be used to pinpoint component faults, as all later components on the chain will receive their inputs with some delay due to the tight coupling.

6.1.1 Certification of UAS

There are two types of risks associated with the operation of UAS: air risks, i.e. collisions with other manned or unmanned aircraft and ground risks, i.e. for people or property on the ground. Air risks are currently mainly mitigated by keeping UAS in separate airspace from manned flight and by mandating visual-line-of-sight flying for most UAS. Ground risks are mitigated by restricting the flight areas, for example by prohibiting flights over crowds or urban areas and restricting the weights of aircraft. A comprehensive model of the risks of UAS can be found in [46].

The regulatory environment of unmanned aircraft systems (UAS) is not yet fully established. There has been significant progress in recent years both on an international level, i.e. through JARUS[1] and EU regulations proposed by EASA[2] [31, 30, 32] and national implementations, such as the German "Drohnenverordnung" [75]. However, most of the existing regulation concern small UAS in the so-called *open* category: below 25kg take-off weight with heights below 100m in visual-line-of-sight (VLOS) of the operator, which include most of the currently commercially available systems.

A second, more challenging category of these EASA/JARUS regulatory proposals is the *specific* category of UAS, which contains all aircraft which do not fall into the limits of the open category (and are not part of the certified category, which contains fully certified aircraft). Here, the flight permission is based on the specific concept of operations for the specific aircraft and the specific proposed operation. In order to obtain such an operation-specific permission, a risk assessment has to be carried out. The proposed risk assessment method is the so-called *specific operations risk assessment* (SORA) [46]. The goal is to allow for a gradual path to certification for UAS and more and more operations.

Within the risk assessment, safety measures such as geo-awareness and flight termination systems are one way to support the classification and certification, as they present a harm barrier to threats such as leaving the operation area, which can potentially result in a much higher risk than originally specified.

The third category of the proposals is the *certified* category, which would entail a

[1]Joint Authorities for Rulemaking on Unmanned Systems
[2]European Aviation Safety Agency

full safety certification of an aircraft before delivery, similar to manned aircraft as determined by the relevant system and software standards, i.e. DO-178C [58] and by means of e.g. formal methods [59].

As for all safety-critical systems, when performing a safety assessment, fault detection within components and appropriate fault reactions are an important measure to achieve a fault-tolerant system design and ensure the safety objectives. Therefore, dynamically monitoring inputs, outputs and states of components is a standard technique to ensure the proper behavior of the overall system. For monitors as system components, however, verification and validation for the monitor components themselves are an important issue, since significant trust is placed in the components, both in terms of the correct specification to be monitored (validation) and the correctness of the implementation of the monitor (verification).

The case for formal monitoring The validation of monitor specifications is usually performed through reviewing the specification, independently of whether the specifications are written in natural language or in a formal specification language. The advantages of formal specifications are in this case especially the unambiguity and well-defined semantics, which have also been recognized in safety standards and give extra verification credit. In practice, this requires a sufficiently expressive specification language, such that all relevant requirements can be captured and validated by domain experts.

The verification problem for the monitor component in the case of formal specification languages can largely be offloaded to a tool qualification problem for the monitoring tool, which has to be certified to either implementing the semantics of the language correctly, or to certify that its output, i.e. the generated hardware description or software source code is correct. Of course, there is some remaining test effort for the monitor integration to the system.

Beyond monitoring inputs, state, and outputs of individual components, there are two important aspects of autonomous UAS which require the specification and monitoring of system-level behavior:

- The concept of operations, which defines operational restrictions for to the aircraft. These restrictions, such as a maximum flight height or a geo-fence have to be monitored by the aircraft. A dynamic approach of risk assessment with geo-fences and Lola has been described in [66].
- Autonomous functions implemented in the UAS. Once the control software takes high-level decisions based on complex sensor inputs, such as selecting

the next waypoint due to obstacles or changing the time to hover in a location, it becomes impossible to exhaustively test all possible environmental situations of the autonomous UAS to verify its correct behavior. Since we can neither rely on an on-board pilot or an on-ground UAS operator (a communication link might be unreliable, and flights may need to leave the visual line of sight), monitoring can be used to assess the current vehicle state and environment and ensure the correctness of the current situation and decision.

For the path to certification of the systems, the formal monitor specifications and the monitor tool can be independently assessed by appropriate certification authorities, which then supports the safety case of the overall system.

6.1.2 Impact on system development

The software development of the ARTIS software platform is in the context of a dynamic research environment with an established development process with continuous integration and testing on multiple system levels. One important benefit of the specification writing effort within the case study is consistent documentation about the interface assumptions, the source of signals and the basic error cases of the components.

While the primary intended use of the specifications was for generating online monitors for the aircraft, we have also used the specifications for debugging purposes using offline monitoring on previous experimental data, and also used them within the extensive simulation environment.

The main advantage for the development process comes in extended testing and debugging support for the system. Unobtrusively monitoring the components during test and simulation gives an early indication of where faults are present in the system. Long test runs with defective software versions can thus be avoided. The specifications can then be extended to give more debug insight without changing the system itself, all without significantly changing the real-time behavior, as we can see in the experimental section.

6.2 Lola Extension

A small number of syntactic and usability improvements have been added to the language during the case study to adapt it to domain needs:

- Extended keywords (of static type): `position`, which gives access to the current position on the trace, and maximal values for the numeric types, which are useful for default values: `int_min`, `int_max`, `double_min`, `double_max`.

- Absolute offsets: the #-operator in front of a stream access gives the possibility to access an absolute stream position, which is useful for example to obtain the initial value of a stream in the first position.[3]

- `switch`-operator: To avoid long chains of if-then-else expressions for encoded state machines, we have introduced a `switch`-operator which may match in its condition on an integer expression and can therefore be efficiently implemented, leading to more succinct and readable state machines.

- Floating-point arithmetic and trigonometric functions: the type `double` and associated arithmetic functions as well as trigonometric functions were added to support the specifications needed for geo-information and other physical measures.

For the monitoring tool, we added a few usability improvements to support the usage online and offline:

- To improve the online-feedback to the user, the following modes for trigger output are implemented: `trigger`, which prints the specified message whenever the condition holds, `trigger_once`, which triggers only in the first instance when the condition holds, `trigger_change`, which triggers whenever the condition becomes true, and `snapshot`, which prints the current values for all streams.

- To support post-flight analysis and pre-processing, tagging and filtering operators were added for offline analysis, which enabled a multi-stage analysis of log files by reading a log file and producing a modified log file. A filtering-operator would only copy events to the output whenever a condition matches, and a tagging operator would copy the input log file to the output with an additional field containing the tag value.

[3]This can be implemented in finite memory, since a finite specification can only reference finitely many positions.

6.3 Specifications in classic Lola

We present in this section representative specifications from [1] for all three layers of the architecture displayed in Figure 6.2. These are specified in syntactically extended classic Lola, as introduced in the previous section. The specifications have been obtained and developed from interviews with the responsible engineers of the software modules and through collaborative specification writing. Initial validation of the specifications was performed through an offline analysis of previous flight data sets. The full set of specifications has been documented in [65].

The time-triggered nature of the system design with a target event frequency of 50 Hz gives a new event every 20ms in the ideal case. The incoming events are timestamped, represented by two fields: `time_s` and `time_micros`.

Sensor data validation The sensor data validation specification, shown in Listing 10, on a technical level primarily detects short-time sensor deviations of two sensor signals, the GPS sensor and the IMU sensor. Also, this specification computes frequency measurements on the input signals for debugging purposes. On an architectural level, it checks plausibility of the output of the navigation filter part of the sensor fusion component, which is used to estimate the state of the vehicle. The state of the vehicle is used later on in further software components as part of the planning for further actuator commands.

In lines 4–10, the input frequency of the monitor events is analyzed, as missing or delayed event values indicate a problem in the system.

The main part of the specification compares the traveled distance from two sensor sources to detect conflicting information. The first estimation is through the current velocity vector (line 12) of the IMU measurements `ug,vg,wg` and the time passed since the last measurement (lines 30 and 31). The second distance estimation is through the GPS latitude and longitude with the Haversine formula for point-to-point distance on a sphere, implemented in lines 13–28, where the conversion to radians is handled in lines 16–19. Both distances are then compared against a threshold (lines 32–34), which accounts for signal noise and insignificant height differences in the GPS signal. Overall, this ensures that significant GPS jumps are detected via comparison to a more trustworthy local sensor.

All calculations are performed in Lola itself, using the introduced trigonometric functions and floating point support, as well as the snapshot-feature to mark differences in the sensor signals for later developer interpretation.

```
input  double lat, lon, ug, vg, wg, time_s, time_micros
output double time := time_s + time_micros / 1000000.0
output double flight_time := time - time#[0,0.0]
output double frequency := switch position{
                            case 0  { 1.0 / ( time[1,0.0] - time ) }
                            default { 1.0 / ( time - time[-1,0.0] ) } }
output double freq_sum := freq_sum[-1,0.0] + frequency
output double freq_avg := freq_sum / double(position+1)
output double freq_max := max( frequency, freq_max[-1,double_min] )
output double freq_min := min( frequency, freq_min[-1,double_max] )

output double velocity := sqrt( ug^2.0 + vg^2.0 + wg^2.0 )
const  double R        := 6373000.0
const  double pi       := 3.1415926535

output double lon1_rad := lon[-1,0.0] * pi / 180.0
output double lon2_rad := lon * pi / 180.0
output double lat1_rad := lat[-1,0.0] * pi / 180.0
output double lat2_rad := lat * pi / 180.0

output double dlon     := lon2_rad - lon1_rad
output double dlat     := lat2_rad - lat1_rad
output double a := (sin(dlat/2.0))^2.0  +
                   cos(lat1_rad) *
                   cos(lat2_rad) *
                   (sin(dlon/2.0))^2.0
output double c := 2.0 * atan2( sqrt(a), sqrt(1.0-a) )
output double gps_distance := R * c

output double passed_time     := time - time[-1,0.0]
output double distance_max    := velocity * passed_time
output double dif_distance    := gps_distance - distance_max
const  double delta_distance  := 1.0
output bool   detected_jump   := switch position {
                                  case 0 { false }
                                  default { dif_distance >  delta_distance } }
snapshot detected_jump with "Invalid GPS signal received!"
```

Listing 10: The specification used for Sensor Data Validation

Flight Phase Detection The monitored component in this specification (Listing 11) is the flight control component. The actual velocity of the aircraft is given as input signals `vel_x, vel_y, vel_z`, calculated by the navigation filter. The actuator commands from the flight control component are given as reference input signals `vel_r_x, vel_r_y, vel_r_z` to the specification. Additional inputs are the current battery level (`power`) and the fuel level (`fuel`).

Again, to detect possible timing problems in the component, we calculate statistics on the timing of input events in lines 5–11. Phase detection for a flight controller is of interest to measure and model the fuel and power performance of a controller.

The first main function of the monitor, detecting flight phases, where the aircraft is approximately stationary for at least three seconds, is specified in lines 13–25. Here, the current inertial measurement velocity is compared to a dynamic velocity window specified by `velocity_max` and `velocity_min`. Once the window is larger than `vel_bound`, the dynamic window resets. The output `unchanged` keeps a counter since the last reset, and if it reaches 150 (equivalent to three seconds if the frequency is 50 Hz), the hover phase is detected. These hover phases can in post-flight analysis be compared against higher-level commands.

A second main function of the monitor evaluates the performance of the flight controller and actuators by comparing the commanded velocity vector to the actual, measured velocity vector in lines 27–34. In `dev_sum`, we integrate over the velocity deviation, keep an average as `dev_avg`, and additionally keep track of the worst deviating position and value in `worst_dev_pos` and `worst_dev`, again for later inspection by the flight and development engineers.

A third monitor function tracks the fuel level and power usage since the initial stream position and triggers notifications and alarms based on the levels in lines 36–43.

On a system level, the monitor specification ensures the flight control component works in a proper manner, and that the actuators actually react to the commanded velocity commands. Additionally, the monitor performs basic boundary checking for fuel and power.

Mission Monitoring In the third specification (Listing 12), the high-level mission manager component is subject to monitoring. The component is responsible to implement the uploaded flight mission which the component and the monitor receives from the ground control station, based on the state estimate of the vehicle and the results of the obstacle detection. The component includes an online mission planning part. In the specification, the different states of the component state machine are encoded and recovered from the input streams (lines 2–30). The state information is additionally enriched with information about the entry time into the current state (lines 32–33). The primary monitoring function is specified in lines 34–38, where a bounded liveness property is checked: If the state machine switches to state *Landing*, the `OnGround` signal becomes '1' within 20 seconds. In an extended version of the specification, more conditions for state transitions have been checked similarly.

```
input  double time_s, time_micros, vel_x, vel_y, vel_z,
              fuel, power, vel_r_x, vel_r_y, vel_r_z
output double time := time_s + time_micros / 1000000.0
output double flight_time := time - time#[0,0.0]
output double frequency := switch position{
                             case 0  { 1.0 / ( time[1,0.0] - time ) }
                             default { 1.0 / ( time - time[-1,0.0] ) } }
output double freq_sum := freq_sum[-1,0.0] + frequency
output double freq_avg := freq_sum / double(position+1)
output double freq_max := max( frequency, freq_max[-1,double_min] )
output double freq_min := min( frequency, freq_min[-1,double_max] )

const  double vel_bound       := 1.0
output double velocity       := sqrt( vel_x^2.0 + vel_y^2.0 + vel_z^2.0 )
output double velocity_max   := if reset_max[-1,false] { velocity }
                                else { max( velocity, velocity_max[-1,0.0]) }
output double velocity_min   := if reset_max[-1,false] { velocity }
                                else { min( velocity, velocity_min[-1,0.0]) }
output double dif_max        := difference(velocity_max, velocity_min)
output bool   reset_max      := dif_max > vel_bound
output double reset_time     := if reset_max | position = 0 { time }
                                else  { reset_time[-1,0.0] }
output int unchanged         := if reset_max[-1,false] { 0 }
                                else { unchanged[-1,0] + 1 }
snapshot unchanged = 150 with "Phase detected!"

output double  vel_dev := difference(vel_r_x,vel_x) + difference(vel_r_y,vel_y)
                          + difference(vel_r_z,vel_z)
output double  dev_sum   := vel_dev + dev_sum[-1,0]
output double  vel_av    := dev_sum / double((position+1)*3)
output int worst_dev_pos := if worst_dev[-1,double_min] < vel_dev { position }
                          else { worst_dev_pos[-1,0] }
output double worst_dev  := if worst_dev[-1,double_min] < vel_dev { vel_dev }
                          else { worst_dev[-1,0.0] }

output double fuel_p  := ( ( fuel#[0,0.0] - fuel ) /  (fuel#[0,0.0]+0.01)  )
output double power_p := ( (power#[0,0.0] - power) /  (power#[0,0.0]+0.01) )
trigger_once fuel_p  < 0.50  with "Fuel below half capacity"
trigger_once fuel_p  < 0.25  with "Fuel below quarter capacity"
trigger_once fuel_p  < 0.10  with "Urgent: Refill Fuel!"
trigger_once power_p < 0.50  with "Power below half capacity"
trigger_once power_p < 0.25  with "Power below quarter capacity"
trigger_once power_p < 0.10  with "Urgent: Recharge Power!"
```

Listing 11: The specification used for Flight Phase Detection

```
input double time_s, time_micros
input int stateID_SC, OnGround
const int Start                 := 0
const int MissionControllerOff  := 1
                                    ...
const int HammerHeadTurn        := 16

output double time := time_s + time_micros / 1000000.0
output double flight_time := time - time#[0,0.0]

output bool change_state := switch position {
                              case 0 { false }
                              default { stateID_SC != stateID_SC[-1,-1] } }
trigger change_state

output string state_enum := switch stateID_SC  {
                              case 0 { "Start" }
                              case 1 { "MissionControllerOff" }
                               ...
                              case 16{ "HammerHeadTurn" }
                              default{ "Invalid" }  }
output string state_trace :=
   switch position {
     case 0  { state_enum }
     default { if change_state
                 { concat(concat(state_trace[-1,""]," -> "),state_enum) }
                else
                 { state_trace[-1,""] }
             }
     }

output double entrance_time  := if change_state { time }
                              else { entrance_time[-1,0.0] }
const double landing_timebnd := 20.0
output double landing_info   := if stateID_SC = Landing { 0.0 }
                              else { time - entrance_time[-1,0.0] }
output bool landing_error    := stateID_SC = Landing & OnGround != 1  &
                                landing_info > landing_timebound
```

Listing 12: The specification used for Mission Planning and Execution

6.4 Specifications in RTLola

The case study described earlier was developed with a syntactically extended version of Lola, as described before, but did not use the newer developments of Lola 2.0 and RTLola. We have additionally reformulated and extended specifications from

```
frequency 50Hz
timeinput double time_s + time_ms / 1000000.0

input double velocity

output int freq_deviations = if (velocity[20ms,count,0] != 1 )
                        then freq_deviations[-1,0] + 1
                        else freq_deviations[-1,0]
```

Listing 13: Frequency Checks in RTLola

the application scenario using new features, to demonstrate the usefulness of the extensions in the application scenario and give a comparison to the earlier version of Lola.

Frequency Checks For the time-triggered ARTIS system, frequency checks on the input signals of the monitor interface were used to identify misbehaving components. In Listing 13, we show one way to implement a check for frequency deviations using time windows.

Bounded Landing Time In the mission monitoring specification (Listing 12), a bounded liveness property (lines 29–33) was checked after entering the *Landing* state, where the input signal `onGround` should be equal to `true` within 20 seconds. A similar property can be formalized in RTLola as shown in Listing 14. The window-operator is used to determine whether there has been a landing command within the last 20 seconds and no `onGround` signal was received in the same time window.

Flight Phase A second example of a more elegant way to specify a property in RTLola is the flight phase detection specification. Before, this dynamic region was

```
input int stateID_SC
input bool onGround

const int eStateSC_Landing := 13

output bool land_cmd := stateID_SC = eStateSC_Landing

output bool missed_land := land_cmd[20sec,or,false] & !onGround[20sec,or,false]

trigger missed_land
```

Listing 14: Landing time bound in RTLola

```
input double vel_x, vel_y, vel_z

output double velocity := sqrt ( vel_x^2.0 + vel_y^2.0 + vel_z^2.0 )

const double velocity_bound := 1.0

output double diff_velocity := difference(velocity[3s,max,0.0],
                                          velocity[3s,min,velocity])

output bool hover := diff_velocity < velocity_bound

output bool frozen_velocity := vel_x[0.2s,allequal,true]
                             & vel_y[0.2s,allequal,true]
                             & vel_z[0.2s,allequal,true]
```

Listing 15: Flight Phase Detection in RTLola

specificed in Listing 11 in lines 13–25. Here in Listing 15, we specify a similar property in RTLola and use window operators to determine the minimum and maximum within the last three seconds.

Additionally, we add a monitor property `frozen_velocity` to check for a freeze in the input signals if they remain unchanged in the last 0.2 seconds.

Dynamic Waypoints This specification example in Listing 16 uses both the real-time and data parameterization features to check whether a dynamically extendable list of waypoints is reached by the aircraft. Such a specification would be interesting to monitor as a liveness property in the mission manager component, since the ground station can extend the mission plan at any point in time.

Note that due to the dynamic nature of the length of the list of waypoints, and the fact that int cannot be bounded a priori, this specification is not expressible in classic Lola. For a time-triggered system with a known input frequency, it is possible to express time-bounded reachability properties in classic Lola, as we have seen before, but for a more general class of systems, the expressivity RTLola is needed to express the time bound for each waypoint.

As inputs, the specification needs the current timestamp as well as the current position of the aircraft and the incoming waypoint with its time-to-reach estimate. We use a counter (`wp_count`) as an index to the waypoint list (stored in the default values of `lookup`) to parameterize the individual statistics for the timing of the waypoints.

All the specifications of this section are efficiently monitorable.

```
timeinput int ts
input (double,double,double) waypoint
input (double,double,double) pos
input int estimated_time

trigger any( wp_reached_c[-1,0] > wp_reached_c )

output int wp_reached_c <int c>
        invoke: wp_count
        terminate: wp_reached
                        := if wp_reached(c) then c else -1

output bool wp_reached <int c>
        invoke: wp_count
                      := pos = lookup(c)

output (int,int) reach_inv := (wp_count, estimated_time)

output bool wp_reached_intime <int c, int est_time>
        invoke: reach_inv
                      := ( wp_start_time(c) + est_time ) < ts

output int wp_start_time <int c>
        invoke: wp_count
                      := if !(wp_end_time(c-1)[0,5] = -1)
                            then ts
                            else wp_start_time(c)[-1,-1]

output int wp_end_time <int c>
        invoke: wp_count
                      := if wp_reached(c)
                            then ts
                            else wp_end_time(c)[-1,-1]

output int wp_count    := if (waypoint[-1,(0.0,0.0,0.0)] != waypoint)
                           then wp_count[-1,0] + 1
                           else wp_count[-1,0]

output (double,double,double) lookup <int c>
        invoke: wp_count
                      := lookup(c)[-1,waypoint]
```

Listing 16: Dynamic Waypoints in RTLola. Note that the last line requires a slight extension to the semantics to allow dynamic default values for stream accesses.

6.5 Monitor Integration

Ideally, to achieve *non-interference* (i.e. unobtrusiveness in the sense of [61]), a monitor would be integrated into the system such that it does not have an impact on the system execution apart from the monitor outputs. One way to realize this is to use an existing hardware interface, such as a communication bus, to add the monitor as a passive bus receiver. As the ARTIS software platform did not have such a bus with the needed input information, we used the pre-existing logging interface responsible for writing log data and added function calls to the monitoring interface. Thus, we add one instrumentation point per system component. The ARTIS software platform is implemented in C and C++. Our Lola monitor itself is realized in a multi-threaded C implementation, and the interface and integration is realized in C++. For stream accesses, we use a ring buffer with array indexing as the main data structure. Due to the real-time requirements, the monitor only allocates memory once at startup for effiently monitorable specifications. The monitor threads use shared memory for stream value computation and monitor output. Since we could not change the system design to achieve non-interference by design, we instead measured the impact of the monitor on the system performance and in particular the timing. The specific experiments and results are described in Section 6.6.2.

6.6 Experiments

The primary goal of the experiments was to establish the suitability of the prototype tool Lola in offline mode for specification development and regression testing, and in online mode as a real-time capable monitor in this application context.

6.6.1 Offline Trace Analysis

Here, we present experimental data where Lola was used as an offline monitor on previously recorded flight data. Since there are regular test flights with the different aircraft and an extensive simulation environment, a large number of flight data traces were readily available. The offline mode was also used during specification development to debug specifications or to re-identify already known issues in certain test flights. Using an additional tool to automate the execution of Lola on a large set of traces, we used the offline mode to derive signal thresholds and boundary values from these traces to obtain initial estimates for the constant values and offsets in the specifications.

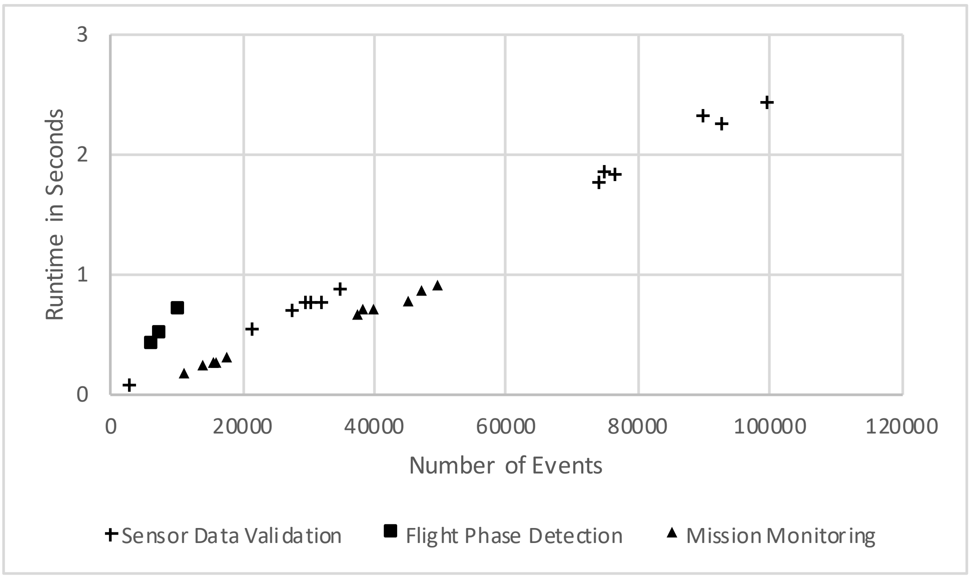


Figure 6.4: Runtime results for offline trace analysis experiments for a superset of the specifications from Section 6.3 with input data traces of varying length.

In offline mode, the command-line binary receives as arguments one or more Lola specifications and the log data file to be processed. As an optional parameter, an output file for the monitor output may be set. These experiments were run on a laptop with a 2.6GHz dual-core i5 processor with 8GB RAM, where the input log data files were on the internal SSD. The runtime results in seconds for log data files of different length are shown in Figure 6.4. The overall memory consumption was below 1.5 MB. The actual flight time covered in these traces generated from the simulation environment was up to fifteen minutes. The specifications used here are the ones presented in Section 6.3.

The runtime results displayed in Figure 6.4 show that our implementation is capable of processing reasonably-sized log data files within seconds. Also interesting is the relative comparison between the three types of specifications: the specification for flight phase detection is the most computationally complex of the three.

6.6.2 Hardware-in-the-loop Online Monitoring

The experiments in this section were devised to investigated the performance impact of monitoring the full set of Lola specifications in an online manner on the actual flight hardware. The flight hardware/software setup on the hardware-in-

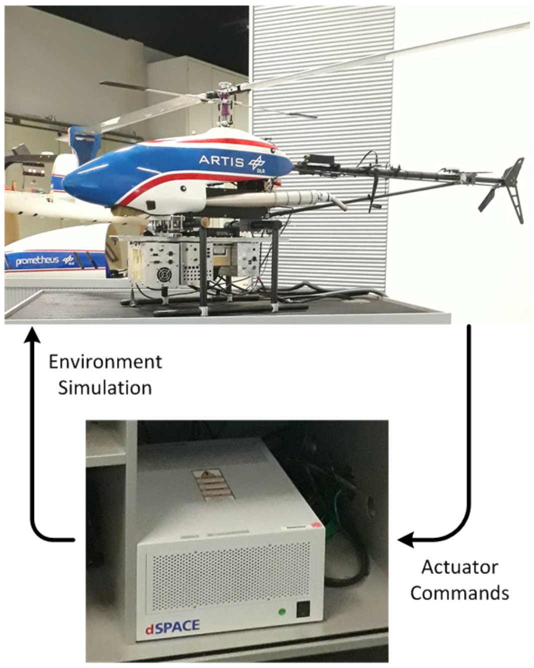


Figure 6.5: Hardware-in-the-loop test rig used for online monitoring experiments.

the-loop (HIL) test rig consists of a flight computer with an Intel Pentium Processor with a 1.8GHz single-core CPU and 1GB RAM, which runs a Unix-based RTOS. The test rig, shown in Figure 6.5, consists of real UAS aircraft, where the actuators are plugged into a closed-loop real-time environment simulation of the flight environment, which in turn delivers simulated sensor values to the aircraft, based on a model of the flight dynamics and a pre-computed world model and flight situation. Visualizations of the flight situations are depicted in Figure 6.6. As a first check, the actual flight paths of the aircraft with and without monitoring were compared via visual inspection. In order to measure the performance impact of the monitor on the system, the monitors were gradually introduced. We measured the average frequency of the main system loop, computed from the timestamped event data, either online through the Lola specification for the components navigation filter, flight control and mission manager, or through a post-analysis of the log data for the other components. The results are displayed in Figure 6.7. These results suggest that the timing behavior of the system is not affected and thus, the current implementation is sufficiently fast to monitor all the specifications online on the aircraft non-intrusively, even with the limited computing power available.

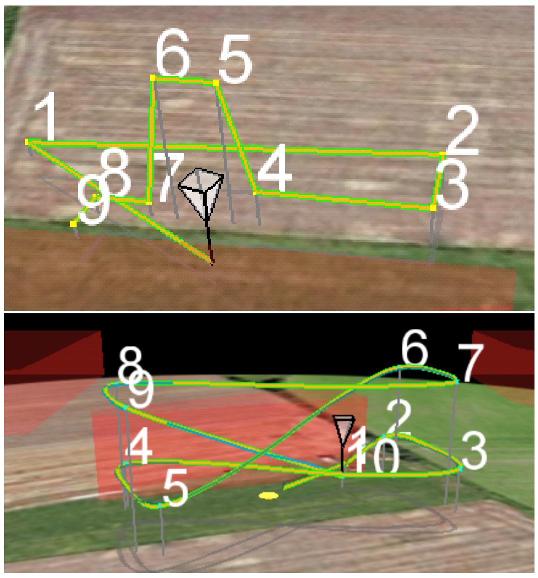


Figure 6.6: Visualizations of the missions flown in the online monitoring experiments. The top picture shows a hover-to simulation, the bottom picture a fly-to simulation.

monitor	avg. freq. (Hz)			(post-analysis) avg. freq. (Hz)			
Module	nav	ctrl	mgr	gps-p	gps-v	imu	mgn
No Monitor	*50.0*	*50.0*	*50.0*	20.0	20.0	100.0	10.0
nav_monitor	**50.0**	*50.0*	*50.0*	20.0	20.0	100.0	10.0
nav_monitor	**50.0**	-	*50.0*	20.0	20.0	100.0	10.0
ctrl_monitor	-	**50.0**	-	20.0	20.0	100.0	10.0
nav_monitor	**50.0**	-	-	20.0	20.0	100.0	10.0
ctrl_monitor	-	**50.0**	-	20.0	20.0	100.0	10.0
mgr_monitor	-	-	**50.0**	20.0	20.0	100.0	10.0

Figure 6.7: Overview of the results from online monitoring experiments with the specifications from Section 6.3. The columns identify system components, for which the frequency was measured with and without monitoring. The first row denotes the baseline without any monitoring. In the next sections, more monitors are added with each row. Frequency values in bold have been determined online by Lola, the others in a post-analysis of the respective component log files.

7 Conclusion

This thesis presents three improvements to the expressibility of specification languages for runtime monitoring and a case study to evaluate stream-based specification languages for autonomous unmanned aerial systems. Parametric and real-time extensions to specification languages support a succinct way of specifying monitoring properties ranging from network monitoring, data analysis, and safeguarding of autonomous unmanned aerial systems.

For parametric linear-time temporal logic, we have described the impact of step bound parameters for the monitoring problem and identified unambiguity as the deciding complexity factor for memory-efficient implementations in Chapter 2. Future dependencies, a standard feature in languages for design verification, lead to high complexity for online monitoring. Future dependencies present a trade-off for online monitoring: If future dependencies can be eliminated from the specification language, significantly better complexity results can be established.

We have introduced the the stream-based specification language Lola 2.0 in Chapter 4, which extends Lola with data parameters. This allows us to treat individual substreams in local monitors and used bounded instantiation to support new classes of specifications. The concept of efficient monitorability, originally defined for Lola, has been adapted for Lola 2.0 and makes it possible to provide a guarantee for the bounded use of resources in the online monitoring implementation. The practical usage of the language was demonstrated for network monitoring tasks.

For a real-time system model of stream monitoring, we showed in the specification language RTLola that sliding windows with efficient aggregation functions still allow resource bounded monitoring for this important class of systems in Chapter 5. The necessary conditions for the specification to allow the resource bounded monitoring can be analyzed via a static analysis. In the experiments, we have obtained empirical evidence that the language is capable of expressing data analysis tasks and that the monitor implementation performs well on large data sets.

In a case study for our benchmark application of autonomous UAS, we have demonstrated in Chapter 6 that an extension of Lola and RTLola is sufficiently expressive to express important functional safety properties of these unmanned aircraft systems. The memory guarantees required by the application could be satisfied statically for the specifications. We have additionally evaluated the monitor performance in an experimental evaluation on the flight software of the DLR UAS fleet and described the impact of monitoring with formal specification languages on the system development process in a realistic application environment. The formalization efforts are able to document the interface assumptions of components and can also be used for the offline analysis of flight log files for problem analysis and debugging.

For the field of runtime verification, this thesis demonstrates the possibility of enhancing the expressivity of specification languages with parameters, provides realistic stream-based specifications for autonomous unmanned aerial systems, and shows that resource guarantees can be provided at design time for runtime monitor implementations which are useful in important embedded applications.

Outlook The benchmark application of unmanned aircraft systems with autonomy has validated the general concept of stream-based specification languages for the properties needed and has inspired further language design. Continuing this language design to further tailor stream-based specification languages to the application described here, but also to related application domains such as autonomous cars, is an important further step towards usable monitor specification languages.

Since the application of runtime monitoring into regulated domains leads to the situation where the monitor itself has to be considered as a safety-critical component, further work is needed to certify the integrity of the monitor implementation. The interpreter approach used for the implementations in the experiments of this thesis would lead to extensive verification effort for the qualification of the implementations themselves. An easier path to monitor qualification may be to switch to compiling the specification to an equivalent C program with an accompanying correctness certificate, and use existing qualified tool chains to compile the C program as well as methods from translation validation to independently check the compilation steps and the certificate. A direct compilation to a hardware description language such as VHDL and a monitor realization on configurable hardware such as an FPGA is also a promising forward direction.

For complex sensor-processing systems with autonomy, runtime monitoring with concise and readable formal specifications can play an important role to monitor system- and component-level functional safety properties. The monitor as a trusted component ensures that the system behavior follows the specification and can increase user and regulator trust.

Bibliography

[1] Florian-Michael Adolf, Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Christoph Torens. "Stream Runtime Monitoring on UAS". In: *Proceedings of Runtime Verification - 17th International Conference, RV 2017*. Vol. 10548. Lecture Notes in Computer Science. Springer, 2017, pp. 33–49. DOI: 10.1007/978-3-319-67531-2_3. URL: https://doi.org/10.1007/978-3-319-67531-2_3.

[2] Rajeev Alur, Kousha Etessami, Salvatore La Torre, and Doron A. Peled. "Parametric Temporal Logic for 'Model Measuring'". In: *Proceedings of Automata, Languages and Programming, 26th International Colloquium, ICALP'99*. Vol. 1644. Lecture Notes in Computer Science. Springer, 1999, pp. 159–168. DOI: 10.1007/3-540-48523-6_13. URL: http://dx.doi.org/10.1007/3-540-48523-6_13.

[3] Rajeev Alur, Kousha Etessami, Salvatore La Torre, and Doron A. Peled. "Parametric temporal logic for 'model measuring'". In: *ACM Trans. Comput. Log.* 2.3 (2001), pp. 388–407. DOI: 10.1145/377978.377990. URL: http://doi.acm.org/10.1145/377978.377990.

[4] Rajeev Alur, Dana Fisman, and Mukund Raghothaman. "Regular Programming for Quantitative Properties of Data Streams". In: *Proceedings of Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016*. Vol. 9632. Lecture Notes in Computer Science. Springer, 2016, pp. 15–40. DOI: 10.1007/978-3-662-49498-1_2. URL: https://doi.org/10.1007/978-3-662-49498-1%5C_2.

[5] Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, and David E. Rydeheard. "Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors". In: *Proceedings of FM 2012: Formal Methods - 18th International Symposium*. Vol. 7436. Lecture Notes in Computer Science. Springer, 2012, pp. 68–84. DOI: 10.1007/978-3-642-32759-9_9. URL: https://doi.org/10.1007/978-3-642-32759-9%5C_9.

[6] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. “Rule-Based Runtime Verification”. In: *Proceedings of Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004*. Vol. 2937. Lecture Notes in Computer Science. Springer, 2004, pp. 44–57. DOI: 10.1007/978-3-540-24622-0_5. URL: https://doi.org/10.1007/978-3-540-24622-0%5C_5.

[7] Howard Barringer, David E. Rydeheard, and Klaus Havelund. “Rule Systems for Run-time Monitoring: from Eagle to RuleR”. In: *J. Log. Comput.* 20.3 (2010), pp. 675–706. DOI: 10.1093/logcom/exn076. URL: https://doi.org/10.1093/logcom/exn076.

[8] David A. Basin, Felix Klaedtke, and Samuel Müller. “Policy Monitoring in First-Order Temporal Logic”. In: *Proceedings of Computer Aided Verification, 22nd International Conference, CAV 2010*. Vol. 6174. Lecture Notes in Computer Science. Springer, 2010, pp. 1–18. DOI: 10.1007/978-3-642-14295-6_1. URL: https://doi.org/10.1007/978-3-642-14295-6%5C_1.

[9] David A. Basin, Felix Klaedtke, Samuel Müller, and Eugen Zalinescu. “Monitoring Metric First-Order Temporal Properties”. In: *J. ACM* 62.2 (2015), 15:1–15:45. DOI: 10.1145/2699444. URL: http://doi.acm.org/10.1145/2699444.

[10] David Basin, Bhargav Nagaraja Bhatt, and Dmitriy Traytel. “Almost Event-Rate Independent Monitoring of Metric Temporal Logic”. In: *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems: 23rd International Conference, TACAS 2017*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 94–112. ISBN: 978-3-662-54580-5. DOI: 10.1007/978-3-662-54580-5_6. URL: http://dx.doi.org/10.1007/978-3-662-54580-5_6.

[11] David Basin, Felix Klaedtke, and Eugen Zălinescu. “Greedily Computing Associative Aggregations on Sliding Windows”. In: *Inf. Process. Lett.* 115.2 (Feb. 2015), pp. 186–192. ISSN: 0020-0190. DOI: 10.1016/j.ipl.2014.09.009. URL: http://dx.doi.org/10.1016/j.ipl.2014.09.009.

[12] David Basin, Dmitriy Traytel, and Srđan Krstić. *Aerial: Almost Event-Rate Independent Algorithms for Monitoring Metric Regular Properties*. 2017. URL: https://www21.in.tum.de/~traytel/papers/rvcubes17-aerial_tool/index.html.

[13] Andreas Klaus Bauer and Yliès Falcone. “Decentralised LTL Monitoring”. In: *Proceedings of FM 2012: Formal Methods - 18th International Symposium*. Vol. 7436. Lecture Notes in Computer Science. Springer, 2012, pp. 85–100. DOI: 10.1007/

978-3-642-32759-9_10. URL: https://doi.org/10.1007/978-3-642-32759-9%5C_10.

[14] Andreas Bauer, Rajeev Goré, and Alwen Tiu. "A First-Order Policy Language for History-Based Transaction Monitoring". In: *Proceedings of Theoretical Aspects of Computing - ICTAC 2009, 6th International Colloquium*. Vol. 5684. Lecture Notes in Computer Science. Springer, 2009, pp. 96–111. DOI: 10.1007/978-3-642-03466-4_6. URL: https://doi.org/10.1007/978-3-642-03466-4%5C_6.

[15] Andreas Bauer, Jan-Christoph Küster, and Gil Vegliach. "From Propositional to First-Order Monitoring". In: *Proceedings of Runtime Verification - 4th International Conference, RV 2013*. Vol. 8174. Lecture Notes in Computer Science. Springer, 2013, pp. 59–75. DOI: 10.1007/978-3-642-40787-1_4. URL: https://doi.org/10.1007/978-3-642-40787-1%5C_4.

[16] Andreas Bauer, Martin Leucker, and Christian Schallhart. "Comparing LTL Semantics for Runtime Verification". In: *J. Log. Comput.* 20.3 (2010), pp. 651–674. DOI: 10.1093/logcom/exn075. URL: https://doi.org/10.1093/logcom/exn075.

[17] Andreas Bauer, Martin Leucker, and Christian Schallhart. "Runtime Verification for LTL and TLTL". In: *ACM Trans. Softw. Eng. Methodol.* 20.4 (2011), 14:1–14:64. DOI: 10.1145/2000799.2000800. URL: https://doi.org/10.1145/2000799.2000800.

[18] Gérard Berry. "The foundations of Esterel". In: *Proof, Language, and Interaction, Essays in Honour of Robin Milner*. The MIT Press, 2000, pp. 425–454.

[19] Laura Bozzelli and César Sánchez. "Foundations of Boolean stream runtime verification". In: *Theor. Comput. Sci.* 631 (2016), pp. 118–138. DOI: 10.1016/j.tcs.2016.04.019. URL: https://doi.org/10.1016/j.tcs.2016.04.019.

[20] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. "Lustre: A Declarative Language for Programming Synchronous Systems". In: *Proceedings of Conference the Fourteenth Annual ACM Symposium on Principles of Programming Languages, POPL 1987*. ACM Press, 1987, pp. 178–188. DOI: 10.1145/41625.41641. URL: http://doi.acm.org/10.1145/41625.41641.

[21] Brian Caswell, James C. Foster, Ryan Russell, Jay Beale, and Jeffrey Posluns. *Snort 2.0 Intrusion Detection*. Syngress Publishing, 2003. ISBN: 1931836744.

[22] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 2001. ISBN: 978-0-262-03270-4. URL: `http://books.google.de/books?id=Nmc4wEaLXFEC`.

[23] Lukas Convent, Sebastian Hungerecker, Martin Leucker, Torben Scheffel, Malte Schmitz, and Daniel Thoma. "TeSSLa: Temporal Stream-Based Specification Language". In: *Proceedings of Formal Methods: Foundations and Applications - 21st Brazilian Symposium, SBMF 2018*. Vol. 11254. Lecture Notes in Computer Science. Springer, 2018, pp. 144–162. DOI: `10.1007/978-3-030-03044-5\_10`. URL: `https://doi.org/10.1007/978-3-030-03044-5%5C_10`.

[24] Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, and Zohar Manna. "LOLA: Runtime Monitoring of Synchronous Systems". In: *Proceedings of 12th International Symposium on Temporal Representation and Reasoning (TIME 2005)*. IEEE Computer Society, 2005, pp. 166–174. DOI: `10.1109/TIME.2005.26`. URL: `https://doi.org/10.1109/TIME.2005.26`.

[25] Jennifer A. Davis, Matthew Clark, Darren D. Cofer, Aaron Fifarek, Jacob Hinchman, Jonathan Hoffman, Brian Hulbert, Steven P. Miller, and Lucas Wagner. "Study on the Barriers to the Industrial Adoption of Formal Methods". In: *Proceedings of FMICS'13*. 2013, pp. 63–77.

[26] Evan T. Dill, Steven D. Young, and Kelly J. Hayhurst. "SAFEGUARD: An assured safety net technology for UAS". In: *Proceedings of 2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*. IEEE, Sept. 2016. DOI: `10.1109/dasc.2016.7778009`. URL: `https://doi.org/10.1109/dasc.2016.7778009`.

[27] Alexandre Donzé, Thomas Ferrère, and Oded Maler. "Efficient Robust Monitoring for STL". In: *Proceedings of Computer Aided Verification - 25th International Conference, CAV 2013*. Vol. 8044. Lecture Notes in Computer Science. Springer, 2013, pp. 264–279. DOI: `10.1007/978-3-642-39799-8\_19`. URL: `https://doi.org/10.1007/978-3-642-39799-8%5C_19`.

[28] Alexandre Donzé and Oded Maler. "Robust Satisfaction of Temporal Logic over Real-valued Signals". In: *Proceedings of the 8th International Conference on Formal Modeling and Analysis of Timed Systems*. FORMATS'10. Klosterneuburg, Austria: Springer-Verlag, 2010, pp. 92–106. ISBN: 3-642-15296-1. URL: `http://dl.acm.org/citation.cfm?id=1885174.1885183`.

[29] Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout. “Reasoning with Temporal Logic on Truncated Paths”. In: *Proceedings of Computer Aided Verification, 15th International Conference, CAV 2003*. Vol. 2725. Lecture Notes in Computer Science. Springer, 2003, pp. 27–39. DOI: `10.1007/978-3-540-45069-6_3`. URL: `https://doi.org/10.1007/978-3-540-45069-6_3`.

[30] European Aviation Safety Agency (EASA). *Advance Notice of Proposed Amendment 2015-10, Introduction of a regulatory framework for the operation of drones*. 2015.

[31] European Aviation Safety Agency (EASA). *Concept of Operations for Drones, A risk based approach to regulation of unmanned aircraft*. 2015.

[32] European Aviation Safety Agency (EASA). *Opinion No 01/2018 Introduction of a regulatory framework for the operation of unmanned aircraft systems in the ‘open’ and ‘specific’ categories*. 2018.

[33] Peter Faymonville, Bernd Finkbeiner, and Doron A. Peled. “Monitoring Parametric Temporal Logic”. In: *Proceedings of Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014*. Vol. 8318. Lecture Notes in Computer Science. Springer, 2014, pp. 357–375. DOI: `10.1007/978-3-642-54013-4_20`. URL: `http://dx.doi.org/10.1007/978-3-642-54013-4_20`.

[34] Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Hazem Torfah. “A Stream-Based Specification Language for Network Monitoring”. In: *Proceedings of Runtime Verification - 16th International Conference, RV 2016*. Vol. 10012. Lecture Notes in Computer Science. Springer, 2016, pp. 152–168. DOI: `10.1007/978-3-319-46982-9_10`. URL: `https://doi.org/10.1007/978-3-319-46982-9_10`.

[35] Peter Faymonville, Bernd Finkbeiner, Maximilian Schwenger, and Hazem Torfah. “Real-time Stream-based Monitoring”. In: *CoRR* abs/1711.03829 (2017). arXiv: `1711.03829`. URL: `http://arxiv.org/abs/1711.03829`.

[36] Bernd Finkbeiner and Lars Kuhtz. “Monitor Circuits for LTL with Bounded and Unbounded Future”. In: *Proceedings of Runtime Verification, 9th International Workshop, RV 2009*. Vol. 5779. Lecture Notes in Computer Science. Springer, 2009, pp. 60–75. DOI: `10.1007/978-3-642-04694-0_5`. URL: `https://doi.org/10.1007/978-3-642-04694-0_5`.

[37] Bernd Finkbeiner and Henny Sipma. "Checking Finite Traces Using Alternating Automata". In: *Formal Methods in System Design* 24.2 (2004), pp. 101–127. doi: 10.1023/B:FORM.0000017718.28096.48. url: https://doi.org/10.1023/B:FORM.0000017718.28096.48.

[38] Johannes Geist, Kristin Y. Rozier, and Johann Schumann. "Runtime Observer Pairs and Bayesian Network Reasoners On-board FPGAs: Flight-Certifiable System Health Management for Embedded Systems". In: *Proceedings of Runtime Verification - 5th International Conference, RV 2014*. Vol. 8734. Lecture Notes in Computer Science. Springer, 2014, pp. 215–230. url: http://dx.doi.org/10.1007/978-3-319-11164-3_18.

[39] Dimitra Giannakopoulou and Klaus Havelund. "Automata-Based Verification of Temporal Properties on Running Programs". In: *Proceedings of 16th IEEE International Conference on Automated Software Engineering (ASE 2001)*. IEEE Computer Society, 2001, pp. 412–416. doi: 10.1109/ASE.2001.989841. url: https://doi.org/10.1109/ASE.2001.989841.

[40] Felipe Gorostiaga and César Sánchez. "Striver: Stream Runtime Verification for Real-Time Event-Streams". In: *Proceedings of Runtime Verification - 18th International Conference, RV 2018*. Vol. 11237. Lecture Notes in Computer Science. Springer, 2018, pp. 282–298. doi: 10.1007/978-3-030-03769-7_16. url: https://doi.org/10.1007/978-3-030-03769-7%5C_16.

[41] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Springer US, 1993. doi: 10.1007/978-1-4757-2231-4. url: https://doi.org/10.1007/978-1-4757-2231-4.

[42] Klaus Havelund. "Rule-based runtime verification revisited". In: *International Journal on Software Tools for Technology Transfer (STTT)* 17.2 (2015), pp. 143–170. doi: 10.1007/s10009-014-0309-2. url: https://doi.org/10.1007/s10009-014-0309-2.

[43] Ruining He and Julian McAuley. "Ups and Downs: Modeling the Visual Evolution of Fashion Trends with One-Class Collaborative Filtering". In: *Proceedings of the 25th International Conference on World Wide Web, WWW 2016*. 2016, pp. 507–517. doi: 10.1145/2872427.2883037. url: http://doi.acm.org/10.1145/2872427.2883037.

[44] ISO. *ISO 26262: Road vehicles – Functional safety*. Norm ISO 26262. International Organization for Standardization, 2011.

[45] Dongyun Jin, Patrick O'Neil Meredith, Choonghwan Lee, and Grigore Rosu. "JavaMOP: Efficient parametric runtime monitoring framework". In: *Proceedings of 34th International Conference on Software Engineering, ICSE 2012*. IEEE Computer Society, 2012, pp. 1427–1430. DOI: `10.1109/ICSE.2012.6227231`. URL: `https://doi.org/10.1109/ICSE.2012.6227231`.

[46] Joint Authorities for Rulemaking of Unmanned Systems (JARUS). *JARUS Guidelines on Specific Operations Risk Assessment (SORA)*. Joint Authorities for Rulemaking of Unmanned Systems, 2017.

[47] Ron Koymans. "Specifying Real-Time Properties with Metric Temporal Logic". In: *Real-Time Systems* 2.4 (1990), pp. 255–299. DOI: `10.1007/BF01995674`. URL: `https://doi.org/10.1007/BF01995674`.

[48] *Lectures on Runtime Verification - Introductory and Advanced Topics*. Vol. 10457. Lecture Notes in Computer Science. Springer, 2018.

[49] Edward A. Lee and Sanjit Seshia. *Introduction to Embedded Systems, A Cyber-Physical Systems Approach*. Second Edition. MIT Press, 2017. ISBN: 978-0-262-53381-2. URL: `http://chess.eecs.berkeley.edu/pubs/710.html`.

[50] Martin Leucker, César Sánchez, Torben Scheffel, Malte Schmitz, and Alexander Schramm. "TeSSLa: runtime verification of non-synchronized real-time streams". In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018*. ACM, 2018, pp. 1925–1933. DOI: `10.1145/3167132.3167338`. URL: `http://doi.acm.org/10.1145/3167132.3167338`.

[51] Martin Leucker and Christian Schallhart. "A brief account of runtime verification". In: *J. Log. Algebr. Program.* 78.5 (2009), pp. 293–303. DOI: `10.1016/j.jlap.2008.08.004`. URL: `https://doi.org/10.1016/j.jlap.2008.08.004`.

[52] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. "No pane, no gain: efficient evaluation of sliding-window aggregates over data streams". In: *SIGMOD Record* 34.1 (2005), pp. 39–44. DOI: `10.1145/1058150.1058158`. URL: `http://doi.acm.org/10.1145/1058150.1058158`.

[53] Monika Maidl. "The Common Fragment of CTL and LTL". In: *Proceedings of 41st Annual Symposium on Foundations of Computer Science, FOCS 2000*. IEEE Computer Society, 2000, pp. 643–652. DOI: `10.1109/SFCS.2000.892332`. URL: `https://doi.org/10.1109/SFCS.2000.892332`.

[54] Oded Maler and Dejan Nickovic. "Monitoring Temporal Properties of Continuous Signals". In: *Proceedings of Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, FORMATS 2004*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 152–166. ISBN: 978-3-540-30206-3. DOI: 10.1007/978-3-540-30206-3_12. URL: http://dx.doi.org/10.1007/978-3-540-30206-3_12.

[55] Oded Maler, Dejan Nickovic, and Amir Pnueli. "From MITL to Timed Automata". In: *Proceedings of Formal Modeling and Analysis of Timed Systems, 4th International Conference, FORMATS 2006*. Vol. 4202. Lecture Notes in Computer Science. Springer, 2006, pp. 274–289. DOI: 10.1007/11867340_20. URL: https://doi.org/10.1007/11867340%5C_20.

[56] Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. "Copilot: A Hard Real-Time Runtime Monitor". In: *Proceedings of Runtime Verification: First International Conference, RV 2010*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 345–359. ISBN: 978-3-642-16612-9. DOI: 10.1007/978-3-642-16612-9_26. URL: http://dx.doi.org/10.1007/978-3-642-16612-9_26.

[57] Lee Pike, Sebastian Niller, and Nis Wegmann. "Runtime Verification for Ultra-Critical Systems". In: *Proceedings of RV*. Vol. 7186. Lecture Notes in Computer Science. Springer, 2011, pp. 310–324.

[58] Radio Technical Commission for Aeronautics (RTCA). *DO-178C/ED-12C Software Considerations in Airborne Systems and Equipment Certification*. 2011.

[59] Radio Technical Commission for Aeronautics (RTCA). *DO-333/ED-216 Formal Methods Supplement to DO-178C and DO-278A*. 2011.

[60] Giles Reger, Helena Cuenca Cruz, and David E. Rydeheard. "MarQ: Monitoring at Runtime with QEA". In: *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015*. Vol. 9035. Lecture Notes in Computer Science. Springer, 2015, pp. 596–610. DOI: 10.1007/978-3-662-46681-0_55. URL: https://doi.org/10.1007/978-3-662-46681-0%5C_55.

[61] Thomas Reinbacher, Kristin Yvonne Rozier, and Johann Schumann. "Temporal-Logic Based Runtime Observer Pairs for System Health Management of Real-Time Systems". In: *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2014*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 357–372. ISBN: 978-3-642-54862-8.

[62] Grigore Rosu and Feng Chen. “Semantics and Algorithms for Parametric Monitoring”. In: *Logical Methods in Computer Science* 8.1 (2012). DOI: `10.2168/LMCS-8(1:9)2012`. URL: `https://doi.org/10.2168/LMCS-8(1:9)2012`.

[63] Kristin Y. Rozier. “Specification: The Biggest Bottleneck in Formal Methods and Autonomy”. In: *Proceedings of Verified Software. Theories, Tools, and Experiments, VSTTE 2016*. Springer International Publishing, 2016, pp. 8–26. ISBN: 978-3-319-48869-1.

[64] IEC SC 65A. *Functional safety of electrical/electronic/programmable electronic safety-related systems*. Tech. rep. IEC 61508. The International Electrotechnical Commission, 1998.

[65] Sebastian Schirmer. “Runtime Monitoring with Lola”. Master’s Thesis. Saarland University, 2016.

[66] Sebastian Schirmer, Christoph Torens, and Florian Adolf. “Formal Monitoring of Risk-based Geofences”. In: *Proceedings of 2018 AIAA Information Systems-AIAA Infotech @ Aerospace*. American Institute of Aeronautics and Astronautics, Jan. 2018. DOI: `10.2514/6.2018-1986`. URL: `https://doi.org/10.2514/6.2018-1986`.

[67] Johann Schumann, Patrick Moosbrugger, and Kristin Y. Rozier. “R2U2: Monitoring and Diagnosis of Security Threats for Unmanned Aerial Systems”. In: *Proceedings of Runtime Verification - 6th International Conference, RV 2015*. Vol. 9333. Lecture Notes in Computer Science. Springer, 2015, pp. 233–249. URL: `http://dx.doi.org/10.1007/978-3-319-23820-3_15`.

[68] Volker Stolz. “Temporal Assertions with Parametrised Propositions”. In: *Proceedings of Runtime Verification, 7th International Workshop, RV 2007*. Vol. 4839. Lecture Notes in Computer Science. Springer, 2007, pp. 176–187. DOI: `10.1007/978-3-540-77395-5\_15`. URL: `https://doi.org/10.1007/978-3-540-77395-5%5C_15`.

[69] Volker Stolz and Eric Bodden. “Temporal Assertions using AspectJ”. In: *Electronic Notes in Theoretical Computer Science* 144.4 (May 2006), pp. 109–124. DOI: `10.1016/j.entcs.2006.02.007`. URL: `https://doi.org/10.1016/j.entcs.2006.02.007`.

[70] D. E. Swihart, A. F. Barfield, E. M. Griffin, R. C. Lehmann, S. C. Whitcomb, B. Flynn, M. A. Skoog, and K. E. Processor. “Automatic Ground Collision Avoidance System design, integration, flight test”. In: *IEEE Aerospace and Electronic*

Systems Magazine 26.5 (May 2011), pp. 4–11. ISSN: 0885-8985. DOI: `10.1109/MAES.2011.5871385`.

[71] Christoph Torens and Florian-Michael Adolf. "Automated Verification and Validation of an Onboard Mission Planning and Execution System for UAVs". In: *Proceedings of AIAA Infotech@Aerospace (I@A) Conference*. Boston, MA, 19.-22. Aug 2013. DOI: `doi:10.2514/6.2013-4564`. URL: `http://dx.doi.org/10.2514/6.2013-4564`.

[72] Christoph Torens and Florian-Michael Adolf. "Software Verification Considerations for the ARTIS Unmanned Rotorcraft". In: *Proceedings of 51st AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition*. American Institute of Aeronautics and Astronautics, Jan. 2013. ISBN: 978-1-62410-181-6. DOI: `10.2514/6.2013-593`. URL: `http://dx.doi.org/10.2514/6.2013-593`.

[73] Christoph Torens and Florian-Michael Adolf. "Using Formal Requirements and Model-Checking for Verification and Validation of an Unmanned Rotorcraft". In: *American Institute of Aeronautics and Astronautics, AIAA Infotech @ Aerospace, AIAA SciTech* (May 2015). DOI: `doi:10.2514/6.2015-1645`. URL: `http://dx.doi.org/10.2514/6.2015-1645`.

[74] Christoph Torens, Florian Adolf, Peter Faymonville, and Sebastian Schirmer. "Towards Intelligent System Health Management using Runtime Monitoring". In: *Proceedings of AIAA Information Systems-AIAA Infotech @ Aerospace*. American Institute of Aeronautics and Astronautics (AIAA), Jan. 2017. DOI: `10.2514/6.2017-0419`. URL: `https://doi.org/10.2514%2F6.2017-0419`.

[75] Bundesministerium für Verkehr und digitale Infrastruktur. *Verordnung zur Regelung des Betriebs von unbemannten Fluggeräten*. Bundesgesetzblatt Jahrgang 2017 Teil I Nr. 17. June 2017.

[76] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. "Apache Spark: a unified engine for big data processing". In: *Commun. ACM* 59.11 (2016), pp. 56–65. DOI: `10.1145/2934664`. URL: `http://doi.acm.org/10.1145/2934664`.