Statically-Analyzed Stream Monitoring for Cyber-Physical Systems

Dissertation submitted towards the degree
Doctor of Engineering
of the Faculty of Mathematics and Computer Science
of Saarland University

Maximilian Schwenger

Saarbrücken, 2022

Dean:	Prof. Dr. Jürgen Steimle
Chair of the Committee:	Prof. Martina Maggio, Ph.D.
Reviewers:	Prof. Bernd Finkbeiner, Ph.D.
	Prof. César Sánchez, Ph.D.
	Prof. Borzoo Bonakdarpour, Ph.D.
Academic Assistant:	Hadar Frenkel, Ph.D.
Day of the Colloquium:	June 13, 2022

Abstract

Cyber-physical systems are digital systems interacting with the physical world. Even though this induces an inherent complexity, they are responsible for safety-critical tasks like governing nuclear power plants or controlling autonomous vehicles. To preserve trust into the safety of such systems, this thesis presents a runtime verification approach designed to generate trustworthy monitors from a formal specification. These monitors are responsible for observing the cyber-physical system during runtime and ensuring its safety. As underlying language, I present the asynchronous real-time specification language RTLola. It contains primitives for arithmetic properties and grants precise control over the timing of the monitor. With this, it enables specifiers to express properties relevant to cyber-physical systems. The thesis further presents a static analysis that identifies inconsistencies in the specification and provides insights into the dynamic behavior of the monitor. As a result, the resource consumption of the monitor becomes predictable. The generation of the monitor produces either a hardware description synthesizable onto programmable hardware, or Rust code with verification annotation. These annotations allow for proving the correctness of the monitor with respect to the semantics of RTLola. Last, I present the construction of a conservative hybrid model of the underlying system using information extracted from the specification. This model enables further verification steps.

Zusammenfassung

Cyber-physische Systeme sind digitale Systeme, die mit der physischen Welt interagieren. Obwohl das zu einer inhärenten Komplexität führt, sind sie verantwortlich für sicherheitskritische Aufgaben wie der Steuerung von Kernkraftwerken oder autonomen Fahrzeugen. Um das Vertrauen in deren Sicherheit zu wahren, präsentiert diese Doktorarbeit einen Ansatz zur Laufzeitverifikation, konzipiert, um vertrauenswürdige Monitore aus einer formalen Spezifikation zu generieren. Diese Monitore sind dafür verantwortlich, das cyber-physische System zur Laufzeit zu überwachen und dessen Sicherheit zu gewährleisten. Als zugrundeliegende Sprache präsentiere ich die asynchrone Echtzeit-Spezifikationssprache RTLola. Sie enthält Primitiven für arithmetische Eigenschaften und gewährt präzise Kontrolle über das Timing des Monitors. Damit wird es Spezifizierenden ermöglicht Eigenschaften auszudrücken, die für Cyber-physische Systeme relevant sind. Weiterhin präsentiert diese Doktorarbeit eine statische Analyse, die Unstimmigkeiten in der Spezifikation identifiziert und Einblicke in das dynamische Verhalten des Monitors liefert. Aufgrund dessen wird der Ressourcenverbrauch des Monitors vorhersehbar. Die Generierung des Monitors erzeugt entweder eine Hardwarebeschreibung, die auf programmierbarer Hardware synthetisiert werden kann, oder Rust Code mit Verifikationsannotationen. Diese Annotationen erlauben es, die Korrektheit des Monitors bezogen auf die Semantik von RTLola zu beweisen. Abschließend präsentiere ich die Konstruktion von einem konservativen hybriden Modell des zugrundeliegenden Systems anhand von Informationen, die aus der Spezifikation gewonnen wurden. Dieses Modell ermöglicht weitere Verifikationsschritte.

Acknowledgements

First, I would like to thank Bernd Finkbeiner for the opportunity to pursue my research in whatever direction struck my interest. Your continuous guidance, support and all these chances to stand on my own feet — even if it meant running head-first into a wall — shaped me into who I am today.

I also want to thank my colleagues at Saarland University, CISPA, and DLR. In particular, thanks to Jan, Niklas, Florian, Sebastian, and Malte. I am incredibly happy to call you my colleagues and friends. Also, a huge thanks goes to Noemi. Not only did we raise a strong family of plants, nobody but you could remain so patient with me after a Monday full of student meetings or the day of an Embedded Systems deadline. Of course, I am also grateful towards the remaining Reactive Systems crew, I will dearly miss discussions and coffee breaks with you all. Moreover, a warm 'Thank you' to the reviewers and members of the committee; much obliged.

Next, I appreciate all the support from outside the office, too. Thanks to Linda and Marco for always having a room and beer ready when I reached your doorstep. Your support means the world to me. I also want to acknowledge how much my parents have done for me throughout the years. Thanks to Julia, Nora, Pascal, Ferdinand, Ben for all the games nights and D&D sessions, and thanks to Timo, Stefan, Jessica. You're awesome! Both thanks and dearest apologies to Aida for always enduring my violin play: it was a great relief to turn off my brain after overexerting it at work.

I also want to acknowledge my remaining co-authors, every temporary or long-term member of the RTLola crew, and every student I had the pleasure to work with. This thesis would be but a fraction of what it is without you.

Also, I thank Thesaurus for making my words good, as well as Ben, Florian, Jan, Julia, Niklas, and Stefan for proofreading, allowing me to blame all remaining typos on you guys! And a special thanks to Malte, nobody but you has the diligence to spot that $\sigma_{\lambda^*,\lambda^*_{\lambda^*}}$ should *obviously* have been $\sigma_{\lambda^*,\lambda_{\lambda^*}}$.

Contents

Chapter 1

Introduction

Long gone are the days in which computers merely transformed data back and forth, isolated from their surroundings. Nowadays, they are connected to the physical world, gathering data about their environment via sensors, and actively influencing it via actuators. This development allows these systems, commonly referred to as Cyber-Physical Systems (CPS), to take over increasingly complex and significant tasks, like governing power plants, stimulating cardiac rhythms, and controlling autonomous aircraft. While this advances humanity and increases quality of life, it also forces developers of such systems to assume great responsibility. In response to this development, experts from both engineering and computer science started to define increasingly intricate safety standards [Rtc11; Iso18; Nas20]. Compliance with these standards is mandatory for safety-critical systems such as aircraft to be approved by independent certification authorities like the European Union Aviation Safety Agency or the Federal Aviation Administration. Not only does this compel developers to implement immediate safety measures such as additional layers of redundancy, it also requires them to follow a strictly controlled development process. This encompasses regular code audits, several independent implementations for single safety-critical components, and thorough documentation. As a result, safety measures need to be effective, *and* their efficacy needs to be convincingly attested, placing special emphasis on reliability and certifiability.

One step in this direction is the employment of online runtime verification techniques [Kim+99; Lee+99; HG05; LS09; BF18]. This entails deployment of an independent component — the *runtime monitor* — solely responsible for observing the system at runtime, assessing information regarding its current state, and judging its safety. The monitor is automatically generated from a specification, i.e., a description of either the desired behavior or potential error scenarios. It both directly and indirectly contributes to the certifiability of the overall system. Since the monitor is an independent component, it per se constitutes an additional, redundant safety measure. Moreover, the certification authority does not need to understand the concrete details of the monitor but rather of the more abstract specification. Yet, to be applicable in real-world systems, a monitoring approach needs to satisfy several criteria specific to CPS monitoring.

For this reason, this thesis identifies two categories of such criteria and presents the language and monitoring toolkit RTLola specifically designed to fulfill these requirements. First, since the CPS is subject to strict resource limitations, so is the monitor. In particular, this resource consumption has to be statically bounded. Second, the underlying specification language needs to be *sufficiently*, but not overly, expressive. This entails that specifiers can express both low-level properties like the validation of single sensors, or cross-validation thereof. At the same time, it needs to capture mission-level properties such as the mean deviation from a pre-planned trajectory. This requires the language to enable the collection of statistical information regarding the performance of the system as a whole. However, this expressiveness must not jeopardize the bounds on the resource consumption, nor must it affect the third criterion: trust. Here, trust is an umbrella term for factors increasing confidence in the effectiveness and correctness of the monitor and its underlying specification. To this end, the specification language has to assist the specifiers, enabling both a manual and an automatic analysis. On the manual end, a language with a comprehensible syntax allows for detecting errors quickly. Additionally, RTLola generates a graph representation of the textual specification. This illustrates the dependency structure between entities occurring in the specification, enabling specifiers to quickly identify unwanted or missing links.

Moreover, an RTLola specification undergoes an automatic static analysis. It first checks and infers types of *streams*. Streams are the basic building blocks of RTLola: input streams represent information from data sources and output streams state how the input data is supposed to be processed. Each stream has a type based on RTLola's intricate type system. In addition to conventional type constraints, the type system examines if the temporal behavior of the monitor is consistent. While the rules for these checks are complex, a positive result increases confidence and the output of the type inference can easily be verified manually, further increasing certifiability and trust. The second check ensures that the specification is sound, i.e., given a fixed execution of the system, there is a unique evaluation model for the specification. The last check determines how much memory the monitor for the specification needs in the worst case. This not only allows it to entirely forgo expensive dynamic memory allocation, which is a common requirement for embedded components, it also allows developers to validate statically whether the available resources suffice, further increasing trust.

Another core contribution of this thesis are realization options for RTLola monitors, one targeting hardware and one targeting software. A compilation to hardware allows monitors to fully utilize the modularity of RTLola specifications via employing a pipelined architecture. This drastically increases throughput of the monitor. Moreover, the compiler generates a hardware description which a commercial off-the-shelve synthesizer realizes onto a hardware board. This process comprises another analysis determining crucial information regarding the performance of the monitor. It includes the consumption of hardware resources like lookup tables and memory cells, plus the idle and peak power consumption. The software compiler, on the other hand, is a *verifying* compiler, i.e., it injects verification annotations into the code. This enables the static verification of the monitor proving that its verdicts agree with the theoretical evaluation model. This, yet again, benefits trust and certifiability.

Evidently, the design of RTLola revolves around providing a reliable runtime monitor approach with an expressive specification language and low, statically determined resource consumption. This renders it well-suited for an integration into practical and industrial applications. The endeavor already came to fruition: RTLola was successfully deployed for a test flight on an autonomous aircraft of the German Aerospace Center. Moreover, recent work integrates RTLola monitors into aircraft from a leading German manufacturer of electric multicopters for use as air taxis.

1.1. Monitoring

Internal Monitoring

In early forms of runtime verification, the monitor and the system were tightly coupled, a setup called *internal monitoring*. In aspect-oriented programming [Kic+97; HJ08; HV08], for example, the specification is part of the code, though separated from the business logic. Here, instrumentation directives are embedded in the executable code and the monitor runs in tandem with the control code. This grants the monitor deep insights into the internals of the control logic and reduces communication overhead to a minimum.

External Monitoring

In an alternative setup, the monitor is a separate component: *external monitoring*. This component is either handwritten in a general purpose programming language, or generated from a formal specification language. For the latter, the dominating portion uses a temporal logic [HR02; Eis+03]. Here, logics with a *discrete time model* like Linear Temporal Logic (LTL) [Pnu77] or the Property Specification Language PSL [05] allow for automatically constructing a finite state monitoring automaton [Dru00; HR02; FS04; Dah+05; RH05; BLS07; LF07; BZ08; FK09; BLS11; Mas+20]. While the complexity of constructing the automata is high, the results are inherently space- and time-bounded. Other common specification formalism are rules [Bar+04; BRH10] or (timed) regular expressions [ACM02; SR03; Ulu17].

Discrete Time

Real Time

Over time, extensions of these temporal logics arose capturing *continuous real time* rather than a discrete model of time. Prominent examples are Signal Temporal Logic (STL) [DFM13b; WS20] and Metric Temporal Logic (MTL) [TR05] with a variety of dialects incorporating time series [Dru03] or components of a first-order logic [Bas+15; Sch+19]. While there are efficient monitoring algorithms for these languages, they are no longer based on finite state automata.

Quantitative Monitoring

Another area of monitoring is concerned with *quantitative properties*. The meta event definition language [Lee+99], for example, is similar to PSL but has arithmetic capabilities. Similarly, there are quantitative extensions for STL [DM10; DFM13a; Des+17; ZJP21] or MTL [FP06; FP09; BKT17; Alq+18; Jak+18; CM20]. These approaches estimate how much a given input violates or satisfies the specification. This constitutes a measure of robustness of the satisfaction or violation.

Stream Runtime Verification

A common property of all aforementioned approaches is that they ultimately generate a single albeit quantitative verdict for a given trace. In contrast, *stream runtime verification* generates verdicts repeatedly. The Lola [DAn+05] stream-based specification language for synchronous monitors is a pioneer in this category. As such, it is a monitoring-specific variant of the widely and industrially used synchronous programming languages [Ber16] Lustre [Hal+91; Hal05; Bou+17] or Esterel [BG92]. A result of this synchronous paradigm is that the monitor generates a verdict for each element of the input trace, i.e. for each *event*. Here, an event is a tuple with fixed arity of atomic input values. These values correspond to input streams in the Lola specification. The specification also contains output stream declarations with arithmetic expressions over other streams. The monitor evaluates the expression to obtain the verdict.

About a decade after Lola's inception in 2005, several extensions emerged in quick succession. First, Lola 2.0 [Fay+16] adds dynamic stream creation to the language, enabling the monitoring of networks. Recently, HLola [CGS20; GS21] connected Lola to the Haskell ecosystem. Soon after Lola 2.0, three conceptually similar languages followed: RTLola [Fay+17; Fay19; Sch19a], Striver [GS18], and TeSSLa [Con+18a; Leu+18]. The former is the direct predecessor of the language presented in this thesis. All three languages stay true to the stream concept, but extend it to real time and an asynchronous execution model.

Asynchrony

Asynchrony has different meanings in monitoring. In this thesis, it covers two facets: First, neither does the monitor wait for data from the system nor vice versa. This is a common consequence of external monitoring. Second, an event does not necessarily cover every possible input at once. Classic example for the latter point are distributed systems. These systems comprise several independent components. Even if all of these components send regular updates towards a central unit, these updates might arrive at different points in time. A potential source for this asynchrony is the lack of a common clock, or transmission delays induced by bus arbitration. Here, the arbiter might defer transmission of a message in favor of more pressing communication.

1.2. Cyber-Physical Systems

In a CPS, the continuous real world and discrete digital world are intertwined. A CPS assesses information about its surroundings by measuring physical quantities like the air pressure or its own acceleration via sensors. These sensors broadcast their data over a bus where it eventually reaches digital control units. They process the data and make decisions based on it. Such a decision is translated into instructions, and subsequently sent to actuators. These actuators execute the command thereby actively influence the real physical world. This structure of CPS and their relation to both the physical and digital world renders them a special kind of system with particular challenges for monitoring.

Distributed System

First, CPS are inherently *distributed systems*, consisting of several components with varying responsibilities. For example, control units of peripherals, i.e., sensors and actuators, disseminate measured data over a bus, or execute commands received over a bus, respectively. These are low-level tasks, so the component deals with raw data directly sampled for the peripherals. More abstract components combine, cross-validate, and refine this raw data from multiple sources. The output of a GNSS/IMU component, for example, is the amalgamation of data obtained from a Global Navigation Satellite System such as GPS or Galileo, and an inertial measurement unit. This output then constitute the input for high-level components such as navigation units and is used to calculate trajectories necessary for completing the mission.

This already points to several requirements on the monitor. It has to deal with the *asynchronous* nature of a distributed system and a specification has to capture both *low-level* and *mission-level* properties. The latter point also entails that the language needs to provide means to deal with *physical quantities* and *real time*.

Resource Limitations

Second, components of a CPS underlie strict *resource limitations*. While the extent of these limitations depends on the concrete systems, cost is generally a factor. For this reason, components run on slow hardware with little memory. Other factors are the space and power consumption for mobile systems like wearables, cars, or medical implants, as well as weight for avionic application. As a result, the resource consumption of the monitor needs to be *manageable*. This is a deliberately vague term, which becomes more precise when taking the next point into account.

Safety-Critical

Third and last, CPS carry out task with enormous responsibility, ranging from stimulating cardiac rhythms to governing nuclear power plants or steering autonomous vehicles. This renders them *safety-critical systems*. Before such a system is allowed to operate, it is audited by an independent certification authority. This audit ensures that the development abided by the appropriate safety standard. While there are several umbrella standards covering larger industrial areas [Iec10; Nas20], there are also a variety of domain specific ones. Famous examples are the DO-178 for aerospace [Rtc11], DIN EN 50128 for the rail industry [Din06], ISO 26262 for road vehicles [Iso18], and IEC 62304 and 60880 for medical devices [Iec12] and nuclear power plants [Iec06], respectively. This audit requires thorough documentation of all implemented safety features, their development process and impact on the overall safety of the system. A key component in the development of critical components is that major tasks such as the implementation and verification are taken out by technically independent teams.[1] This is a form of redundancy that improves overall safety and should thus also apply to monitoring. Here, this results in the requirement that the monitor should be decoupled from the logic of other components, in particular the ones it monitors.

[1] See, for example, NASA's Software Assurance and Software Safety Standard §4.4.2.2a [Nas20].

Static Analysis

Another standard procedure in the development of components for CPS is the application of type checkers, linters, and *static analysis tools* [Wic+95]. An excellent example for the significance of tool support is the programming language C. In embedded systems, C is the dominant language because of its performance, the fine-grained control it provides, and an environment of approved tools such as certified compilers. This is despite evident problems: the language design is highly permissive and provides little support to the programmers regarding safety or confidence. This permissiveness allows for code without defined behavior; a committee draft of an ISO/IEC standard [II10] for C in 2010 listed nearly 200 such scenarios [NA18]. Hence, there are endeavors to provide alternatives to C for the development of new safety-critical systems such as Ada, its language subset Spark or recently Ferrocene, a language subset of Rust[2]. Yet, the prevalence of C stems from the excellent tool support, which is itself a consequence of C's prevalence. As a result, it proves difficult to break the spiral and transition to a different language.

Runtime monitoring can learn from this example by avoiding the problems of C and ingraining the remedies, i.e., the static analysis tools, right into the language. Hence, the requirement on a specification language for CPS is that it has a formal semantics and is analyzable statically. The possibility to analyze a specification also affects the last point regarding safety: performance reliability. Recall the vague requirement of manageable resource consumption. While the exact dimensions depend on the specific domain and system, it is imperative that specifiers have information regarding the runtime performance statically. Concretely, whether a memory consumption of 2 GB is acceptable depends on the system, but the mere fact that there is a statically determined upper bound on the memory consumption is mandatory. The same idea applies to other resources.

1.3. Conventional Monitoring Meets CPS

After identifying essential criteria for runtime monitors for CPS, the question is to what extent existing approaches satisfy them.

The first set of criteria concern the expressiveness and underlying model of time. Since CPS interact with the real, physical world, a monitor needs to express real-time properties over physical quantities. This excludes the range of discrete time and non-quantitative specification languages, even though they excel for short, abstract specifications like "every request needs to be granted eventually". Moreover, synchronous languages are inapt due to the distributed nature of CPS.

The resource constraints demand a static bound on the memory consumption. As a result, monitoring approaches with linear space complexity in the length of the input trace are not applicable. This either excludes real-time logics like STL or MTL, or it severely limits them as can be seen when considering the property $x \implies F_{[0,3]}y$. It states that

[2]`https://ferrous-systems.com/ferrocene/`; last accessed: 01.02.2022

an event y needs to follow within at most 3 second after an event x. This forces the monitor to memorize the timestamps of each x to ensure that a y followed in time. Yet, the number of xs in 3 seconds is unbounded, thus so is the memory consumption.

Similarly, monitoring approaches with unpredictable running time performance need to be rejected. Here, the constant space complexity already indicates that the running time per event will be constant as well. However, a monitor is not necessarily a purely reactive component. Languages like Striver, TeSSLa, or Faymonville's RTLola enable both exogenous computations, i.e., a reaction to inputs received from the system under observation, or endogenous computations, i.e., computations unprompted by the system. For Striver and RTLola, these endogenous computations are controlled in the sense that there is an a priori bound on the number of endogenous computations per real-time second. This is not the case for TeSSLa, which allows specifiers to define Zeno[3] behavior. In contrast, endogenous computations in RTLola are isochronous, i.e., they occur at statically predefined points in time, and Striver explicitly excludes Zeno behavior.

What is left are further safety concerns. First, internal monitors violate the technical independence between control components and the monitor. Figure 1.1 illustrates an example architecture for internal and external monitors. Since internal monitors operate on the same physical object as the control logic, they are inherently coupled and thus influence each other. Moreover, internal monitoring requires annotations in the code of the controller. These are either explicit inline assertions or contracts regarding the behavior of logical components such as functions. The latter case constitutes a looser coupling since the business logic is separate from the monitor annotations. However, changes to the controller also require changes to the monitor, hindering technical independence. Hence, while internal monitoring is undoubtably valuable for monitoring singular components, their deployment is complementary to the comprehensive monitor for the CPS.

A key element of safety and certifiability is that a specification must be clear and comprehensible. This is a notoriously subjective topic when it comes to programming languages. However, while logics work well for short properties or a conjunction of multiple short properties, they become increasingly opaque the more complex properties grow. This is particularly true when nesting is involved. Programming languages and specification language with a clear syntax like TeSSLa are generally preferable for large-scale properties.

A counterpoint is the option to analyze a language automatically, where the roles are reversed. Logics usually have a low number of base operations, which is great for arguing about them. Programming languages on the other hand have a significantly greater number of base operations. Moreover, languages with imperative aspects allow for functions with side effects. This renders automatic analysis exceedingly hard. While stream-based specification languages are syntactically similar to programming languages, they are generally more restrictive. For example, streams are not necessarily independent

[3] A system is *Zeno* if it attempts to complete an infinite amount of actions in a finite amount of time.

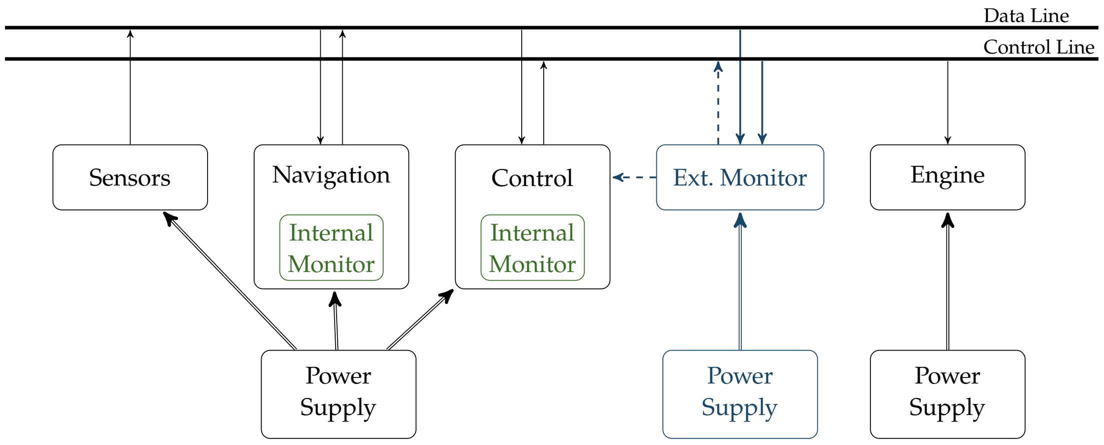


Figure 1.1.: Example system architecture with an external and two internal monitor components. While the internal monitors reside in the navigation and in the control unit, the external one constitutes a separate component. It passively listens to information communicated over the bus. When it identifies a contingency, it either raises an alarm over the bus or over a direct line to the controller.

of each other, but they are inherently modular and cannot implicitly affect each other due to a lack of side effects. As a result, while analyzing such specification languages is less convenient than logics, it is still manageable. Though, for the aforementioned languages, such analyses are not stable, yet.

1.4. Monitoring CPS with RTLola

The last section has shown that the current state of the art features several monitoring approaches with different advantages and drawbacks regarding their applicability to CPS. Yet, neither of them perfectly fits this niche. For this reason, this thesis presents the specification language RTLola with an accompanying static analysis and realization options. All three components are specifically designed to fit into the niche of monitoring CPS.

RTLola is an asynchronous real-time stream-based specification language syntactically similar to programming languages. It is both expressive and comprehensible, focusing on clarity over brevity. For this, it provides primitives for common arithmetic operations and grants specifiers precise control over the temporal behavior of the monitor. This control allows timely exogenous computations, regularly scheduled endogenous computations, and a mix of both. The latter allows for real-time related reactions like "ensure path planning produces a trajectory after at most two minutes". Here, the monitor needs to start a two-minute timer (endogenous) right after the path planner started to work (exogenous).

Ingrained into the language is a static analysis. This analysis consists of an intricate type system, an analysis of the memory consumption, and a check for inconsistencies in the specification. Part of a type check is to determine which stream accesses can statically be guaranteed to succeed and which potentially do not. It then demands specifiers to supply default values for fallible accesses, compelling them to cover corner cases. Moreover, it issues a warning when a default value is superfluous since this points to a mismatch between specification and intention. The analysis of the memory consumption enables specifiers to verify statically that the available memory suffices to host the monitor. It also enables the monitor to forgo expensive and fallible dynamic allocation — a common requirement for embedded devices.

Further, there are realization options for a specification. The first one generates a hardware description of a monitor. This description is then synthesized onto a hardware board with a commercial off-the-shelve synthesis tool. During this process, the tool generates an additional report regarding the resource consumption of the monitor. In particular, it states the maximum power consumption and determines statically whether the available hardware board is sufficient to host the monitor. The second realization is a verifying compiler. It injects verification annotations into the generated Rust code. These annotations enable an off-the-shelve static verification tool to prove the monitor correct with respect to the theoretical underlying semantics of the specification language.

This renders RTLola perfectly suited for this niche of monitoring thanks to its expressiveness, analyzability, and resource awareness.

1.5. Contributions

For this reason, this thesis presents a runtime verification approach that is expressive and performant enough for the application in real-world CPS and boosts reliability and certifiability.

Chapter 2: RTLola To this end, the thesis introduces the RTLola specification language and discusses its design principles as well as their effect. These principles guide design decisions to obtain a language with the following key characteristics:

- The language is comprehensible. Tough the specification is designed by dedicated experts, its quintessence is easy and quick to grasp for other developers and certification authorities.

- The language design guarantees that specification can be monitored with a bounded amount of memory plus the computation time for processing single events is bounded.

- The language supports specifiers by offering declarative language primitives, prevention of side effects through modularity, and reducing superfluous information such as boilerplate code.

The language also allows for dynamic stream creation and filtering, plus it lifts the restriction on isochronous computations.

Ingrained in the language is an intricate static type system. This type system increases confidence in the specification since it ensures an internal consistency by catching specification errors. Beside classic type errors[4] prominent in programming languages, it also captures timing errors. In particular, it classifies value access operations as fallible and infallible depending on whether the monitor can verify that the value is present at the time of access. If not, the type system requires specifiers to supplement default values. This is a similar concept to optional types in programming languages. Here, a fallible operation returns a value of type `Option<T>` rather than `T`. This forces programmers to either supply recovery code or risk a runtime error. However, in RTLola, specifiers are not allowed risk runtime errors because a crashing monitor is not an option in safety-critical CPS. Yet, to prevent excessive error handling, the type system statically determines whether operations succeed invariably. If so, no error handling is required. This imposes a certain complexity onto the type system, rendering type annotations extensive. As a counter measure, this thesis also introduces type inference, which allows users to omit almost all type annotations.

In addition to the type checker, this thesis presents two more static analyses. The first one is a well-formedness analysis, which determines whether a specification has a unique semantics. This is necessary due to the declarativeness of RTLola specifications. While they provide convenience for specifiers, they can also lead to the absence or abundance of models for a specification. The second static analysis determines the memory requirement of the monitor. This has two major consequences. First, the monitor does not require expensive dynamic memory allocation, which is a potential point of error when attempting to allocate memory after depleting the available one. Second, specifiers can determine whether the available hardware can host the monitor pre-deployment.

The discussion of the language concludes with an empirical evaluation of the open source implementation of RTLola and an overview over application areas in which RTLola was successfully deployed.

Hence, the contributions can be summarized as:

→Sec. 2.1.2, p. 27

- Section 2.1.2: A list of language design maxims for specification languages for safety-critical cyber-physical systems.

→Sec. 2.2, p. 31

- Section 2.2 The syntax of the RTLola specification language with dynamic stream creation and filtering both in regular and desugared form.

[4]Common examples are attempts to multiply strings or dereference boolean values.

- Section 2.3: A type system for RTLola which allows for identifying fallible and infallible stream accesses. →Sec. 2.3, p. 42
- Section 2.4.1: The formal semantics of RTLola. →Sec. 2.4.1, p. 62
- Section 2.5.1: The definition of a dependency graph for RTLola specification. →Sec. 2.5.1, p. 70
- Section 2.5.2: The formulation of a syntactic criterion for well-formedness of RTLola specification and a proof that this criterion implies the semantic criterion of well-definedness, i.e., the existence and uniqueness of an evaluation model. →Sec. 2.5.2, p. 71
- Section 2.5.3: The definition of an evaluation order for RTLola which generates the aforementioned model plus a constructive proof of its existence. →Sec. 2.5.3, p. 78
- Section 2.5.4: An algorithm determining the memory bounds of an RTLola specification. →Sec. 2.5.4, p. 79
- Section 2.7: An empirical evaluation of the implementation covering the parsing of a specification, as well as all of its analysis steps, and the performance of an interpreter. →Sec. 2.7, p. 85
- Section 2.8: A showcase of the application areas in which RTLola was successfully deployed. →Sec. 2.8, p. 91

Chapter 3: Realizations While this provides an excellent theoretical foundation for monitors for CPS, a concrete realization is fundamental for the integration into a system. To this end, this thesis presents two compilers, one targeting hardware and one targeting software.

The first one consists of a mathematical formulation of a monitor for an RTLola specification. This formulation then translates to a hardware description language, based on which the prototype implementation generates code in the hardware description language VHDL. This description is then synthesized onto a field-programmable gate array (FPGA).

Hardware solutions have dramatic advantages over software-based solutions, as they are generally faster and consume less power. The exact values vary depending on the specification, yet, an empirical evaluation revealed that an RTLola hardware monitor for an aircraft requires as little as $0.121\,W$ when idle and $1.620\,W$ under peak pressure. In comparison, under the same conditions a Raspberry Pi Model 2 requires $1.1\,W$ and $2.1\,W$.[5] A particular source of improvement is the synergy between the inherently parallel nature of hardware, and the modular design of RTLola specifications. This allows for employing a pipelined and concurrent structure, i.e., the monitor processes several events and several sub-tasks of the same event at once. This results in a 91% increase in throughput with a negligible increase in power consumption of approximately $10\,\mathrm{m}W$.

[5] `https://www.pidramble.com/wiki/benchmarks/power-consumption`; last accessed: 01.02.2022

The second compiler generates Rust code for a monitor.[6] The generated code is highly efficient, exceeding the performance of an interpreter executed on the same hardware by orders of magnitude. While this is interesting per se, the major novelty is that the compiler is a verifying compiler. As such, it addresses the question of whether one can trust the monitor. The complexity of this issue scales with the complexity of the specification and expressiveness of the underlying specification language. An LTL formula, for example, is translated into a finite state machine. While the translation is non-trivial, the correspondence of the automaton with the specification is comprehensible. For more complex languages, this connection grows opaque.

There are several approaches tackling this problem. One option is to verify the compiler itself, as has been done for the synchronous programming language Lustre [Hal+91; Hal05; Bou+17]. As a result, after successful verification, every output of the compiler is immediately trustworthy. However, the verification of the compiler is exceedingly difficult compared to the verification of the result. Plus, every change to the language or output generation warrants a repetition of the effort.

Verifying compilers are a contrary approach. They exploit the fact that the compiler has intimate knowledge regarding the semantics of the specification language. This enables it to inject verification annotations or proof artifacts into the output [NL98b; App01; Nec02; Hoa03; Her+05]. These annotations then guide an automatic verification process. As a result, verification scales significantly better and thus succeeds for programs normally resulting in a timeout or inconclusive results. In the context of runtime monitoring, this approach was successful in proving absence of arithmetic error and undefined behavior [Pik+10] via bounded model checking. The compilation presented in this thesis proves functional correctness in the sense that the semantics of the Rust monitor coincides with the theoretical model and termination of every evaluation cycle. For the actual proof, the compiler uses the Rust-specific Prusti [Ast+19] frontend of the Viper [MSS16] verification framework.

This yields the following contributions:

→Sec. 3.1, p. 100

- Section 3.1 A mathematical description of a pipelined hardware monitor for RTLola specification with dynamic stream creation.

→Sec. 3.1.6, p. 122

- Section 3.1.6 An empirical evaluation covering the hardware resources required for the realization, the throughput, and power consumption.

→Sec. 3.2, p. 128

- Section 3.2 A compiler translating a Lola specification into sequential or concurrent Rust code with verification annotations.

→Sec. 3.2.6, p. 142

- Section 3.2.6 An empirical evaluation of the feasibility of the verification and the performance of the generated monitor.

[6]Note that this translation considers Lola instead of RTLola.

Chapter 4: Conservative Hybrid Automata The last part of this thesis showcases how the design of a runtime monitoring specification assist in subsequent parts of the development process. The core of this work is an algorithm generating a model for a system out of development artifacts. These artifacts are the RTLola specification plus execution traces of the system, which are the result of test runs. The generated model is *conservative*, i.e., it is a provable over-approximation of the system under several realistic assumptions on the inputs. This model can then be used in a variety of following steps, like further analysis of the system, or for predictive measures at runtime.

Hence, the contributions are:

- Section 4.3: The automatic generation of a hybrid model for a system based on an RTLola specification and traces of test runs. ➔Sec. 4.3, p. 162
- Section 4.4: A list of requirements on the input data plus a proof that these requirements ensure that the generated automaton is an over-approximation. ➔Sec. 4.4, p. 171
- Section 4.5: A case study showing the scalability of the approach and the quality of the constructed automaton even for low-quality traces. ➔Sec. 4.5, p. 178

1.6. Publications

This thesis is based on several peer-reviewed publications that arose from joint work with my colleagues and advisor. The following list contains all essential publications.

StreamLAB: Stream-based Monitoring of Cyber-physical Systems
Peter Faymonville, Bernd Finkbeiner, Malte Schledjewski, Maximilian Schwenger, Marvin Stenger, Leander Tentrup, and Hazem Torfah
In Proceedings of the 31st International Conference on Computer Aided Verification 2019 [Fay+19a].

FPGA Stream-Monitoring of Real-time Properties.
Jan Baumeister, Bernd Finkbeiner, Maximilian Schwenger, and Hazem Torfah
Published in Transactions on Embedded Computer Systems 2019, presented at the International Conference on Embedded Software 2019 [Bau+19a].

Verified Rust Monitors for Lola Specifications
Bernd Finkbeiner, Stefan Oswald, Noemi Passing, and Maximilian Schwenger
In Proceedings of the 20th International Conference on Runtime Verification 2020 [Fin+20].

Monitoring Cyber-Physical Systems: From Design to Integration
Maximilian Schwenger
In Proceedings of the 20th International Conference on Runtime Verification [Sch20].

Real-time Stream Monitoring with StreamLAB
Peter Faymonville, Bernd Finkbeiner, Malte Schledjewski, Maximilian Schwenger, Leander Tentrup, Hazem Torfah
Presented at the 4th Workshop on Monitoring and Testing of Cyber-Physical Systems at CPSWeek 2019 [Fay+19b].

On the Similarities of Aircraft and Humans: Monitoring CPS with StreamLAB
Jan Baumeister, Bernd Finkbeiner, Maximilian Schwenger, and Hazem Torfah
Presented at the CyberCardia Workshop at ESWeek 2019 [Bau+19b].

Conservative Hybrid Automata from Development Artifacts
Niklas Metzger, Sanny Schmitt, and Maximilian Schwenger
Published on ArXiv [MSS21]; not peer-reviewed.

In addition to these, there are several publications that emerged in the context of this thesis. While not being an essential part of it, they helped shape the thesis to what it ultimately became.

RTLola Cleared for Take-Off: Monitoring Autonomous Aircraft
Jan Baumeister, Bernd Finkbeiner, Sebastian Schirmer, Maximilian Schwenger, and Christoph Torens
32nd International Conference on Computer Aided Verification 2020 [Bau+20a].

RTLola on Board: Testing Real Driving Emissions on your Phone
Sebastian Biewer, Bernd Finkbeiner, Holger Hermanns, Maximilian A. Köhl, Yannik Schnitzer, and Maximilian Schwenger
In Proceedings of the 27th International Conference on Tools and Algorithms for the Construction and Analysis of Systems [Bie+21].

Robust Monitoring for Medical Cyber-Physical Systems
Bernd Finkbeiner, Andreas Keller, Jessica Schmidt, and Maximilian Schwenger
Proceedings of the Workshop on Medical Cyber Physical Systems and Internet of Medical Things 2021 [Fin+21].

Simplex Architecture Meets RTLola
Bernd Finkbeiner, Jessica Schmidt, and Maximilian Schwenger
Presented at the 5th Workshop on Monitoring and Testing of Cyber-Physical Systems at CPSWeek 2020 [FSS20].

Automatic Optimizations for Runtime Verification Specifications
Jan Baumeister, Bernd Finkbeiner, Matthis Kruse, Stefan Oswald, Noemi Passing, and Maximilian Schwenger
Presented at the 6th Workshop on Monitoring and Testing of Cyber-Physical Systems at CPSWeek 2021 [Bau+21].

Chapter 2

The RTLola Specification Language

A monitor is generated from a formal specification. This puts the specification language in the center of the process. It details information on what constitutes either a safety violation or the desired behavior of the system. These properties need to be expressed in a specification language that can be translated into an executable artifact such as programmable hardware (cf. Section 3.1) or executable code (cf. Section 3.2). The specification is grounded in a stream-based model, i.e., rather than receiving a single input value and computing a single output value, the monitor operates on a *stream*. This means, the monitor receives a sequence of values peu à peu and provides an appropriate verdict at various points in time. Though there is a wide selection of potential specification languages, they can be roughly sorted onto a scale ranging from highly formal languages to "informal" programming languages.

→Sec. 3.1, p. 100
→Sec. 3.2, p. 128

Stream

Prominent examples on the formal end of the spectrum are *temporal logics* such as Linear Temporal Logic [Pnu77] (LTL), Signal Temporal Logic [MN04] or Metric First-Order Temporal Logic [Cho95]. For some of them, their original definitions assume accesses to the entire input at once. However, there are alternative algorithms not relying on this assumption, which generate multiple verdicts during the execution of the monitor rather than only one at the end [MN04; BLS11; DFM13b; Bas+15; Mas+20]. These languages allow for automatic translation into executable monitors with precise knowledge regarding their performance. An LTL specification, for example, can be transformed into a finite state automaton [BLS11]. The automaton can then easily be translated into executable code where both the exact memory consumption and the number of instructions required for the reception of a new input is statically determined. While these are optimal conditions for a safety-critical component, the expressive power of LTL and similar logics is severely limited. Generally, formal languages excel when dealing with short, abstract specification like "Every request must be answered" or

Temporal Logics

"The aircraft must land within three time units". However, they are inadequate when specifying complex quantitative properties.

Programming Languages

On the other extreme end of the spectrum lie *general-purpose programming languages*. They excel in terms of expressiveness, enabling specifiers to formalize any kind of property. However, analyzing their dynamic behavior statically poses an enormous challenge due to the state space explosion problem. Moreover, expressing properties directly in code is more explicit and less abstract than in logics. For example, the LTL formula $\mathsf{F}p$ requires that the event p occurs at some point of the execution. The specification leaves details open regarding when exactly the event occurs, as well as how to check the validity of the property. A representation in code is comparatively verbose and specific, for example:

```
typedef enum verdict { Sat, Undecided } Verdict;
enum ap { p, ... };
Verdict check_Fp(bool *event) {
    static verdict last_verdict = Undecided;
    if event[p] {
        last_verdict = Sat;
    }
    return last_verdict;
}
```

This is unproblematic if the code is final, i.e., it will be deployed on the system as is. However, specifications are usually designed and validated early in the development process when they are still subject to change. In this case, regularly adapting and revising large chunks of the monitor code is costly.

Specification Language

Dedicated *specification languages* such as RTLola attempt to eradicate problems of either of these options, hence occupying a sweet spot on the scale. For this, they provide specifiers an abstract view on the specification, establish a safe framework suitable for the development of safety-critical systems, and provide sufficient expressiveness to be applicable for cyber-physical systems. They are particularly useful for large, low-level, system-wide specification.

The remainder of this chapter introduces the RTLola language. After providing an intuition into the language, it defines its syntax, the rationale behind its design, as well as its type system and an analysis of a specification. Next, it presents an empirical evaluation of the RTLola toolkit and outlines projects, in which it was practically used. Last is an outline of related work. This in particular compares the version of RTLola presented in this thesis against earlier versions, most notably the one presented by Faymonville in 2019 [Fay19].

2.1. Language Design

One of the design principles of RTLola is to be easy to understand. Embracing this decision, it is best to understand the language by considering examples.

2.1.1. RTLola by Example

In essence, an RTLola specification declares inputs and states how to transform them into outputs.

```
input lon: Float64
input lat: Float64
output distance_to_zero := sqrt(lon**2 + lat**2)
```

Due to syntactic similarity with programming languages, one can immediately grasp the gist of the specification. A monitor for it waits on two input streams, which carry the current longitude and latitude of the system. Upon reception of these values, it computes the distance to zero via the Pythagorean theorem. Note that the monitor assumes the inputs to operate on a common global clock.

RTLola has a strong static type system with type inference. This can be seen in the type annotations of both input streams, which renders them 64-bit wide floating point numbers. The output stream lacks a type annotation. This is valid because its type can be inferred: The square root function has the polymorphic type $\mathit{sqrt}\langle \mathsf{T} : \mathit{Float}\rangle : \mathsf{T} \to \mathsf{T}$ meaning that the input needs to be arbitrary type representing floating point numbers. If this is the case, then the output carries the same type as the input, rendering the output stream in the specification of type `Float64` as well.

Value Type

This type system only argues about the shape of *values*. This is sufficient for programming languages, yet, there is another, more critical dimension for RTLola due to its stream-based nature: time. The first facet of time is the questions of *synchrony*. For this, recall the specification above in which timing is intuitive: the monitor re-computes `distance_to_zero` whenever it gets new values for `lon` and `lat`. However, since RTLola operates in an asynchronous setting, inputs are independent, i.e., the monitor can receive updates for each input at the same time or an arbitrary, non-empty subset thereof. Hence, while the *"and"* in the timing of the monitor is natural, it is not imperative. The timing could as well be a *disjunction*, so the monitor re-computes the output whenever `lon` *or* `lat` is updated. Syntactically, the specification would look as follows:

(A-) Synchrony

Conjunctive Timing

Disjunctive Timing

```
input lon: Float64
input lat: Float64
output distance_to_zero @ lon∨lat :=
    sqrt(lon.hold(or: 0) ** 2 + lat.hold(or: 0) ** 2)
```

In this specification, the specifier explicitly states that they want the disjunctive timing as per the `@ lon ∨ lat` annotation. Now suppose the monitor receives an update for lon without one for lat. A re-computation of the output requires the monitor to have access to a value for lat. In the specification above, the specifiers declares that it should resort to the latest value of the stream via a 0-order hold. Since this value is potentially non-extant, RTLola requires the .hold syntax rather than defaulting to this behavior. Moreover, the specification needs to contain a fall-back (or: 0) in case the lat stream does not contain a value, yet. In summary, RTLola defaults to the more intuitive conjunctive timing, and allows specifiers to opt in the disjunctive timing provided they supply enough information for the execution.

Overriding default timing behavior gives specifiers a new dimension of control over the monitor. Consider the following specification.

```
input time: Float64
input waypoint_reached: NoValue

output time_at_wp @waypoint_reached := time.hold(or: 0)
output time_for_wp @waypoint_reached := δ(time_at_wp)
```

Purely Temporal Streams

There are two points of interest in the specification. First, the waypoint_reached stream has value type NoValue. This brands the stream as *purely temporal*; its values are meaningless, their point of arrival only indicates when the system reached a waypoint. And this information serves as clock for the output streams. Even though their values solely depend on the time stream, the monitor evaluates them whenever the system reaches a waypoint. Hence, time_at_wp carries the timestamp of the last waypoint event and time_for_wp shows how much time passed between reaching two consecutive waypoints.

For the computation for these streams, note that time_at_wp accesses the time stream with a hold operation and provides a default value. This is necessary precisely because the timing of the output is decoupled from the time input stream. On the other hand, time_for_wp is computed at the same time as the stream it accesses, so there is no need for a hold or default values. However, note that the expression invokes the δ function. This is a shorthand notation for accessing the ultimate and penultimate value of the target stream and computing the difference. If at least one of these values does not exist, the difference is 0.

Periodic Streams

So far, the evaluation of all streams temporally depended on updates to input streams, so-called *events*. However, there is a plethora of use cases where a monitor is supposed to check a condition independent of events. Most trivially, consider a specification in which a monitor checks whether a sensor ceases to generate readings. This would be futile if the monitor only checks the condition upon reception of a sensor value. Hence, consider the following specification monitoring a sensor which is supposed to produce readings with 10 Hz:

```
input sensor: Int32
output readings_per_min @1Hz :=
   sensor.aggregate(over_precisely: 1min, using: count).defaults(to:
      600)
output sensor_faulty := readings_per_min < 590 ∨ readings_per_min >
   610
```

Then, the first peculiarity is the timing annotation of `1Hz`. Its effects meet the intuition: the monitor evaluates `readings_per_minute` *periodically* with a frequency of 1 Hz. The second peculiarity is the expression of the first output. Intuitively, rather than accessing a set amount of values of the input stream, it aggregates — in this case: counts — all values that were produced in the last minute. Such an aggregation is a *sliding window aggregation*. Appendix A.1.3 provides background information on how to handle them efficiently and which aggregation functions are permitted.

Periodic Stream

Sliding Window Aggregation

→ Appx. A.1.3, p. 211

There are two questions that come to mind: First, if there is an `over_precisely` argument, is there an imprecise counterpart? Second, why does the expression require a default value even though one would expect 0 as default for counting. Both questions are related. The instruction `over_precisely` instructs the monitor to start the evaluation only *after* the duration of the window, i.e., 1 min, has passed. Hence, it will always aggregate over *precisely* 1 min.

When dropping the `_precisely` suffix, the aggregation starts immediately. As a result, for a healthy sensor, the expected first aggregation result after a second would be 10. However, this is an improper violation of the condition of the second output, which constitutes a false positive. With the `_precisely` qualifier in place, the specifier has to provide the default output up until one minute has passed. In this particular case, the value of 600 is the expected number of readings per minute for a healthy sensor. This means that the monitor assumes the sensor to work properly until presented with evidence against this assumption.

There are two more things to note about timing annotations. First, the second output does not require a timing annotation. Since it accesses the first output directly, RTLola will infer both timings to be identical. Second, the periodic timing of the first stream is mandatory, any event-based timing results in a type error. The reason behind this is the way the sliding window aggregations work. Consider the following specification:

```
input i: T
output o @i := i.aggregate(over: δ, using: γ)
```

Here, whenever i generates a new value, then so does o by aggregating all values of i in the last δ time units. Suppose, the monitor receives an update for i at time t. This value is potentially relevant for the aggregation until precisely time $t + \delta$, meaning both the value and its timestamp needs to be stored until then. This renders lossy[1] pre-aggregation unsound, as the following example illustrates: Suppose the monitor receives two updates

[1]Lossless aggregation allows for restoration of both inputs, but has a non-constant memory requirement.

for i, the first one has value v_1 at time t_1 and the second one is v_2 at time $t_2 = t_1 + \varepsilon$. Pre-aggregation yields a singular value v for a timestamp t. If the stream i later receives an update at time $t_1 + \delta + \varepsilon/2$, the monitor has to aggregate earlier values of i. This has to account for v_2, but not for v_1, which renders v worthless.

Therefore, an aggregation function in an event-based stream forces the monitor to memorize every single event that occurred in the window of time. Since the frequency of input streams is unknown, this raises the memory consumption of the monitor to an unbounded level. This is unacceptable for RTLola since it specifically targets CPS where static guarantees on the runtime behavior is critical.

Isochronous Timing

Demanding streams with aggregations to be periodic renders them *isochronous*, i.e., when the initial point in time is known[2], then so are all further time points of evaluation. This knowledge allows the monitor to apply a technique presented by Li et al. [Li+05]: It separates the timeline into panes of equal size and pre-aggregates all values within them. While this technique is lossy in general, the pre-determined points of evaluation renders it lossless [Sch19a]. Lastly note that this technique only works for a select group of aggregation functions, i.e., *list homomorphisms*. RTLola provides an array of pre-defined aggregation functions such as integration, averaging, summation, and (co-)variance. For an overview over their characteristics and how to compute them efficiently, confer Definition A.8.

List Homomorphisms

→ Def. A.8, p. 211

Advantage of Precise Timing

Primitives for event-based and periodic operations grant the specifier precise control over the timing of a specification. This in particular enables them to design specifications for high-level properties. As an example, consider a drone that is supposed to travel to a waypoint. Upon arrival, it will receive a new destination. Besides low-level properties like sensor (cross-) validation, the monitor should determine whether there is a correlation between deviations from the shortest path between consecutive waypoints and their distance. This property translates to the following specification in RTLola:

```
input wp, pos: (Float64, Float64)
output wp_dist := δ(wp)
output traveled := δ(pos) + traveled.last(or: 0)
output traveled_at_wp @wp := traveled.hold(or: 0)
output deviation @wp :=
    abs(wp_dist.last(or: 0) - traveled_at_wp.last(or: 0))
output distance_v_deviation @wp := (wp_dist, devi)
output cov @ 1Hz := distance_v_deviation.covariance(over: inf)
output var_dist @1Hz := wp_dist.variance(over: inf)
output var_devi @1Hz := devi.variance(over: inf)
```

[2]This criterion becomes relevant a little later.

```
output corr := cov / (sqrt(var_devi) * sqrt(var_dist))
```

Listing 2.1: Specification monitoring for stale connections.

As input, the monitor receives the currently active waypoint and the position of the drone. The first output stream computes the distance between two consecutive waypoints. The timing of this computation is tied to the input stream wp since it only depends on it. Similarly, the next output only depends on pos. Upon reception of a position update, the monitor computes the distance between two such updates and sums up the results. This yields the total distance traveled.

Since there is no temporal tie between the two input streams, neither is there one between wp_dist and traveled. Hence, the third output stream samples traveled upon arrival at a waypoint w_i, i.e., at the time of reception of waypoint w_{i+1}. This is necessary such that the monitor can — at these points in time — compute the deviation of the drone from the shortest path. To this end, it takes the ideal distance between the last two waypoints, i.e., w_{i-1} and w_i, via wp_dist.last, and the actually traveled distance via traveled_at_wp.last. The deviation is then the absolute difference between the values.

So far, all computations were event-based. Before transitioning to periodic computations of high-level statistics, the specification declares a tuple stream containing the distance between waypoints zipped together with the deviation measured when traveling to the waypoint. For the next step, recall that the correlation of two series of values is their correlation over the product of standard deviation, i.e., $\mathrm{cor}_{X,Y} = \mathrm{cov}_{X,Y} \left(\sqrt{\mathrm{var}_X}\sqrt{\mathrm{var}_Y}\right)^{-1}$ Hence, the monitor computes the co-variance of wp_dist and devi as well as their variances with an indefinite aggregation operation. Lastly, the corr stream is the natural translation of the formula for the correlation.

Dynamic Stream Creation and Filtering

While sliding window aggregations always look into the past, some specifications require a glimpse into the other direction. Consider a system that can receive a landing command which it has to obey within 100 seconds. This requirement can be translated into RTLola as follows:

```
input landing_cmd: NoValue
input altitude: Float32
output compliance
    spawn @landing_cmd
    eval @10mHz with altitude.aggregate(over: 100s, using: min) < ε
    close immediately
```

Listing 2.2: Specification checking if the system obeys a landing command within 100 s.

One can immediately see that the output stream declaration contains significantly more logic than before. We will go through it line by line. The first line merely declares the name of the stream. Second, the *spawn clause* states a condition under which the Spawn Clause stream starts to exist. In this case, the stream is spawned as soon as the system receives a landing command. While this is a purely temporal condition, it can also be semantic such as `spawn @landing_cmd when altitude > 5`. The third line contains the evaluation information including the evaluation frequency of 10 mHz, which translates to a period of 100 s. The expression checks whether the lowest measured altitude in the last 100 s was below some ε, indicating a successful landing. The combination of evaluation frequency and sliding window size results in a singular check precisely 100 s after reception of the landing command taking all measurements of the altimeter in the interim into account.

Close Clause

Last, the *close clause* states when to terminate the stream. Here, `immediately` refers to the point in time after the first evaluation of the stream, i.e., after 100 s. In combination, the specification declares a stream that is spawned when receiving a landing command, waits for 100 seconds, checks whether the system obeyed, terminates, and waits on the next command.

This concept of dynamic stream creation is a special case of RTLola's parametrization semantics. Generally, output streams can be parametrized by a sequence of values. These values are determined at the point of stream creation via the spawn clause. Both the evaluation and close clause can refer to the parameters. This mechanic is commonplace in network monitoring, so consider the following specification which checks for stale connections.

```
input src: IPv4
input tcp_flags: UInt16

output fin := tcp_flags ^ 0x01 > 0
output syn := tcp_flags ^ 0x02 > 0
output rst := tcp_flags ^ 0x04 > 0

output terminate := fin ∨ rst

output heartbeat(ip: IPv4): NoValue
    spawn when syn with src
    eval when src = ip
    close when src = ip ∧ terminate

output stale(ip: IPv4)
    spawn when syn with src
    eval @10mHz with heartbeat(ip).count(over: 100s) = 0
    close when terminate ∧ src = ip
```

Listing 2.3: Specification monitoring for stale connections.

The specification expects two inputs, the source IP address of a connection and the TCP flags[3]. First, the specification extracts the three relevant flags and defines a stream indicating termination of a connection. Second is a purely temporal stream indicating an incoming heartbeat from a specific IP address. This IP address is the parameter of the stream, declared in parentheses. The spawn clause states the semantic condition under which the stream is spawned, i.e., when the SYN flag is set. Since the stream has a parameter, the spawn clause needs to provide it via an expression following the `with` keyword. That means, whenever an incoming connection has the SYN flag set, an *instance* of the stream is spawned with the source IP address as parameter unless such an instance already exists. The spawned instance ticks whenever the semantic criterion in the evaluation clause is true, i.e., when the source address coincides with the parameter of the stream. Finally, the termination of the instance has a temporal and a semantic criterion. Temporally, it can only terminate when it ticks, i.e., when there is an incoming connection from the respective IP address. Semantically, the connection needs to be terminating, i.e., either the FIN or the RST flag is set. The last output stream is `stale`. It follows the same spawn pattern. Its termination is restricted by the same semantic criterion as `heartbeat`: an incoming connection needs to be terminating while its source coincides with the parameter address. Only the evaluation clause varies strongly: The stream ticks periodically every 100 s and checks whether there was at least one heartbeat in the interim.

Stream Instance

Putting it Together: Running Example

The following specification summarizes most of RTLola's features in the context of CPS. Here, the monitor is supposed to check two properties: First, upon reception of a `pause` command, the system has to stop within half a second and remain in place for one and a half seconds. Second, the system surveils its immediate environment for alien objects, assigns them an ID, and computes their distance to the system. The monitor has to compute the approximate time to a potential collision provided. This assumes that the velocity of the system does not drastically change, and the foreign object is stationary.

```
input oid: Int8
input distx, disty, spdx, spdy: Float32
input pause_cmd, obj_lost: NoValue

output velo := sqrt(spdx*spdx + spdy*spdy)
output obey_pause
    spawn @pause_cmd
    eval @0.5Hz with velo.integrate(over: 1.5s) < ε
    close immediately
```

[3]This requires only 9 bits (or 12 when counting the reserved bits), so the input is padded arbitrarily to fill 16 bits.

```
output sx @distx := spdx.hold(or: 0.0)
output sy @disty := spdy.hold(or: 0.0)
output collision_possible := angle((distx, disty), (sx, sy)) > ε_φ
output ttc(id: Int8)
    spawn @oid with oid
    eval
        when collision_possible
        with min(
            if sx ≠ 0 then distx / sx else disty / sy,
            if sy ≠ 0 then disty / sy else distx / sx
        )
    close @obj_lost ∧ oid when oid = id
```

Listing 2.4: Specification monitoring for obedience and the time to a collision.

The first output stream is an auxiliary stream that computes the length of the velocity vector. The second output is similar to the `compliance` stream in Listing 2.2. It waits for two seconds and then ensures that the change in position in the last one and a half seconds was close to zero, i.e., that the system obeyed the pause instruction.

The second and third streams sample the velocity of the system whenever it emits an update regarding a foreign object. Since a collision is only possible if the angle between the distance vector and velocity agree, the fourth output determines this angle and compares it against the threshold ε_φ. The `angle` function here is an abbreviation for the actual computation, which is $\arccos(\mathtt{dist} \cdot \mathtt{s} \cdot (|\mathtt{dist}| \cdot |\mathtt{s}|)^{-1})$, i.e., the inverse cosine of the scalar product of both vectors divided by the product of their length.

Last, `ttc` computes the time to collision with some object. Its spawn and close clause state that a new stream instance should be created whenever a new object is in range and persist until the object is lost. The static filter prevents an evaluation if a collision is not possible. Otherwise, the stream computes the minimal time to collision per dimension. Note that these values will either be almost equal or one of them is undefined if either speed value is 0. Hence, the expression computes the minimum of these values but substitutes the undefined division by the value of the other dimension.

Triggers

So far, an RTLola specification was a collection of input and output streams. However, there is a third kind of stream: *triggers*. While output streams contain quantitative information about the current state of the system and monitor, a trigger is a qualitative measure, i.e., a boolean constraint. These constraints encode safety violations. Whenever they turn true, the monitor issues an alarm such that the system can react appropriately. This reaction could be a change of course, a hot swap to a redundant replacement module, or the initiation of an emergency stop.

Trigger

Syntactically, triggers are mere RTLola expressions coupled with a format string:

```
input altitude
trigger alititude > 5000 "Warning: altitude of {} exceeds clearance."
    altitude
```

Triggers constitute the main interaction point between the monitor and the system during the execution. However, on the theoretical side, they can be treated as output streams except when emitting verdicts. Hence, in the following, they play a subordinate role.

Conclusion

This concludes the introduction into RTLola. The difference in complexity of single stream declarations when comparing early versus late examples shows that RTLola takes a lot of work out of the specifier's hands by providing default values. This is especially noticable for streams with simple timing behavior. However, RTLola allows specifiers to override these defaults to express complex temporal dependencies between streams. Thanks to RTLola's declarative nature, the specifiers only need to state dependencies; management of stream instances and their evaluation happens behind the scenes.

2.1.2. Design Maxims

The RTLola specification language is designed for use in safety-critical cyber-physical systems. Here, safety is of utmost importance. Satisfying this requirement goes beyond having a formal semantics of the language: Specifying a large low-level requirement in a logic is formally sound, but it is also a hassle because the specification quickly becomes convoluted. Hence, the intuitive and formal meaning becomes opaque to the specifiers, rendering it a potential source for error. Exacerbatingly, the development process of such systems concerns people of vastly different backgrounds, including computer scientists, domain experts, engineers, and external reviewers among others. Even though some people do not need to understand every detail of the specification, a specification should allow non-experts to quickly discern its quintessence. Moreover, the language should support specifiers in their task by reducing the cognitive burden through modularity and allowing them to focus on the task at hand rather than distracting them with boilerplate code or implementation details. Lastly, CPS have strictly limited resources, hence the monitor has to operate within its statically allocated space, power, and time bounds. All these requirements manifest in the following five design maxims for RTLola.

First and foremost, M1: Clarity requires specifications to be comprehensible for everyone involved in the development process. This involves a syntax that is explicit and features semantic names. Consider, for example, the C standard function `int strcmp(const char *str1, const char *str2)`. Without prior or contextual knowledge, it is difficult to apprehend the semantic of the function. The name is a vowel-less abbreviation of "string compare", taking two strings as input, which are constant character pointers. With this in mind, the return value of integer type might be a surprise, too. Though

M1: Clarity

signed integers often represent boolean values in C, in this case the result is of ternary nature, indicating which string — if any — is "less" than the other.

Such a function contradicts the maxim of clarity. Rather, functions should have names comprehensible without context and should eradicate surprises such as the return value. In this particular example, strings can be checked for equality with the =-operator. The <-operator is not defined for strings since their order is highly ambiguous: the comparison could be alpha-numerically, either case-sensitive or case-insensitive, or it could primarily refer to the length, either in bytes or in characters[4]. In lieu of the direct comparison, specifiers have access to dedicated functions e.g. for checking the character-length of a string. Note that when comparing strings, the = operator requires fewer keystrokes than `strcmp`, but this is not generally the case. Benefiting clarity usually detriments brevity and hence convenience.[5] As an example, consider the `x[0, 3]` statement of an older syntax for RTLola [Fay19]. This translates equivalently to the significantly more verbose syntax `x.hold().defaults(to: 3)`. The former syntax implicitly performs a 0-order hold operation on the `x` stream. If the hold fails because the stream has never produced a value, the monitor substitutes the value 3. The trade-off is clear: the old syntax is brief and cryptic whereas the new one is lengthy and explicit.

MII: RESOURCE-AWARENESS

Not only are strings a good example for the maxim of clarity, it also exemplifies the next maxim, MII: RESOURCE-AWARENESS. RTLola supports only data types of fixed size such as strings of a pre-defined maximum length. While this severely limits the expressive power, it allows for a significantly more resource-aware execution. Similarly, consider integer types in Python and Standard ML. Here, the type *int* refers to an unbounded data type. It is up to the compiler or interpreter to decide how many bits are required to represent a certain value. In resource-aware languages like RTLola or Rust, numeric types have a specific bit-width such as `Int16` or `i16`, respectively. Access to this information allows for a more precise analysis regarding the resource consumption. Moreover, the explicit representation furthers the maxim of clarity. Yet, this decision does not come without drawbacks: both of the presented maxims stand in a conflict with the third one.

MIII: DECLARATIVENESS

MIII: DECLARATIVENESS states that specifiers have to provide as little information as possible, allowing them to neglect implementation details such as eviction of data from memory, the order of stream evaluation, and how to precisely compute sliding window aggregations. Evidently, RTLola does not fully comply to this maxim as it is subordinate to MI: CLARITY and MII: RESOURCE-AWARENESS: RTLola embraces declarativeness only when there is no conflict between different maxims. Hence, memory management is entirely internal, and so is the order of evaluation. However, note that this order has a direct impact on the semantics of stream accesses, so clarity arguably demands an explicit statement of the order. Yet, since all operations influencing the order are designed in a way that they convey a natural

[4]This can make a difference, depending on the character encoding.

[5]Note that the functions are not perfectly comparable: C's handling of the comparison has other benefits as the function is multi-functional, resulting in smaller binaries and better code-locality. Yet, the main point of the comparison still stands.

meaning, RTLola chooses the order in a way that adopts this intuition. So the language sacrifices a little clarity in favor of a lot of declarativeness. As an example, suppose the expression of a stream σ_1 refers to another stream σ_2. It is natural to assume that the monitor accesses the current, most recent value of σ_2 rather than any older value. Thus, the monitor will evaluate σ_2 before σ_1 such that the evaluation of the latter has access to the most recent value of its target. This detail is hidden from the specifier.[6] Another major example of declarativeness is the expression `a.aggregate(over: 3s, using: ∫)`, which leaves a multitude of implementation details implicit due to their irrelevancy for the specifier.

The next maxim is commonly found in programming languages and style guides: MIV: MODULARITY. It dictates that certain elements of the code or specification do not or cannot influence each other. A prime example for modularity in RTLola is freedom of side effects, which also furthers declarativeness and clarity. To understand this, consider a logical conjunction in C. During an execution, the left operand will be evaluated first. If the result is `false`, the right operand will not be evaluated, a process called short-circuit evaluation. If this were different, it might change the semantics of the execution because the evaluation of the right operand can be tainted with side effects. Hence, being unaware of short-circuit evaluations or the left-to-right evaluation order, can lead to bugs even though both mechanisms are implicit. On the contrary, freedom of side effects as a consequence of modularity in RTLola eliminates the need to know about these mechanisms. The maxim also manifests in stream accesses. While the semantics of a stream is directly influenced by all streams it actively accesses, it is agnostic towards which streams access *it*. Hence, given a specification, any additional stream declaration added later will be just that: additions.

MIV: MODULARITY

The last maxim is MV: DISTRACTION-FREEDOM. It states that a specification should not contain unnecessary information. This is most evident when considering stream accesses. If a stream σ_1 accesses σ_2, it can be opaque for the specifier whether the access is fallible as it depends on the timing of both participating streams. If RTLola were unable to determine this information, it would require specifiers to declare a default value even though it is meaningless for infallible accesses. To circumvent this, RTLola performs an intricate analysis (cf. Section 2.3) such that it only requires specifiers to supply a default value if there is an execution which requires it. For a further example, recall that MII: RESOURCE-AWARENESS leads to type names explicitly stating the number of bits required to store a single value of it. Requiring specifiers to spell out the type of each stream clutters a specification with duplicated information. Hence, similar to modern imperative programming languages and most functional languages, RTLola allows for type omission and instead infers types whenever possible.

MV: DISTRACTION-FREEDOM

→ Sec. 2.3, p. 42

In summary, the design maxims impacting RTLola are as follows:

[6]More on the order of stream evaluation in → Sec. 2.5.3, p. 78.

Mi: Clarity Usage of semantic intuitive names, such that the essence of a specification is easy to grasp for non-RTLola experts.

Mii: Resource-Awareness Memory and power consumption as well as running time needs to be statically defined and low.

Miii: Declarativeness Determine implementation details automatically unless this sacrifices clarity or resource-awareness.

Miv: Modularity Streams are independent, so they cannot impact others, and expressions are free of side effects.

Mv: Distraction-Freedom Specification contains no superfluous, duplicate, or meaningless information.

The next section details the syntax of RTLola. Here, design decisions were a direct result of these maxims.

2.2. Syntax

RTLola has two different syntaxes, the "regular" one and the *desugared* one. The regular syntax is what specifiers would write and allows for eliding certain syntactic fragments or usage of syntactic shortcuts, also known as *syntactic sugar*. This is commonly known in conventional programming languages: In C$^\sharp$ the variable declaration `var x = 3` constitutes type elision and in Rust's method call syntax `a.foo(b)` is syntactic sugar equivalent to a function call `A::foo(a, b)` assuming `a` is of type `A`. Along this line, regular RTLola syntax allows specifiers to omit the type of a stream, prompting the RTLola toolkit to infer the type. Similarly, the expression `x.hold(or: 12)` is a shorthand for `x.async().offset(by: 0).defaults(to: 12)`. Removal of this sugar yields — as the name suggests — a specification in its desugared form. This syntax permits less variety and is thus easier to handle. The difference in variance is immediately evident when comparing the syntax diagrams for regular RTLola (Figure 2.1) against desugared RTLola (Figure 2.2).

Desugaring

Syntactic Sugar

The remainder of this section discusses the desugared syntax, outlines how to obtain it from a regular RTLola specification, and introduces some additional syntactic sugar.

2.2.1. Type Annotations

Simply put, *type annotations* state the type of a stream. Since types play a central role in RTLola, type annotations are an equally integral component of the syntax. Hence, in the desugared syntax, each stream has several type annotations. Declaring this information manually is cumbersome, which is why type annotations can typically be omitted in the regular syntax. This is only necessary when either the type inference algorithms fails to determine the type, or the specifiers want to deviate from the default semantics. For example, by default, a stream has a type indicating that it should be evaluated whenever there is a change in a stream on which it depends. This is common practice in user interface frameworks such as SwiftUI, where the `@State` property wrapper for a variable states that a change in the variable should trigger a re-evaluation of the interface. Omission of the wrapper leads to a deferral of the computation until the next scheduled update[7]. The analogue in RTLola is a frequency annotation like `@1Hz`, which prompts the monitor to defer the computation to the next time the period of 1 s passed. Note that in this analogy, RTLola has the opposite default behavior from SwiftUI. The rationale behind choosing this default behavior is that timely responses are critical in monitoring. As a result, a delayed computation is counter-intuitive unless explicitly requested by the specifier. Hence, type omission leads to the default, immediate behavior whereas an explicit type annotation allows specifiers to deviate from the default.

Type Annotation

A full type annotation consists of several atomic type annotation, each conforming to one of the following kinds:

[7]In SwiftUI, this is the next time the View is computed.

Value Type If a stream has value type $\nu \in \mathcal{N}$, then a single value v of the stream is drawn from ν, i.e., $v \in \nu$. This is the only type that describes the *shape* of a stream and its values as opposed to timing related type components. Value types are common in programming languages, typical examples include "String" in Java, "int" in C, "single" in MATLAB, or "(i8, f64)" in Rust.
Appearance: Value types are preceded by a colon. Example: `output a: Int32` where `Int32` is the value type of `a`.

Periodic Type A stream with a periodic type $\pi \in \mathcal{P}$ is computed periodically with a period of π^{-1}. The period relates to the real, physical time such as $\pi^{-1} = 0.2\,\mathrm{ms}$. This relation can either be with respect to the global time, i.e., the starting time of the monitor, indicated by a `global` keyword, or local. The exact meaning will become clear shortly.
Appearance: Periodic types are surrounded by `@` at the front and a unit of frequency at the back, e.g., `output a eval @ 3Hz` where $3\,\mathrm{Hz}$ is the periodic type of `a`.

Event-Based Type The timing of a stream can also depend on the occurrence of a certain event, such as a new measurement of a sensor or the reception of a message, rather than the physical time. These events correspond to updates to input streams. Hence, an event-based type $\varepsilon \in \mathcal{E}$ is a condition on the input streams determining whether the respective stream needs an update. An example for event-based types is $\mathcal{E} = 2^{\mathcal{S}^{\downarrow}}$ intuitively stating that a stream σ with event-based type $\varepsilon \in 2^{\mathcal{S}^{\downarrow}}$ is updated if every member $\sigma^{\downarrow} \in \varepsilon$ is updated as well. RTLola employs a more expressive model where $\mathcal{E} = \mathbb{B}^{+}_{\mathcal{S}^{\downarrow}}$, so — intuitively — an event-based type states a positive boolean condition[8] on the input streams. The stream is only updated if the condition holds.
Appearance: Due to some inherent similarity between periodic and event-based type, the `@` symbol also precedes the latter. However, it lacks any unit of measure. Example: `output a eval @ b ∨ c` declares the event-base type $a \vee b$ for `a`.

Semantic Type The semantic type $\zeta \in \mathcal{Z}$ of a stream relates to the current state of the monitor, i.e., it defines a semantic criterion under which an update should take place. Such an annotation can for example be an expression allowing for all or a subset of the operations found in RTLola stream expressions.
Appearance: Semantic types are syntactically identical to expressions, yet, they have an introductory `when` keyword. Example: The `when` clause in `output a eval when b > 7` declares the semantic type $b > 7$.

Note that neither of these atomic types in isolation suffices to fully describe the type of a stream. A value type for example does proclaim the shape of its values but lacks information regarding the timing of a stream and thus has to be paired with a periodic or event-based type. Moreover, a semantic type indicates neither the timing nor the shape

[8]cf. Definition A.7, p. 211

of stream values. Hence, a full description of the type of a stream requires multiple components.

While Section 2.3 contains all details, roughly speaking, a stream has four type components, one value type and three *timelines*. Each timeline consists of a base clock, i.e., either an event-based or a periodic type, coupled with a *semantic filter*. Intuitively, the first timeline refers to the behavior of the stream itself. The second and third timeline are parameters for dynamic stream creation, i.e., the *spawn and the close condition*. The spawn condition determines when an instance of a stream will be created, the close condition when it ceases to exist.

➔Sec. 2.3, p. 42

Timeline

Semantic Filter

Spawn and Close Condition

2.2.2. Streams

First and foremost, an RTLola specification consists of stream declarations, as can be seen in Figure 2.2. All three kinds of streams — inputs, outputs, and triggers — share a similar structure. They start with a keyword indicating their *kind*, i.e., either `input`, `output`, or `trigger`. Next is the *name*, which serves as a unique identifier throughout the specification. Note that triggers are an exception to this rule as they do not require a name since they cannot be referenced from other streams.[9] Moreover, output stream names are followed by a potentially empty, comma-separated list of parameters. Each parameter is a pair of a name and a value type annotation.

Stream Name

Remark 2.1 (Trigger Names)**.** Triggers lacking a name seems counter-intuitive to Mi: Clarity since names convey an intuition on the semantics of the trigger. However, a trigger declaration demands a string message to be displayed to the user or system in case of a trigger violation. The message usually contains an identifier of the respective safety requirement embodied by the trigger. Hence, requiring it to additionally carry a name leads to duplication of information, going against Mv: Distraction-Freedom and maintainability.

After the name come type annotations. These annotations have to unambiguously declare the full type of a stream, i.e., a value type and three timelines. For this, the requirements on the syntax vary strongly between different stream kinds. Input streams represent incoming events, hence they always have a specific event-based type, plus they cannot be parametrized, nor semantically filtered.[10] Hence, input streams merely require a value type annotation. Trigger declarations are similarly sparse in type annotations since their purpose in the desugared syntax is to mark certain output streams as safety conditions without carrying logic themselves. Hence, their value type is of binary nature, and they mirror the marked output streams in terms of timing, which alleviates the need for type annotations in the trigger declaration.

[9]Note that the node labeled "name" in the syntax diagram for triggers does not refer to the name of the trigger but the name of the target of the trigger

[10]If only certain input values are relevant, specifiers have to declare an output stream mirroring and filtering the input stream.

Lastly, the type annotation of an output stream is most elaborate. The first part is the value type annotation, just as for the input stream. The next three parts structurally resemble the life-cycle of a stream instance, i.e., creation, evaluation, and termination, where all parts largely follow the same pattern. They start with the `spawn`, `eval`, and `close` keyword, followed by either a periodic or an event-based type annotation. Separated by the `when` keyword follows a semantic type annotation.

The last component of each part is an expression with an introductory `with` keyword. These are no longer part of the type of the stream. Instead, they are instructions for the monitor on how to compute the parameter values of a stream instance for the expression in the spawn-clause, or the values of the stream itself, in case of the eval-clause. The close-clause lacks this last component.

The last piece of syntax is unique to triggers. As mentioned before, triggers merely mark output streams as safety conditions. To this end, a trigger declaration states the target stream name, followed by a string message that is to be conveyed to the system or user in case of a violation.

Differences to the Regular Syntax There are three major differences between the desugared and regular syntax of streams. First, the regular syntax tags several parts of the specification as optional if intuitive defaults exist. This most notably affects type annotations, yet it also allows for omitting an empty pair of parentheses. Second, it provides shorthand notations for common patterns in a specification. These syntactic sugars for example allow for drastically abbreviating an output stream declaration for simple streams. Since the list of syntactic sugars and their desugaring grows steadily, the syntax diagram only contains the output stream shorthand. Section 2.2.4 introduces some of the most useful ones.

→Sec. 2.2.4, p. 39

Third, there are two kinds of output stream declarations in the regular syntax: parametrized and non-parametrized. The former requires a non-empty list of stream parameters and a `with`-expression in the spawn-clause to supply parameter values. The latter rejects both a parameter list and a `with`-expression. Instead, it mandates a clock type annotation in the spawn declaration. The desugared syntax forgoes this difference and instead requires all of these clauses to be present, though potentially with semantically null values. Their appropriateness is checked on a semantic level during the type inference. Having a syntactic distinction is generally more convenient than a semantic one and the option to omit unnecessary values agrees with Mv: Distraction-Freedom.

2.2.3. Expressions

Figure 2.3 illustrates the syntax of RTLola expressions. The first few cases are conventional: unary and binary operations, conditional expressions, and calls to pre-defined, potentially polymorphic functions. The polymorphism is resolved by supplying the desired value types in angle brackets for the desugared syntax, and implicitly via type inference in the regular syntax. Function arguments are listed in parentheses.

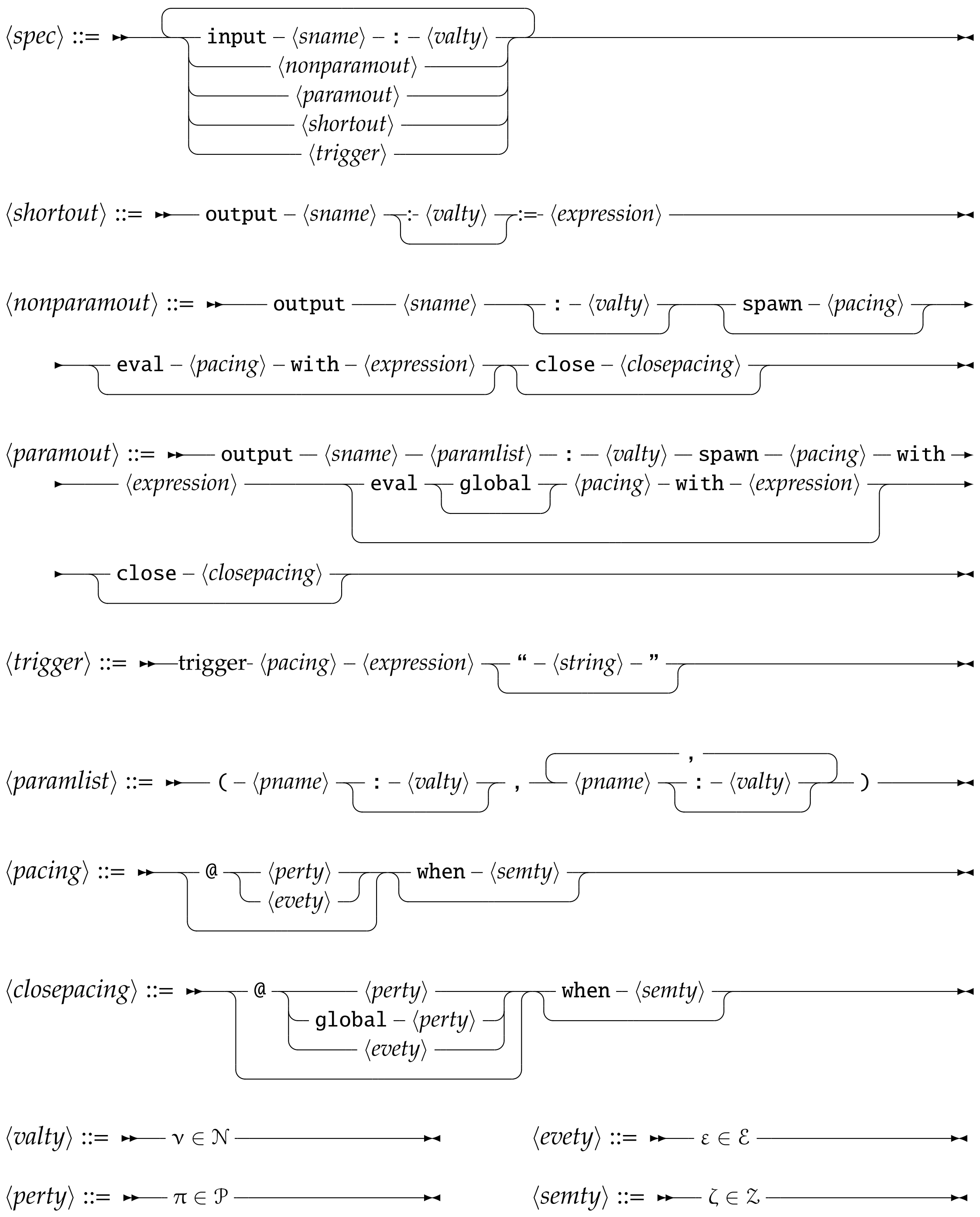

⟨spec⟩ ::=
input – ⟨sname⟩ – : – ⟨valty⟩
⟨nonparamout⟩
⟨paramout⟩
⟨shortout⟩
⟨trigger⟩
⟨shortout⟩ ::= output – ⟨sname⟩ :- ⟨valty⟩ :=- ⟨expression⟩
⟨nonparamout⟩ ::= output ⟨sname⟩ : – ⟨valty⟩ spawn – ⟨pacing⟩
eval – ⟨pacing⟩ – with – ⟨expression⟩ close – ⟨closepacing⟩
⟨paramout⟩ ::= output — ⟨sname⟩ — ⟨paramlist⟩ — : — ⟨valty⟩ — spawn — ⟨pacing⟩ — with
⟨expression⟩ eval global ⟨pacing⟩ – with – ⟨expression⟩
close – ⟨closepacing⟩
⟨trigger⟩ ::= trigger- ⟨pacing⟩ – ⟨expression⟩ " – ⟨string⟩ – "
⟨paramlist⟩ ::= (– ⟨pname⟩ : – ⟨valty⟩ , ⟨pname⟩ : – ⟨valty⟩ ,)
⟨pacing⟩ ::= @ ⟨perty⟩ ⟨evety⟩ when – ⟨semty⟩
⟨closepacing⟩ ::= @ ⟨perty⟩ global – ⟨perty⟩ ⟨evety⟩ when – ⟨semty⟩
⟨valty⟩ ::= ν ∈ 𝒩
⟨evety⟩ ::= ε ∈ ℰ
⟨perty⟩ ::= π ∈ 𝒫
⟨semty⟩ ::= ζ ∈ 𝒵

Figure 2.1.: Railroad diagram for the regular RTLola syntax.

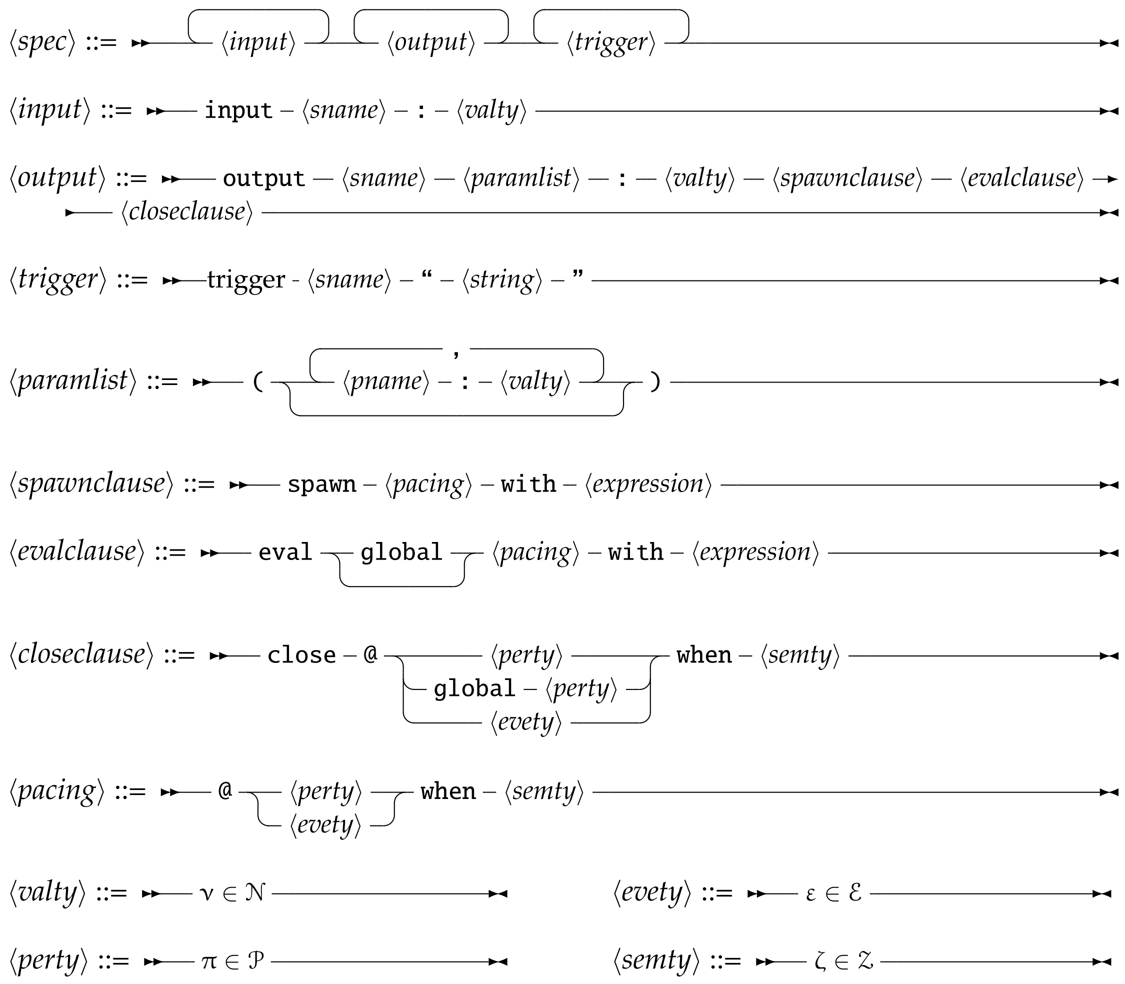

⟨spec⟩ ::= ⟨input⟩ ⟨output⟩ ⟨trigger⟩
⟨input⟩ ::= input – ⟨sname⟩ – : – ⟨valty⟩
⟨output⟩ ::= output — ⟨sname⟩ — ⟨paramlist⟩ — : — ⟨valty⟩ — ⟨spawnclause⟩ — ⟨evalclause⟩ — ⟨closeclause⟩
⟨trigger⟩ ::= trigger - ⟨sname⟩ – “ – ⟨string⟩ – ”
⟨paramlist⟩ ::= (⟨pname⟩ – : – ⟨valty⟩ ,)
⟨spawnclause⟩ ::= spawn – ⟨pacing⟩ – with – ⟨expression⟩
⟨evalclause⟩ ::= eval global ⟨pacing⟩ – with – ⟨expression⟩
⟨closeclause⟩ ::= close – @ ⟨perty⟩ global – ⟨perty⟩ ⟨evety⟩ when – ⟨semty⟩
⟨pacing⟩ ::= @ ⟨perty⟩ ⟨evety⟩ when – ⟨semty⟩
⟨valty⟩ ::= ν ∈ 𝒩
⟨evety⟩ ::= ε ∈ ℰ
⟨perty⟩ ::= π ∈ 𝒫
⟨semty⟩ ::= ζ ∈ 𝒵

Figure 2.2.: Railroad diagram of the desugared RTLola syntax.

The next three cases constitute stream accesses. Here, a stream reference is comparable to a variable access in other programming languages. These start with the name of a stream, followed by an optional list of parameters. This allows for discerning the stream instance targeted by the access. Moreover, they have a `.sync` or `.async` suffix. Intuitively, this suffix indicates whether timing of the streams are couples or decoupled, respectively. This has consequences for the type inference and semantics.

The first stream access is a regular access, i.e., it refers to a single stream value. To this end, it declares a stream reference and provides a negative integer offset argument to the `offset` access modifier. If the offset is n, the stream access refers to the $-n+1^{\text{st}}$-to-last value of the accessed stream, so $n = 0$ yields the latest value, $n = -1$ the second-to-last and so on. Note that such a value does not necessarily exist, hence an access can be *fallible* resulting in an empty value: `None`. Next are two flavors of aggregation, syntactically only differing in the name of the first argument. Both aggregations require two arguments: a time declaration t and an aggregation function $\gamma \colon T_1^* \to T_2$. Suppose $t = 3\,\text{s}$ and $\gamma = \sum$. Then, when aggregating the values of stream σ, the expression yields the sum of all values of σ which the monitor computed or received within the last three seconds. The difference of the two variants takes effect when targeted stream instance exists for less than $3\,\text{s}$. In this case, an aggregation with argument name `over` aggregates all values present whereas an `over_exactly`-aggregation yields `None` until the $3\,\text{s}$ have passed. The last expression allows for transforming `None`-values into usable data by providing a default value. The monitor substitutes the default in when the preceding expression failed. This kind of expression has counterparts in most modern or functional programming languages featuring optional values or generally monads, like in Swift `expr ?? dft` or Rust `expr.unwrap_or(dft)`.

Remark 2.2 (Naming Analysis)**.** Note that the syntax diagrams contain different categories for different kinds of names such as `sname` or `pname`. For the purpose of parsing, this differentiation is meaningless: all names follow the same rule[11]. However, it plays a central role in the naming analysis on the abstract syntax tree. Here, the analysis checks that every function occurs in the RTLola standard library, every stream reference has a matching declared stream, and every occurrence of a parameter or stream name in an expression or type annotation refers to a parameter of the enclosing stream or an input stream.

2.2.4. Desugaring

The process of desugaring is concerned with transforming a specification into the canonical, desugared form. This transformation requires context-agnostic changes such as replacing shorthand notations, and also context-sensitive additions. These mainly stem from syntax elements such as type annotations which are optional in the regular

[11]Every identifier has to comply with the Unicode identifier standard according to UAX31-D1 `https://www.unicode.org/reports/tr31/#D1`; last accessed: 07.02.2022.

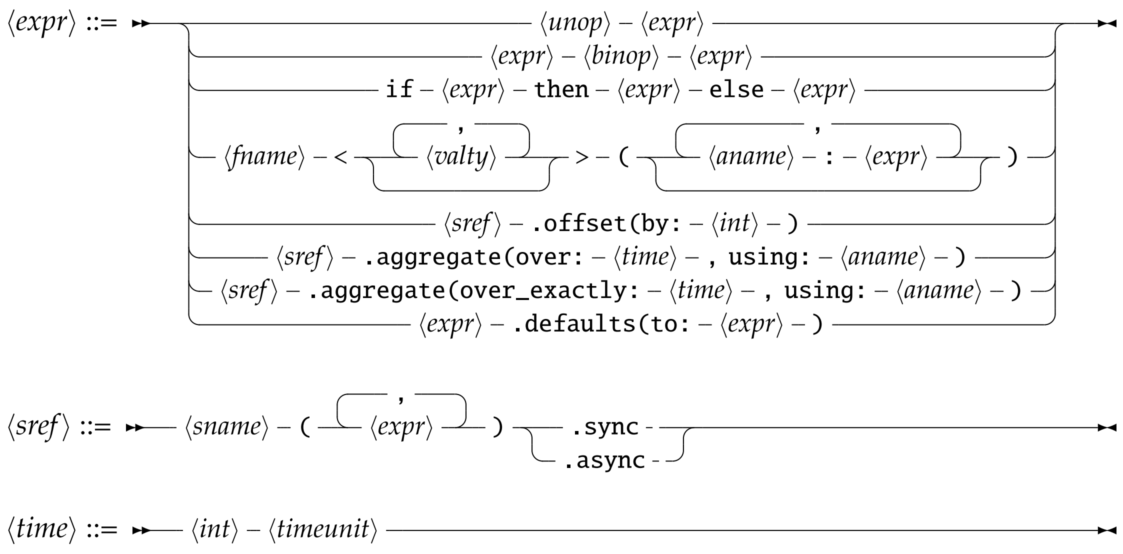

⟨expr⟩ ::=
⟨unop⟩ – ⟨expr⟩
⟨expr⟩ – ⟨binop⟩ – ⟨expr⟩
if – ⟨expr⟩ – then – ⟨expr⟩ – else – ⟨expr⟩
⟨fname⟩ – < – ⟨valty⟩ , – > – (– ⟨aname⟩ – : – ⟨expr⟩ , –)
⟨sref⟩ – .offset(by: – ⟨int⟩ –)
⟨sref⟩ – .aggregate(over: – ⟨time⟩ – , using: – ⟨aname⟩ –)
⟨sref⟩ – .aggregate(over_exactly: – ⟨time⟩ – , using: – ⟨aname⟩ –)
⟨expr⟩ – .defaults(to: – ⟨expr⟩ –)
⟨sref⟩ ::= ⟨sname⟩ – (– ⟨expr⟩ , –) – .sync / .async
⟨time⟩ ::= ⟨int⟩ – ⟨timeunit⟩

Figure 2.3.: Syntax of desugared RTLola expressions. In the regular syntax, the `.sync` and `.async` suffixes are optional and so are angle brackets provided the respective list is empty.

syntax and mandatory in the desugared syntax. The following enumeration outlines the first kind of transformations:

Triggers If the expression e of a trigger is more than just a stream name, or if the trigger has type annotations, create a new stream with expression e. Copy type annotations over. Replace e in the trigger by the name of the new stream and remove all type annotations.

Outputs For non-parametrized streams, add delimiters for the empty list of parameters. Also, replace outputs of shape `output a: T := e` with `output a: T eval with e` where `: T` is optional.

Hold Expressions Replace `.hold()` by `.async.offset(by: 0)`.

Aggregations Prefix `.aggregation(...)` with `.async`.

Offsets Prefix any occurrence of `.offset` not preceded by `.async` with `.sync`. Also, prefix all stream accesses which have neither `.async` nor `.sync` with `.sync.offset(by: 0)`

Ordering All streams are reordered such that outputs follow inputs and triggers follow outputs.

Partial Desugaring

Applying these steps yields a *partial desugaring* of a specification, a syntax in-between regular and desugared. This intermediate representation eases the type inference which in turn enables the full desugaring. For this, assume type information is available. Then, for each output stream, including ones created in the process of desugaring trigger, add type information. This affects the value type and all three timelines.

Additional Syntactic Sugar

Named Function Argument

RTLola employs an array of syntactic sugars as shorthand notation for common operations, drastically boosting usability. In particular in combination with *named function arguments*, as in `s.offset(by: -1)`, they do not negatively impact clarity. Several programming languages like Python, VBA, and C♯ allow programmers to use named function arguments, whereas relatively few like Swift enforce them or provide fine-grained control. Named function arguments are relatively rare in programming languages, with the most prominent example being Swift. However, it is common practice for integrated development environments to display hints on the parameter names when calling a function. Functions in the RTLola standard library and also function notations in syntactic sugar, every argument can be unnamed or named. The former is useful for functions where the purpose of the argument is abundantly clear, such as in the `sqrt` function. The latter case forces specifiers to supply the name aiming for better readability and disambiguation of arguments. Consider, for example, an expression containing `.hold(1)` versus `.hold(or: 1)`. The former variant contains no hint on the semantics of the argument whereas it is clear that the `or`-argument provides some kind of alternative. For a fallible operation such as a 0-order hold, it is no leap of logic to realize that the argument is the default value. Similarly, consider an expression with `.offset(-1, -2)`. Since the value type of both arguments is identical, the order is opaque, inviting error. In contrast, `.offset(by: -1, or: -2)` provides clarity.

The following examples show syntactic sugar and their desugared syntax in juxtaposition.

```
output a := x.hold(or: y)
output b := x.offset(by: n, or: y)
output c := x.last(or: y)
output d := delta(c) // or: δ(c)
output e := self.last(or: x)
output f: IPv4 := x
output h eval @xHz close immediately
output g @zHz := x.γ(over: ∞)
```

```
output a := x.hold().defaults(to: y)
output b := x.offset(by: n).defaults(to: y)
output c := x.offset(by: −1, or: y)
output d := c − c.last(or: c)
output e := e.last(or: x)
output f: UInt32 := x
output h eval @xHz close @xHz when true
output g @zHz :=
      ⊕γ(g.last(or: ε), x.γ(over: z^-1))
```

The syntactic sugar displayed in stream `e` is clearly beneficial in terms of clarity. However, it truly shines for parametrized streams since they often refer to themselves, for example when they are supposed to terminate as soon as they produce a certain value. These self-references are lengthy when using descriptive stream- and parameter names. Thus, self-explanatory abbreviations are a boon for such specifications.

➔Listing 2.4, p. 25

Example 2.1. Recall the running example specification from Listing 2.4. The partial desugaring looks as follows:

```
input oid: Int8
input distx: Float32
input disty: Float32
input spdx: Float32
input spdy: Float32
input pause_cmd: NoValue
input obj_lost: NoValue

output velo() eval with sqrt(
    spdx.sync.offset(by: 0)*spdx.sync.offset(by: 0)
    + spdy.sync.offset(by: 0)*spdy.sync.offset(by: 0)
)
output obey_pause()
    spawn @pause_cmd with ()
    eval @0.5Hz with velo.async.integrate(over: 1.5s, using: ∫) < ε
    close @0.5Hz when true
output sx @distx eval with spdx.async.offset(by: 0).defaults(to: 0.0)
output sy @disty eval with spdy.async.offset(by: 0).defaults(to: 0.0)
output collision_possible eval with angle(
        (distx.sync.offset(by: 0), disty.sync.offset(by: 0)),
        (sx.sync.offset(by: 0), sy.sync.offset(by: 0))
    ) > ε_φ
output ttc(id: Int8)
    spawn @oid with oid.sync.offset(by: 0)
    eval
        when collision_possible.sync.offset(by: 0)
        with min(
            if sx.sync.offset(by: 0) ≠ 0
                then distx.sync.offset(by: 0) / sx.sync.offset(by: 0)
                else disty.sync.offset(by: 0) / sy.sync.offset(by: 0),
            if sy.sync.offset(by: 0) ≠ 0
                then disty.sync.offset(by: 0) / sy.sync.offset(by: 0)
                else distx.sync.offset(by: 0) / sx.sync.offset(by: 0)
        )
    close @obj_lost ∧ oid when oid.sync.offset(by: 0) = id
```

△

2.2.5. Stream Notation

The following definitions clarify some notation which is convenient when arguing about RTLola streams.

Let $\mathcal{S}^{\downarrow}, \mathcal{S}^{\uparrow}$, and $\mathcal{S} = \mathcal{S}^{\downarrow} \dot{\cup} \mathcal{S}^{\uparrow}$ be the set of *input stream names*, *output stream names*, and *stream names*, respectively. For a given stream $\sigma \in \mathcal{S}$, p^{σ} denotes the number of parameters declared for this stream in the specification. This number is always 0 for input streams. Further, let $\mathcal{T}_1^{\sigma} \dots \mathcal{T}_{p^{\sigma}}^{\sigma}$ be the potentially nullary tuple of parameter types, and $\mathcal{T}^{\sigma}$ be the value type of σ. A *stream identifier* $(\sigma, \vec{p})$ is a pair of a stream name and a tuple of concrete parameter values with $|\vec{p}| = p^{\sigma}$ plus a creation date. The function *SId* maps a stream name to the set of possible stream identifiers:

Def. Stream Names

Def. Stream Identifier

$$\mathit{SId}\colon \mathcal{S} \to \bigcup_{\sigma \in \mathcal{S}} \{\sigma\} \times \mathcal{T}_0^{\mathcal{S}} \dots \mathcal{T}_{p^{\sigma}}^{\mathcal{S}^{\uparrow}} \times \mathbb{R}^{+}$$

Furthermore, dot notation represents *access to different constituents* of streams. Here, $\sigma^{\uparrow}.\mathit{param}$ accesses the parameters of $\sigma^{\uparrow}$, $\sigma^{\uparrow}.\mathit{spawn}$ its spawn clause, $\sigma^{\uparrow}.\mathit{eval}$ its evaluation clause, $\sigma^{\uparrow}.\mathit{close}$ the close clause. For a clause c, the functions $\mathit{clock}(c)$, $\mathit{filter}(c)$, and $\mathit{expr}(c)$ provide access to the clock, semantic filter, and expression of a clause, respectively. When applied on a stream, it implicitly refers to its evaluation clause. Moreover, recall that the clock can either be event-based or periodic. This distinction is solved via meta-variables, i.e., $\mathit{clock}(\sigma) = \mathit{clock}(\sigma.\mathit{eval}) = \pi$ indicates that the evaluation clause of σ has a periodic clock with period π whereas $\mathit{clock}(\sigma) = \mathit{clock}(\sigma.\mathit{eval}) = \varepsilon$ indicates that σ has the event-based type ε.

Def. Constituent Access

Given an expression *ex*, $\mathit{ex}\,[x \mapsto y]$ is an expression where all occurrences of x were replaced by y, for details regarding alpha-renaming, confer Definition A.5. Finally, $\circ\colon X^n \times X \dot{\cup} \{\bot\} \to X^n \cup X^{n+1}$ is the *gap-resistant sequence concatenation* defined as:

→ Def. A.5, p. 210

Def. Gap-Resistant Concatenation

$$v_1 \dots v_n \circ x = \begin{cases} v_1 \dots v_n x & \text{if } x \neq \bot \\ v_1 \dots v_n & \text{otherwise} \end{cases}$$

2.3. Supportive Type System

The RTLola type system follows the paradigm "Everything is a Stream". Hence, the types of stream declarations and expressions are indistinguishable since in essence, expressions are streams as well. Each stream is a sequence of values. As such, the type system answers two questions: How do these values look and when are they produced. These questions correspond to the *value type* and *timeline* of a stream in RTLola. The first question is found in most conventional programming languages where the value type is simply called the *type* of a value or expression. Common examples are the `int` type in C or the `nuint` type in C$^\sharp$. The second question is specific to stream-based languages.

In its core, the type system of RTLola mainly revolves around timelines. In a nutshell, a timeline induces a set of points in time. These points state when a timeline progresses and when it starts or ceases to do so.

For the foundation of a timeline, recall that streams run either with a fixed frequency with respect to the real-time, or based on when events reach the monitor. The notion of a *clock* reflects this distinction. Every timeline has a pacing that is based on one of two discrete clocks. The *periodic clock* ticks in the frequency of the greatest common denominator (gcd) of all periods that occur in the specification. Semantically analogously, the *event-based clock* ticks with every incoming events that updates at least one input stream. Refer to the center three timelines of Figure 2.4 for an illustration of the relation between the real time axis and the clocks. Note that both relations are dynamic, i.e., only at runtime can every tick of either timeline be mapped to a specific point in time on the real-time axis. Note further that the periodic clock is *isochronous*, i.e., when the starting point of the timeline is known, then so are the points in time of all its ticks. This is not the case for the event-based clock.

Clock

The clocks form a baseline for timelines as they represent the fastest possible rate. This rate can be scaled down by applying *filters*. The first kind of filter are *static filters*. Their effect is based on the timeline itself: any annotated frequency in the specification, for example, is slower or equally fast as the least common multiple (lcm) for all frequencies[12]. Hence, if they are strictly slower, they naturally filter out a number of points. Similarly, an event-based clock that is a disjunction of a real subset of inputs $S \subset \mathcal{S}^{\downarrow}$ skips all updates which only cover $\mathcal{S}^{\downarrow} \setminus S$. This stands in contrast to *semantic filters* which impose semantic restrictions on the state of the monitor. Practically speaking, semantic filters are just RTLola expressions. A point in time is skipped iff evaluating the expression at this time yields *false*. Referring back to Figure 2.4, the top- and bottom-most timelines show a semantic filter in action.

Static Filter

Semantic Filter

The next two components formulate the dynamic aspects of timelines. The natural intuition is that timelines are eternal; they start with the beginning of time and never cease. However, since timelines represent streams, this intuition is not entirely adequate. A stream can be created after a while, produce data afterwards, and cease to do so

[12]And thus the gcd of all periods.

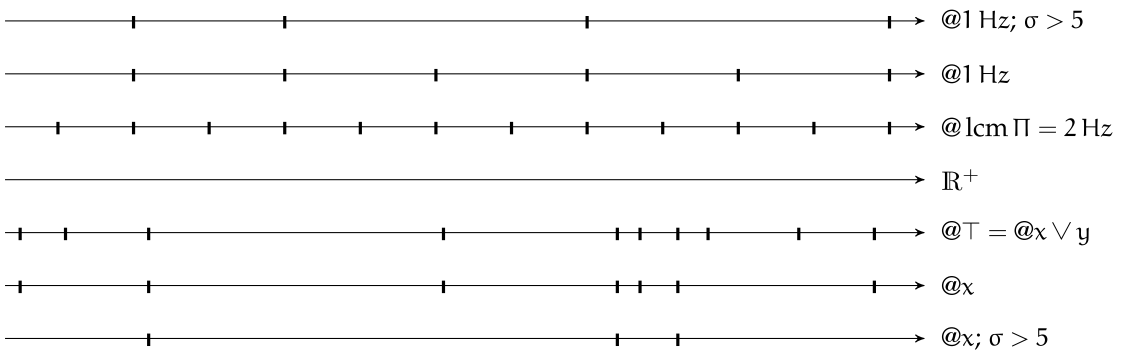


Figure 2.4.: Illustration of basic timelines. The timeline in the center represents the real time axis. Periodic and event-based timelines are above and below the real time axis, respectively. Moving further outwards indicates stronger filtering, first by applying a static, then a semantic filter.

some time later. This process can repeat, i.e., after termination of a timeline, it can be re-spawned and re-terminated. As a result, both the spawn and the termination are sets of points in time and thus timelines. As such, they refer to a clock, are equipped with static and dynamic filters, and can be dynamic themselves.

Before discussing the implication of the recursive definitions, consider the following caveat: Imaging a timeline is spawned by an event-based timeline and itself refers to the periodic clock. There are two appropriate interpretations of this reference: either the timeline refers to the *global-periodic clock* or the point of spawn induces a new, *local-periodic clock*. The following example illustrates the point:

Local- and Global-Periodic Clocks

Example 2.2 (Local- and Global-Periodicity). Recall the command compliance example in Listing 2.2. Here, a drone can receive commands dynamically from a central command unit and has to comply within a set amount of time. In the specification, the output stream is a timeline that is spawned upon reception of the command. It then ticks periodically with $10\,\mathrm{mHz}$ and terminates at the same rate. Here, the period of $(10\,\mathrm{mHz})^{-1} = 100\,\mathrm{s}$ is relative to the local, spawned timeline rather than the global clock. This ensures that the first evaluation of the stream happens $100\,\mathrm{s}$ after its creation rather than at the next point in time divisible by $100\,\mathrm{s}$. Confer Figure 2.5 for an illustration of the example. Both variants of `compliance` are created when the monitor receives a command. Yet, the termination of `compliance` occurs 100 seconds after its creation whereas the termination period of `compliance'` is relative to the global real time axis. The latter case leads to an early termination which does not reflect the intended semantics.

➔ Listing 2.2, p. 23

Now consider the central command unit. This unit can receive messages from drones announcing their presence. Once per hour, the command unit is supposed to issue a command to every available drone. In this case, the specification would define a

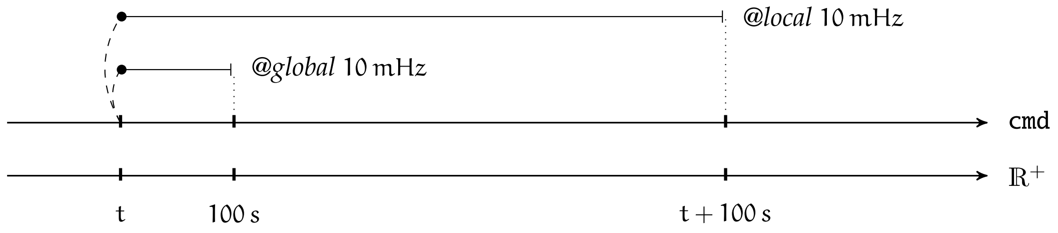


Figure 2.5.: Illustration of Example 2.2. Both variants of the `verify` timelines are created when the monitor receives a command, but they differ in the semantics of their evaluation and termination. The local period is relative to its own timeline whereas the global period refers to the real time axis.

new timeline for each drone. These timelines then tick once per hour *relative to the global-periodic clock*. Hence, this scenario requires the converse semantics. △

The observation from the example is that neither semantics is fundamentally correct. As a result, a timeline needs to be able to refer to either the local timeline induced by its own point of spawn, or the global timeline. Note that this problem only arises for periodic timelines because the local and global timelines are shifted. On the contrary, ticks of event-based timelines are universal, because they are based on the reception of events. Hence, the local and global timeline collapse. In practice, references to the local timeline are significantly more common and this is also the more intuitive semantics. Hence, Mv: Distraction-Freedom dictates that local-periodicity is the default behavior while references to the global timeline require the dedicated `global` keyword.

Coming back to the recursive definition of timelines, note that a consequence thereof is that timelines can be *nested*, depending on the spawn and close sub-timelines. While in theory, this nesting can go arbitrarily deep, each nesting introduces a new level of complexity since every timeline could refer to all lower nesting layers. Such functionality is practically only necessary to a certain degree. Hence, even though it impacts expressiveness of the language, RTLola only provides syntax for the following forms of nesting:

Nested Timelines

- Spawn clauses may refer to the global timeline only.
- Evaluation clauses may refer to the global timeline (`global` keyword) or the local timeline induced by the spawn (default).
- Close clauses may refer to the global timeline (`global` keyword) or the local timeline induced by the spawn (default).

In summary, a timeline consists of the following components:

Clock The clock is either event-based, global-periodic, or local-periodic, i.e., referring to its local timeline induced by its spawn timeline.

Static Filter The static filter reduces the clock based on either the point in time for periodic timelines or input stream updates for event-based timelines.

Semantic Filter The semantic filter reduces the clock based on a semantic criterion which requires access to the current internal state of the monitor.

Spawn Timeline Determines when a timeline starts. Consists of a clock, a static and a semantic filter.

Close Timeline Determines when a timeline ends. Consists of a clock, a static and a semantic filter.

Note that the static and semantic filter refers to the evaluation clause of a stream declaration.

Example 2.3 (Timelines)**.** Recall the running example specification from Listing 2.4. All input streams follow the event-based clock. Their static filter is their own name, i.e., `oid` ticks only when an event contains `oid`, analogously for all other input streams. As input streams, both their spawn and termination timelines do not exist, and its semantic filter is tautological, indicating that no ticks are filtered out. ➔ Listing 2.4, p. 25

The `velo` stream depends on both `spd` inputs, hence it runs by the event-based clock and its static filter is the conjunction of these streams. In terms of spawn and termination, the stream behaves like an input.

The `ttc` stream is most involved. The evaluation clause dictates its static filter and base clock, both inferred from the expression. Since it accesses `sx, sy, distx`, and `disty`, its timing is tied to these four streams. All of them follow the event-based clock, with static filter `distx` for `sx` and `distx` due to — respectively — its nature as input stream and the type annotation `@distx`. Similarly, the static filter for `sy` and `disty` is `disty`. As a result, `ttc` also follows the event-based clock and its static filter is the conjunction of both distance inputs. Its semantic filter is explicit part of the specification, i.e., the expression `collision_possible`.

What remains is the spawn and close timeline. The former has an explicit type annotation, hence it follows the event-based clock with static filter `oid` and does not have a semantic filter. Similarly, the latter follows the event-based clock with static filter `obj_lost` $\wedge$ `oid`. Its semantic filter is the expression `oid = id`.

△

This example illustrates how to obtain each constituent of a stream's timeline. The next section formally defines a timeline and how to derive it for an RTLola specification.

2.3.1. RTLola Types

Formally, an RTLola type is a four-tuple $\tau = (\nu, \lambda, \lambda^{\Rsh}, \lambda^{\Lsh})$ with a value type ν, an evaluation timeline λ, spawn timeline $\lambda^{\Rsh}$, and close timeline $\lambda^{\Lsh}$. Each timeline is a pair of a semantic filter ζ and a pacing ψ. Here, ψ summarizes the clock including its point of reference,

Table 2.6.: Overview over the RTLola type hierarchy and meta variables.

τ (Type)					
ν (Value)	λ (Timeline)			$\lambda^{↱}$ (Spawn)	$\lambda^{↳}$ (Close)
ν	ζ (Filter)	ψ (Pacing; Disjunctive)		$\lambda^{↱}$	$\lambda^{↳}$
ν	ζ	π (Period)	ε (Ev-Based)	$\lambda^{↱}$	$\lambda^{↳}$
`:Int`	`when x > 3`	`@5Hz`	`@x`	$\lambda^{↱}$	$\lambda^{↳}$

and static filter. Table 2.6 depicts all constituents in their type hierarchy as well as their names, meta variables, and an example syntax.

2.3.2. Lattices

Lattices are a convenient tool to describe type systems. There are multiple notions of lattices, however, this thesis only deals with bounded meet-semilattices as defined below. Any further mention of lattices refers to this definition.

Definition 2.4 (Bounded Meet-Semilattice)

Def. Meet-Semilattice

Def. Bounded Meet-Semilattice

A partially ordered set $(L, \sqsubseteq)$ is a *meet-semilattice* if any two elements $x, y \in L$ have a unique greatest lower bound with respect to $\sqsubseteq$. This element $z = x \sqcap y$ is the meet of x and y. Further, the meet-semilattice is *bounded*, if it has a greatest ($\top \in L$) and least ($\bot \in L$) element in L, i.e., for any $x \in L$: $\bot \sqsubseteq x \sqsubseteq \top$. These elements are the top and bottom elements of the lattice.

Remark 2.3 (Notation and Meta Variables). This chapter makes heavy use of meta variables for its notation. Hence, symbols for different type components as indicated in Table 2.6 implicitly qualify a type, e.g., τ always refers to a full type and any ν is implicitly a member VT, which is the set of value types. Moreover, static filters will continue to be separated into periodic types with symbol π and event-based types with symbol ε. Destructing a timeline into a pair renders it a spawn or close timeline, where the destruction in four components renders it a full timeline. Lastly, when working with lattices, the subscript of different comparisons is dropped if unambiguously possible, i.e., $\nu_1 \sqsubseteq \nu_2 \iff \nu_1 \sqsubseteq_{VT} \nu_2$

2.3.3. Atomic Type Lattices

The RTLola type lattice consists of multiple sub-lattices, mirroring the structure of a type itself. Hence, each leaf of Table 2.6 has a full type lattice whereas the lattices for timelines and full types are induced by a piece-wise comparison of their respective constituents.

The type lattices follow the intuition that the *least* element is the most restricted or *most concrete* entity. As a result, if a type τ is "less" than τ', i.e., $\tau \sqsubseteq \tau'$, then τ' can be transformed into τ but not vice versa. This process is called *coercion*. For example, an integer that requires 16 bits to be stored can also be fitted into 32 bits without loss of precision, so 16-bit integers are less concrete than 32-bit integers ($Int32 \sqsubseteq Int16$), enabling the coercion. This concept translates seamlessly to timing-related types. If a timeline λ is lower in the lattice, i.e., $\lambda \sqsubseteq \lambda'$, it intuitively produces fewer values[13]. As an example, consider a timeline λ with period $200\,\mu s$ versus λ' with period $100\,\mu s$. The former timeline produces exactly half as many values, thus, λ' can be coerced into λ by dropping every other tick. While this conversion is lossy, it is also possible as opposed to inventing new values.

Coercion

In general, all but the value type lattice are infinite: the period of a timeline, for example, is any natural number of the most fine-grained unit of time. However, the type lattice for any particular specification is finite, as will become evident when considering the following definitions. Hence, for the remainder of this section, let the type lattice be defined for an arbitrary but fixed specification Φ.

Value Type Lattice The value type lattice is generic and similar to type lattices of conventional programming languages. It declares an order on different numeric types, e.g., floating point and (un-)signed numbers; plus a boolean type. This can easily be extended to encompass other data interpretations like enums, fixed-length arrays such as strings, or fixed-arity compound types like tuples and structs.

Definition 2.5 (Value Type Lattice [Sch19a])

The RTLola *value type lattice* is the pair $(VT, \sqsubseteq_{VT})$. Here, the set of value types VT is defined based on a set of types VT^+, from which each element occurs in VT twice, once regular, and once wrapped into an *Option*.

Def. Value Type Lattice

$$VT = VT^+ \cup \{Option(v) \mid v \in VT^+\} \cup \{\top, \bot\}$$

The underlying set is defined as:

$$VT^+ = \{Bool, Int(x), UInt(x), Float(y) \mid x \in \{8, 16, 32, 64\}, y \in \{32, 64\}\}$$

Further, the order $\sqsubseteq_{VT}$ is the minimal order encompassing the following inequalities:

$$\begin{aligned}
Int(x_1) \sqsubseteq_{VT} Int(x_2) &\iff x_1 \geqslant x_2 \\
UInt(x_1) \sqsubseteq_{VT} UInt(x_2) &\iff x_1 \geqslant x_2 \\
Float(y_1) \sqsubseteq_{VT} Float(y_2) &\iff y_1 \geqslant y_2 \\
Option(v) \sqsubseteq_{VT} Option(v') &\iff v \sqsubseteq_{VT} v'
\end{aligned}$$

[13]Caution: "fewer values" in a timing lattice correspond to "more bits" in a value type lattice.

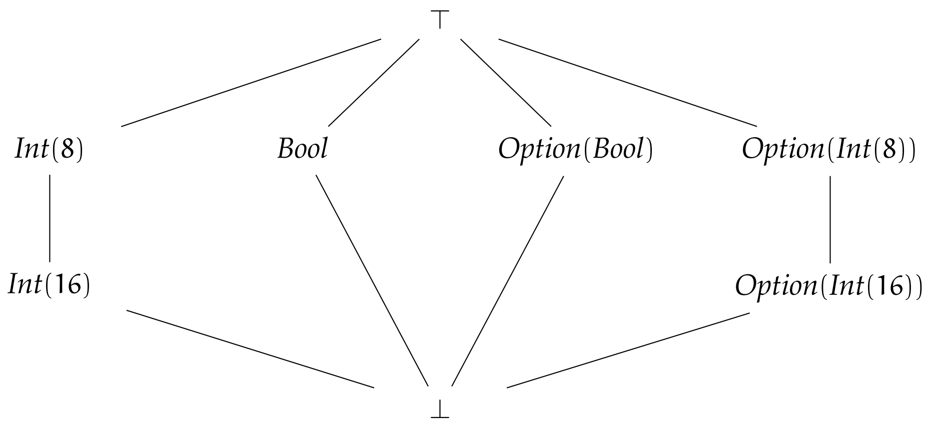


Figure 2.7.: Illustration of a value type lattice with a boolean and two integer base types.

$$\begin{aligned} \nu &\sqsubseteq_{VT} \top && \text{for } \nu \in VT \\ \bot &\sqsubseteq_{VT} \nu && \text{for } \nu \in VT \end{aligned}$$

The last two lines declare $\top$ and $\bot$ as the top and bottom elements of the lattice.

Intuitively, for any two value types $\nu_1 \sqsubseteq \nu_2$, both types come from the same "family" of types and — if applicable — representing ν_1 in memory requires more bits than ν_2. Moreover, $\top$ represents an unrestricted type, $\bot$ the error type, and optional values mirror the lattice of non-optional values.

Example 2.6 (Reduced Value Type Lattice)**.** Consider a value type lattice identical to the RTLola value type lattice except with $VT^+ = \{Bool, Int(8), Int(16)\}$. Example 2.6 illustrates the lattice where node further up in the figure are greater with respect to $\sqsubseteq_{VT}$.

$\triangle$

Pacing Type Lattice The pacing type lattice encompasses both static filters and clocks. It is again split in two: event-based types and periodic types. The former are positive boolean formulas over input stream literals. Hence, the relation between different event-base types is based on implication. Periodic types on the other hand relate to the real time axis. Here, two types are only comparable if they are integer multiples of each other where lower periods are faster and thus higher up in the lattice.

Definition 2.7 (Pacing Type Lattice)

Def. Event-Based Type Lattice

The *event-based type lattice* is based on the set of input streams $\mathcal{S}^{\downarrow}$ occurring in Φ.

$$ET = \mathbb{B}^{+}_{\mathcal{S}^{\downarrow}}$$

$$\varepsilon_1 \sqsubseteq_{ET} \varepsilon_2 \iff (\varepsilon_2 \implies \varepsilon_1)$$

For the *periodic type lattice*, let $\Pi \subseteq \mathbb{N}$ be the set of periods occurring as type annotations in Φ paired with an indicator for the point of reference.

Def. Periodic Type Lattice

$$PeT = \left\{ k \cdot \mathrm{gcd}(\Pi) \mid k \in \left[1, \frac{\max(\Pi)}{\mathrm{gcd}(\Pi)}\right] \subseteq \mathbb{N} \right\} \times \{global, local\}$$

$$\pi_1 \sqsubseteq_{PeT} \pi_2 \iff \pi_1.2 = \pi_2.2 \wedge \exists k \in \mathbb{N}.\, k \cdot \pi_1.1 = \pi_2.1$$

Finally, the *pacing type lattice* $(PT, \sqsubseteq_{PT})$ is the amalgamation of the two disjoint sublattices for event-based types and periodic types.

Def. Pacing Type Lattice

$$PT = ET \,\dot{\cup}\, PeT \,\dot{\cup}\, \{Always, Never\}$$

$$\pi_1 \sqsubseteq_{PT} \pi_2 \iff \pi_1 \sqsubseteq_{ET} \pi_2 \vee \pi_1 \sqsubseteq_{PeT} \pi_2$$

$$\forall \pi \in PT : Never \sqsubseteq_{PT} \pi \wedge \pi \sqsubseteq_{PT} Always$$

A result of the last line is that *Never* is the bottom and *Always* is the top element of the lattice.

There are three things to note here. First, the definition deviates from the one for RTLola in 2019 [Sch19a] in three major points: the inclusion of *Always* and *Never* as top and bottom elements of the shared type lattice, the notion of local and global timelines, and the lifting of event-based types from sets of input streams to boolean formulas. Second, note that $Always \in PT$ and $true \in ET$ are subtly different in that *true* and *false* imposes the type to be event-based whereas *Always* can be either event-based or periodic. The same holds for $Never \in PT$ and $false \in ET$. Last, the periodic type is only an upper-bounded meet-semilattice since there is no fix lower bound. This does not affect the periodic type lattice since *Never* lower-bounds the periodic lattice.

Example 2.8 (Pacing Type Lattice)**.** Consider the following specification where frequencies are stated as periods for better readability:

```
input a: Int16
input b: Int16
output x@1s := 3.1415
output y@3s := 2.7182
```

When constructing the type lattice for the specification, the set of inputs is $\mathcal{S}^{\downarrow} = \{a, b\}$, $\mathrm{gcd}(\Pi) = 1$, and the maximum period is $\max(\Pi) = 3$. Hence, $ET = \{a, b, a \vee b, a \wedge b, \mathit{false}\}$ where $a \vee b \equiv \mathit{true}$ and $PeT = \{1, 2, 3\} \times \{global, local\}$. Figure 2.8 depicts the resulting type lattice. △

Semantic Type Lattice The semantic type is the last leaf type and forms the basis for semantic filters in timelines. For this, recall that every semantic type in RTLola is an expression. The conjunctive closure allows for getting the set of expressions obtainable by conjunction starting with a certain base set.

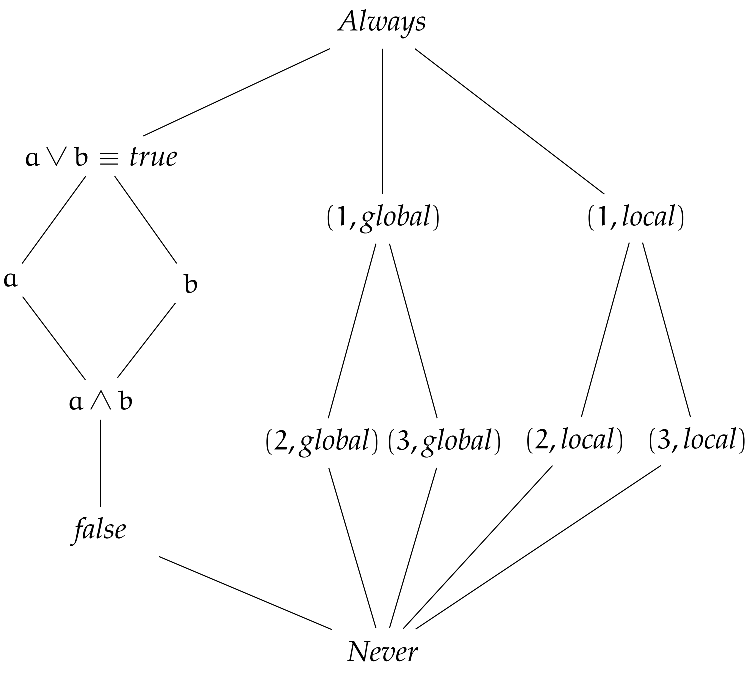


Figure 2.8.: Illustration of a pacing type lattice with two input streams and three possible stream periods.

Definition 2.9 (Conjunctive Closure)

Def. Conjunctive Closure

The *conjunctive closure* of a set of expressions $\mathcal{S}$ yields all expressions that can be composed by conjoining elements of $\mathcal{S}$.

$$\mathsf{closure}(\mathcal{S}) = \bigcup_{i \leqslant |\mathcal{S}|} \left\{ \bigwedge_{j \leqslant i} e_j \mid e_j \in \mathcal{S} \right\}$$

For this to take full effect, recall that on a syntactic level, any ζ is a conjunction with i_ζ conjuncts where $i_\zeta \geqslant 1$, i.e., $\zeta = \bigwedge_{i \leqslant i_\zeta} \zeta_i$.

Definition 2.10 (Semantic Type Lattice)

Def. Semantic Type Lattice

The RTLola *semantic type lattice* is the pair $(ST, \sqsubseteq_{ST})$. Here, the set of semantic types ST is based on the set of expressions occurring in Φ. This set potentially contains *true* and/or *false*, both of which will be removed. Instead, it contains *Always* and *Never* as top- and bottom elements, respectively.

$$ST^b = \left\{ expr(\sigma^\uparrow), filter(\sigma^\uparrow), expr(\sigma^\uparrow.spawn), filter(\sigma^\uparrow.spawn), filter(\sigma^\uparrow.close) \right\}$$

$$ST^+ = \mathsf{closure}\left(\bigcup_{\zeta \in ST^b} \left\{ \zeta_i \mid \zeta = \bigwedge_{i \leqslant i_\zeta} \zeta_i \right\} \right)$$

$$ST = ST^+ \setminus \{\mathit{true}, \mathit{false}\} \cup \{\mathit{Always}, \mathit{Never}\}$$

The comparison of lattice elements is then:

$$\zeta_1 \sqsubseteq_{ST} \zeta_2 \iff \forall m \in [1, i_{\zeta_2}] \colon \exists n \in [1, i_{\zeta_1}] \colon \zeta_{1,n} = \zeta_{2,m}$$
$$\forall \zeta \in ST \colon \mathit{Never} \sqsubseteq_{ST} \zeta \sqsubseteq_{ST} \mathit{Always}$$

Once again, *Always* and *Never* are the top and bottom elements of the lattice, respectively.

Intuitively, the $\sqsubseteq_{ST}$ relation checks for syntactic implication, i.e., if $\zeta_1 \sqsubseteq_{ST} \zeta_2$, then each conjunct of ζ_2 has a syntactically and thus semantically equivalent counterpart in ζ_1. As a result, a semantic type lower in the lattice is active less often than one higher up.

This definition allows for some variation. The semantically optimal definition, for example, would compare the semantics of two expressions, i.e.:

$$\zeta_1 \sqsubseteq^*_{ST} \zeta_2 \iff (\llbracket \zeta_2 \rrbracket \implies \llbracket \zeta_1 \rrbracket)$$

Here, $\llbracket \zeta \rrbracket$ denotes semantic evaluation as will be defined in Definition 2.22. However, checking for semantic equivalence for RTLola expressions is expensive for finite data types and undecidable for infinite ones[14]. This is why Definition 2.10 opts for the syntactic under-approximation, with $\zeta_1 \sqsubseteq_{ST} \zeta_2 \implies \zeta_1 \sqsubseteq^*_{ST} \zeta_2$ but not vice versa.

→ Def. 2.22, p. 64

Example 2.11 (Semantic Type Lattice)**.** Consider the following specification:

```
input a: Bool
input b: Float32
output x eval when a with 1.7320*b
output y eval when a ∧ b > 7.0 with 1.6810*b
output y eval when a ∧ b > 3.0 with 1.4142*b
```

Here, $ST^b = \{a, a \wedge b > 7, a \wedge b > 3\}$ is the set of all semantic types occurring in the specification. The closure yields:

$$ST^+ = \{a, a \wedge b > 7, a \wedge b > 3, b > 7, b > 3, a \wedge b > 3 \wedge b > 7\}$$

Example 2.11 depicts two type lattices. RTLola uses the left one with the order $\sqsubseteq_{ST}$ that checks for syntactic rather than semantic implication. The one on the right uses $\sqsubseteq^*_{ST}$ instead.

△

[14]It can easily be seen that any Diophantine equation can be encoded in RTLola expressions. Checking semantic equivalence can be reduced to solving them, which is undecidable [Mat70]

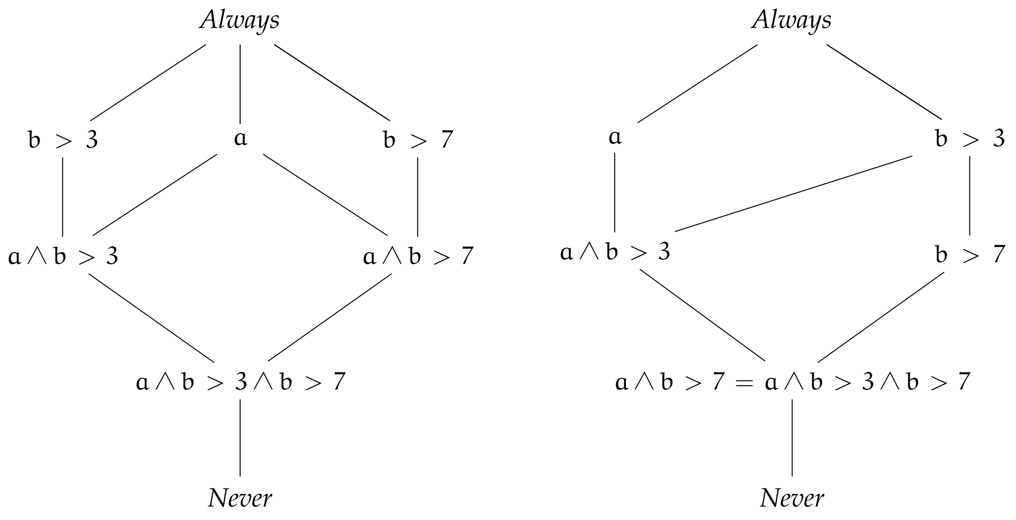


Figure 2.9.: Illustration of a semantic type lattice using the syntactic implication order on the left, and the semantic implication order on the right. Note that $a \wedge b > 7$ syntactically implies a whereas $a \wedge b > 7$ only semantically implies $a \wedge b > 3$.

2.3.4. Timeline Type Lattice

For the most part, this lattice is just the combination of all aforementioned timing-related lattices with no surprises. The exception to this rule is the close sub-lattice. Recall that a timeline that is higher in the type lattice is supposed to "run faster". Evidently, this is the case for the pacing and semantic type lattices. This intuition also translates to the spawn timeline, though less obviously. If a timeline ticks faster and thus more often, the spawn occurs faster. Hence, the timeline is created earlier and thus ticks more often. For the termination timeline, however, the converse is true. If it ticks slowly, the encompassing timeline is closed less quickly, lasts longer and thus ticks more often. Complying with this intuition, in the timeline type lattice, the lattice responsible for the close-component is inverted.

Definition 2.12 (Timeline Type Lattice)

Def. Timeline Type Lattice

The *timeline type lattice* is a pair $(TlT, \sqsubseteq_{TlT})$ with:

$$TlT = (PT \times ST)^3$$

Where the relation $\sqsubseteq_{TlT}$ is defined as follows:

$$\begin{aligned}(\psi_1, \zeta_1, \psi_1^{\Rsh}, \zeta_1^{\Rsh}, \psi_1^{\Lsh}, \zeta_1^{\Lsh}) \sqsubseteq_{TlT} (\psi_2, \zeta_2, \psi_2^{\Rsh}, \zeta_2^{\Rsh}, \psi_2^{\Lsh}, \zeta_2^{\Lsh}) \iff\ & \psi_1 \sqsubseteq_{PT} \psi_2 \wedge \zeta_1 \sqsubseteq_{ST} \zeta_2 \wedge \\ & \psi_1^{\Rsh} \sqsubseteq_{PT} \psi_2^{\Rsh} \wedge \zeta_1^{\Rsh} \sqsubseteq_{ST} \zeta_2^{\Rsh} \wedge \\ & \psi_1^{\Lsh} \sqsupseteq_{PT} \psi_2^{\Lsh} \wedge \zeta_1^{\Lsh} \sqsupseteq_{ST} \zeta_2^{\Lsh}\end{aligned}$$

Its top and bottom elements are respectively:

$$(Always, Always, Always, Always, Never, Never) \quad and$$

$$(Never, Never, Never, Never, Always, Always)$$

2.3.5. RTLola Type Lattice

Finally, the RTLola type lattice is the component-wise conjunction of the value type lattice and the timeline type lattice.

Definition 2.13 (RTLola Type Lattice)

The *RTLola type lattice* is a pair $(RTT, \sqsubseteq_{RTT})$ with:

Def. RTLola Type Lattice

$$RTT = VT \times TlT$$

Where the relation $\sqsubseteq_{RTT}$ is defined as:

$$(\nu_1, \lambda_1) \sqsubseteq_{RTT} (\nu_2, \lambda_2) \iff \nu_1 \sqsubseteq_{VT} \nu_2 \wedge \lambda_1 \sqsubseteq_{RTT} \lambda_2$$

2.3.6. Type Inference

Type inference for RTLola is based on a *type-check relation* $\models$. Given an expression or stream declaration and a type τ, the relation imposes restrictions on τ. This disqualifies a range of types; whatever remains is a set of *candidate types*. While each of them constitutes a valid type assignment, the inference will default to the greatest and thus least restrictive element with respect to the RTLola type lattice. As a result, intuitively, value types take up only as much memory as needed and streams generate as many outputs as possible.

Type-Check Relation

Candidate Type

The type checking procedure utilizes compatible parameter maps of pairs of streams. A map $\alpha^{\sigma \mapsto \sigma'}$ translates parameters of σ' to a set of compatible parameters of σ such that they will be semantically equivalent. This means, during the evaluation of the monitor, if two instances of σ and σ' exist at the same time, $\alpha^{\sigma \mapsto \sigma'}$ indicates which parameters will have the same values. The principle is exemplified after the formal definition. Recall the notation introduced in Section 2.2.5.

→Sec. 2.2.5, p. 41

Definition 2.14 (Compatible Parameters)

A *compatible parameter map* $\alpha^{\sigma \mapsto \sigma'}$ yields for a given parameter of σ' the compatible parameters of σ. Let the spawn expression of both streams be a potentially unary tuple.

Def. Compatible Parameter Map

$$\alpha^{\sigma \mapsto \sigma'}(\sigma'.param[i]) = \\ \{p \mid \exists j \leqslant p^{\sigma} \wedge p = \sigma.param[j] \wedge \sigma.spawn.expr[j] = \sigma'.spawn.expr[i]\}$$

The point behind the compatible parameter maps becomes evident when considering an example.

Example 2.15 (Compatible Parameters)**.** Consider the following specification.

```
input a, b, c: Int16
output x(x1: Int16)
    spawn with a ...
output y(y1: Int16, y2: Int16)
    spawn with (a, c) ..
output z(z1: Int16, z2: Int16, z3: Int16, z4: Int16)
    spawn with (a, b, c, c)
    eval with
        x(z1) // valid
        x(z2) // invalid
        y(z1, z2) // invalid
        y(z2, z1) // invalid
        y(z1, z3) // valid
        y(z1, z4) // valid
```

The specification contains several alternative evaluation expressions for the stream z. The parameter maps of all output streams are:

$$\begin{aligned}
&\alpha^{y\mapsto x}(x_1) = \{y_1\} && \alpha^{x\mapsto z}(z_1) = \{x_1\}\\
&\alpha^{z\mapsto x}(x_1) = \{z_1\} && \alpha^{x\mapsto z}(z_2) = \alpha^{x\mapsto z}(z_3) = \alpha^{x\mapsto z}(z_4) = \emptyset\\
&\alpha^{x\mapsto y}(y_1) = \{x_1\} && \alpha^{y\mapsto z}(z_1) = \{y_1\}\\
&\alpha^{x\mapsto y}(y_1) = \emptyset && \alpha^{y\mapsto z}(z_2) = \emptyset\\
&\alpha^{z\mapsto y}(y_1) = \{z_1\} && \alpha^{y\mapsto z}(z_3) = \alpha^{y\mapsto z}(z_4) = \{y_2\}\\
&\alpha^{z\mapsto y}(y_1) = \{z_3, z_4\}
\end{aligned}$$

Hence, when z accesses y, the first argument has to match the first parameter of the accessee, i.e., y_1, so it has to be a member of $\alpha^{z\mapsto y}(y_1) = \{z_1\}$. Similarly, the second argument has to be drawn from $\alpha^{z\mapsto y}(y_2) = \{z_3, z_4\}$. As a result, the accesses in the last two lines are both valid. △

Scope

The last prerequisite to defining the type check relation concerns parameters and *scopes*. The expressions occurring in the evaluation or close clause of a stream operate in the scope of its enclosing stream. As a result, they may refer to its parameters. These parameters are then effectively constants.

Based on this, the type-check relation is defined as:

Definition 2.16 (Type Inference)

Given a stream declaration S. The *type-check relation* $\tau \models S$ states that τ is a valid type for the stream declared in S. Let *ex* be an expression and σ be a stream. Type τ constitutes a valid type for *ex* in the scope of σ if and only if $\tau \models^{\sigma} ex$. Further, $\tau \models^{\emptyset} ex$ denotes that τ is a valid type for *ex* in an empty scope.

Def. Type-Check Relation

Suppose $T^S = \{\tau \mid \tau \models S\}$ is the set of *candidate types* for S declaring a stream σ. The *inferred type* for σ is then $\max_{\sqsubseteq_{RTT}} T^S$.

Def. Candidate Types

Def. Inferred Type

Finally, let $\lambda \models^{\sigma} \zeta$ denote a valid timeline for the semantic type τ within the context of σ.

Remark 2.4 (Typing a Type)**.** Semantic types are valid RTLola expressions. Hence, naturally, they have a type. While the shape of these values is pre-determined as boolean, their timing is not. For this reason, the value type is omitted, and the timeline type is inferred based on the expression just like for any other expression.

Inference Rules

Inference rules for the RTLola type system can be roughly separated into several groups: constants, operators, stream accesses, and stream declarations. The former two groups are relatively conventional as they are commonly found in programming languages. The latter two groups are specific to RTLola.

Constant Rules For constants, the inference rules are only concerned with the value type. Since constants always exist, their timeline is arbitrary.

The first rule covers integer constants. If the given constant is non-negative, its type defaults to unsigned integers rather than a signed one. The width of the resulting type is the minimal number of bits required to represent the respective value. Here, *range*(T) is the set of all values representable with type T.

$$\frac{T = \begin{cases} \mathit{UInt} & \text{if } c >= 0 \\ \mathit{Int} & \text{otherwise} \end{cases} \qquad x = \operatorname*{arg\,min}_{p \in \{8,16,32,64\}} \; c \in \mathit{range}(Tp) \qquad v = Tx}{(v, \lambda) \models^{\sigma} c}\ \textsc{ConstInt}$$

The rule for floats is nigh identical, deviating mainly in the available bit widths.

$$\frac{c \notin \mathbb{Z} \qquad c \in \mathit{range}(\mathit{Float}64) \qquad x = \operatorname{arg\,min}_{p \in \{32,64\}} c \in \mathit{range}(\mathit{Float}p)}{(\mathit{Float}(x), \lambda) \models^{\sigma} c}\ \textsc{ConstF}$$

The next two rules cover boolean constants and parameters. To understand why the latter are constants, recall that parameters occur in expressions. These expressions are only evaluated when the respective stream is instantiated. At this point in time, the exact value of the parameter is determined, rendering them an effective constant.

The inference rule for boolean values requires that the constant is either *true* or *false*. A parameter needs to occur in the list of parameters of the respective stream and has the annotated value type. This type annotation is mandatory in a specification.

$$\frac{c \in \{\mathit{true}, \mathit{false}\}}{(\mathit{Bool}, \lambda) \models^{\sigma} c}\ \textsc{ConstB} \qquad \frac{\sigma.\mathit{param}\,[i] = p \qquad \mathcal{T}_i^{\sigma} = \nu}{(\nu, \lambda) \models^{\sigma} p}\ \textsc{Param}$$

Operator Rules The rules for unary and binary operators, as well as n-ary functions, all follow the same pattern. First, infer the type of sub-expressions recursively. Second, apply the value type rules of the respective operator or function. Third, the timeline of the expression is the meet of the timelines of the sub-expressions.

$$\frac{(\nu', \lambda') \models^{\sigma} ex \qquad \circ\colon \mathcal{T}' \to \mathcal{T} \qquad \mathcal{T}' \sqsubseteq \nu' \qquad \mathcal{T} \sqsubseteq \nu \qquad \lambda \sqsubseteq \lambda'}{(\nu, \lambda) \models^{\sigma} \circ ex}\ \textsc{Unary}$$

$$\frac{\begin{array}{c}(\nu_1, \lambda_1) \models^{\sigma} ex_1 \qquad (\nu_2, \lambda_2) \models^{\sigma} ex_2 \\ \circ\colon \mathcal{T}_1 \times \mathcal{T}_2 \to \mathcal{T} \qquad \mathcal{T}_1 \sqsubseteq \nu_1 \qquad \mathcal{T}_2 \sqsubseteq \nu_2 \qquad \mathcal{T} \sqsubseteq \nu \qquad \lambda \sqsubseteq \lambda_1 \sqcap \lambda_2\end{array}}{(\nu, \lambda) \models^{\sigma} ex_1 \circ ex_2}\ \textsc{Binary}$$

$$\frac{\begin{array}{c}\bigwedge_{i \leqslant n} (\nu_i, \lambda_i) \models^{\sigma} ex_i \\ f\colon \bigtimes_{i \leqslant n} \mathcal{T}_i \to \mathcal{T} \qquad \bigwedge_{i \leqslant n} \mathcal{T}_i \sqsubseteq \nu_i \qquad \mathcal{T} \sqsubseteq \nu \qquad \lambda \sqsubseteq \bigsqcap_{i \leqslant n} \lambda_i\end{array}}{(\nu, \lambda) \models^{\sigma} f(ex_1, \ldots, ex_n)}\ \textsc{Fn}$$

Default expressions behave similarly, yet with two differences. The type of the subexpression needs to be an optional one. Anything else would render the default expression pointless. Moreover, the resulting value type is the meet of the value type of the subexpression and default value expression.

$$\frac{(\mathit{Option}(\nu_1), \lambda_1) \models^{\sigma} ex_1 \qquad (\nu_2, \lambda_2) \models^{\sigma} ex_2 \qquad \nu \sqsubseteq \nu_1 \sqcap \nu_2 \qquad \lambda \sqsubseteq \lambda_1 \sqcap \lambda_2}{(\nu, \lambda) \models^{\sigma} ex_1.\mathit{defaults}(\mathit{to} : ex_2)}\ \textsc{Dft}$$

Stream Access Rules Stream accesses consist of two components: the selection of a particular stream instance and the selection of accessed values. The instance selection is a three-step process. First, the name of the stream allows for accessing the inferred type of the stream. Then, the access contains expressions constituting parameters, which selects a specific instance of the stream. Lastly, the access can be either synchronous or asynchronous.

The synchronous version intuitively requires compatibility of the timeline of the accessing and accessed stream instance. This translates to the imposition of a temporal and a semantic constraint. Temporally, the timeline of the accessing stream instance needs to be "slower" than the accessed one. This in particular entails that the creation of the accessing instance guarantees the existence of the accessed one. However, the access itself needs to guarantee that it targets a *matching* instance. To this end, the access must adhere to the compatible parameter map defined in Definition 2.14. This guarantees that the access will always succeed, hence the return value is non-optional.

The following rule imposes the respective constraints, starting by accessing the inferred type of the accessed stream. The rule then checks whether the numbers of parameters in the access and declaration of the target agree, followed by checking for compatibility of parameters. Lastly, the timeline of the accessing stream needs to be more concrete than the one of the accessed stream. Here, $\zeta[\alpha]$ replaces occurrences of parameters of σ by a counter-part according to α. Note that there can be several potential replacements. While they differ syntactically, they are semantically equal. However, since $\sqsubseteq_{ST}$ checks for syntactic equality, all options need to be considered. It is sufficient if one of them satisfies the rule.

$$\dfrac{\begin{array}{c}\tau' \models \sigma' \qquad n = p^{\sigma'} \\ \bigwedge_{i \leqslant n} ex_i \in \alpha^{\sigma \mapsto \sigma'}(i) \qquad (\nu, (\psi, \zeta[\alpha], (\psi^{↱}, \zeta^{↱}[\alpha]), (\psi^{↳}, \zeta^{↳}[\alpha]))) \sqsubseteq \tau'\end{array}}{(\nu, (\psi, \zeta, (\psi^{↱}, \zeta^{↱}), (\psi^{↳}, \zeta^{↳}))) \models^{\sigma} \sigma'(ex_1, \ldots, ex_n).\mathit{sync}}\ \textsc{Sync}$$

The asynchronous access imposes an almost entirely different set of constraints. It also starts by accessing the inferred type of the accessed stream. However, the timeline of the accessed stream is irrelevant for the asynchronous access, as its purpose is to decouple the timelines of accessor and accessee. Furthermore, the value type of the access coincides with the value type of the accessed stream, wrapped in an optional. This represents the fallible nature of the asynchronous access. The fourth premise ensures that the number of parameters fits. Apart from this constraint, the asynchronous rule is more liberal regarding the parameters than the synchronous one. The synchronous access implicitly required ex_i to be parameters, whereas in the asynchronous case they can be arbitrary expressions. This requires their types to be inferred by the fifth premise. The next one assures that the value type of each parameter matches its declared type. Finally, the timeline of the access is the meet of all its parameter expressions.

$$\dfrac{\begin{array}{c}(\nu', \_) \models \sigma' \qquad \nu = \mathit{Option}(\nu'') \qquad \nu'' \sqsubseteq \nu' \\ n = p^{\sigma'} \qquad \bigwedge_{i \leqslant n} (\nu_i, \lambda_i) \models^{\sigma} \mathit{ex}_i \qquad \bigwedge_{i \leqslant n} \mathcal{T}_i^{\sigma'} \sqsubseteq \nu_i \qquad \lambda \sqsubseteq \bigsqcap_{i \leqslant n} \lambda_j\end{array}}{(\nu, \lambda) \models^{\sigma} \sigma'(\mathit{ex}_1, \dots, \mathit{ex}_n).\mathit{async}}\ \textsc{Async}$$

The next step after selecting a stream instance is the specification of which values are accessed. The most basic form is the 0-offset, in which case the most recent value of a stream is accessed. Here, the type of the access merely mirrors the type of the accessed instance. When changing the offset to a strictly negative value, the access becomes fallible, hence the access has an optional type. In this case, the chain starting with an asynchronous access followed by a negative offset is of particular interest. Since both render the returned value optional, this results in a type such as $\mathit{Option}(\mathit{Option}(\mathcal{T}))$ rather than the expected $\mathit{Option}(\mathcal{T})$. For this reason the rule explicitly deflates chains of *Option* types.

$$\dfrac{\tau' \models^{\sigma} \mathit{ex} \qquad \tau \sqsubseteq \tau'}{\tau \models^{\sigma} \mathit{ex}.\mathit{offset}(\mathit{by}: 0)}\ \textsc{Ofs}(0) \qquad \dfrac{\begin{array}{c} n \in \mathbb{N} \qquad (\nu', \lambda') \models^{\sigma} \mathit{ex} \qquad \lambda \sqsubseteq \lambda' \\ \nu = \begin{cases} \nu' & \text{if } \nu' = \mathit{Option}(\_) \\ \mathit{Option}(\nu') & \text{otherwise} \end{cases}\end{array}}{(\nu, \lambda) \models^{\sigma} \mathit{ex}.\mathit{offset}(\mathit{by}: -n)}\ \textsc{Ofs}(\text{-}n)$$

While offsets only access a single value, aggregations access a whole sequence of values, aggregates them, and yields the result. To this end, the aggregation contains a time interval and an aggregation function. The rule ensures that the clock of the access is periodic where the period is a divisor of the duration of the aggregation. This and the requirement that the aggregation is a list homomorphism [Mee86] allows for a memory-efficient aggregation [Li+05; Sch19a]. The value type of the expression corresponds to the resulting type of the aggregation. In case of an exact aggregation, the type is wrapped in an *Option* since the aggregation only starts producing values after δ has passed.

$$\dfrac{(\nu', \_) \models^{\sigma} \mathit{ex} \qquad \psi = \pi \qquad \exists k \in \mathbb{N}.\, \delta = k\pi \qquad \gamma\colon \nu'^{*} \to \nu}{(\nu, (\psi, \zeta, \lambda^{\Rsh}, \lambda^{\text{↳}})) \models^{\sigma} \mathit{ex}.\mathit{aggr}(\mathit{over}: \delta, \mathit{using}: \gamma)}\ \textsc{Aggr}$$

$$\dfrac{(\nu', \_) \models^{\sigma} \mathit{ex} \qquad \exists k \in \mathbb{N}.\, \delta = k\pi \qquad \gamma\colon \nu'^{*} \to \nu'' \qquad \nu = \mathit{Option}(\nu'')}{(\nu, (\pi, \zeta, \lambda^{\Rsh}, \lambda^{\text{↳}})) \models^{\sigma} \mathit{ex}.\mathit{aggr}(\mathit{over\_exactly}: \delta, \mathit{using}: \gamma)}\ \textsc{AggrEx}$$

Stream Declaration Rules There are two kinds of stream declarations, inputs and outputs. For input streams, the declaration only provides the name of the stream and the value type. The remaining information is pre-determined: the pacing type is event-based and contains only the input stream itself, the timeline is unfiltered, hence $\zeta = \mathit{Always}$.

$$\dfrac{\nu = T \qquad \psi = \sigma^{\downarrow} \qquad \zeta = \mathit{Always}}{(\nu, (\psi, \zeta)) \models \mathit{input}\ \sigma^{\downarrow} : T}\ \textsc{Input}$$

Output streams on the other hand are highly variable. However, most of the type information is optional to comply with Mv: Distraction-Freedom. These parts are marked with a $*$-subscript. If they are not present in the specification, the type inference substitutes default values instead, i.e., $\top$ for the value type and *Always* for pacing types and semantic types except in the close clause, where it is *Never* instead. Note that these annotations include the reference mark to the global timeline.

The following type rule consists of multiple lines of premises.

$$\dfrac{\begin{array}{c} ((\nu^1_{spw}, \dots, \nu^n_{spw}), \lambda_{spw}) \models^{\emptyset} ex_{spw} \qquad \bigwedge_{i \leqslant n} \mathcal{T}_i \sqsubseteq \nu^n_{spw} \\ \lambda_{sf} \models \zeta^{\Rsh}_{*} \qquad \lambda^{\Rsh} \sqsubseteq (\psi^{\Rsh}_{*}, \zeta^{\Rsh}_{*}) \sqcap \lambda_{spw} \sqcap \lambda_{sf} \\ (\nu_{ex}, \lambda_{ex}) \models^{\sigma} ex \qquad \nu \sqsubseteq \nu_* \sqcap \nu_{ex} \qquad (\psi, \zeta, \lambda^{\Rsh}, \lambda^{\Lsh}) \sqsubseteq \lambda_{ex} \\ \lambda_{fil} \models^{\sigma} \zeta_* \qquad (\psi, \zeta, \lambda^{\Rsh}, \lambda^{\Lsh}) \sqsubseteq \lambda_{fil} \qquad \zeta \sqsubseteq \zeta_* \\ \lambda^{\Lsh} = (\psi^{\Lsh}_{*}, \zeta^{\Lsh}_{*}) \end{array}}{\begin{array}{ll} (\nu, (\psi, \zeta, \lambda^{\Rsh}, \lambda^{\Lsh})) \models & \mathit{output}\ \sigma(p_1 : \mathcal{T}_1, \dots, p_n : \mathcal{T}_n) : \nu_* \\ & \quad \mathit{spawn}\ @\ \psi^{\Rsh}_{*}\ \mathit{when}\ \zeta^{\Rsh}_{*}\ \mathit{with}\ ex_{spw} \\ & \quad \mathit{eval}\ @\ \psi_{*} \mathit{when}\ \zeta_*\ \mathit{with}\ ex \\ & \quad \mathit{close}\ @\ \psi^{\Lsh}_{*}\ \mathit{when}\ \zeta^{\Lsh}_{*} \end{array}}\ \textsc{Output}$$

The first line corresponds to the parameters of the stream. It first infers the type of the spawn expression, which has to yield a tuple with an arity matching the number of parameters. The value type of each element needs to be coercible into the annotated parameter type.

The second line imposes constraints on the spawn timeline. For this, it first infers the timeline of the delay filter. Then, it asserts that the spawn timeline is more concrete than the annotated timeline, the timeline of the spawn expression, and the timeline of the spawn filter combined. This ensures that the type inference respects the type annotations in the specification and whenever the spawn timeline ticks, both the spawn expression and the spawn filter tick as well. As a result, the creation of an instance can be stopped courtesy of the filter, and — if it is not — the spawn expression provides parameters.

The next line is concerned with the stream expression. The value type depends on the expression plus the annotated value type. Moreover, the timeline of the expression must be compatible with the timeline of the stream.

The last line restricts the close timeline and filter of the stream. It first infers the timeline of the filter expression and ensures that it ticks whenever the timeline of the stream itself ticks. Lastly, the filter and close timelines are more concrete than their annotated counterparts.

Conclusion

This concludes the type inference rules for RTLola. While their complexity seems deterring, specifiers do not need to understand them on this level of detail. Most of the time, all they need to know is the intuition behind timelines such that they can override default semantics whenever desired. Apart from these cases, the example specifications in this thesis already illustrate that manual type annotations are rarely necessary.

2.4. Semantics of Monitors

In quintessence, the monitor is a component that reacts upon inputs and delivers a verdict that judges whether a specification was violated. There are several concepts for achieving this goal. The most simple kind are *rule-based monitors*. Their judgment is based solely on the current input, so the monitor realizes a function

Rule-Based Monitors

$$\mathcal{M}^r\colon \left\langle \mathcal{T}_i^{\downarrow} \right\rangle_{i \leqslant n^{\downarrow}} \to \left\langle \mathcal{T}_j^{\uparrow} \right\rangle_{i \leqslant n^{\uparrow}}$$

Here, a single event for the monitor consists of $n^{\downarrow}$ input values. This corresponds to the number of input streams in RTLola, so $n^{\downarrow} = \left|\mathcal{S}^{\downarrow}\right|$. Similarly, the verdict consists of an $n^{\uparrow}$-tuple of values. During an ongoing execution in the stream-based setting, the monitor is called repeatedly.

This model of a monitor is limited since it only has access to a snapshot of the execution, prohibiting it from evaluating temporal properties. Hence, stream-based monitors carry and modify an internal state, so they realize the following function:

$$\mathcal{M}\colon \left\langle \mathcal{T}_i^{\downarrow} \right\rangle_{i \leqslant n^{\downarrow}} \times \Sigma^{\mathcal{M}} \to \left\langle \mathcal{T}_j^{\uparrow} \right\rangle_{i \leqslant n^{\uparrow}} \times \Sigma^{\mathcal{M}}$$

For a sequence of inputs $\langle ev_i \rangle_{i \leqslant k}$, a monitor $M \in \mathcal{M}$ computes its verdict iteratively. Suppose the initial state is Σ_0^M, so the initial verdict is v_1 for $(v_1, \Sigma_0^M) = M(ev_1, \Sigma_0^M)$ and each subsequent verdict is v_{i+1} for $(v_{i+1}, \Sigma_{i+1}^M) = M(ev_{i+1}, \Sigma_i^M)$. An equivalent, state-less formulation requires the entire sequence of input values so far as input:

$$\mathcal{M}\colon \left(\left\langle \mathcal{T}_i^{\downarrow} \right\rangle_{i \leqslant n^{\downarrow}}\right)^k \to \left\langle \mathcal{T}_j^{\uparrow} \right\rangle_{i \leqslant n^{\uparrow}}$$

The same result can be achieved when the monitor carries every input in its internal state. However, without aggregation and filtering, the internal state grows at least linearly this way. For CPS, this is not an option since the number of inputs the monitor receives while it is deployed is unbounded and the system has to cope with strictly limited memory. This would also partially contradict RTLola's MII: RESOURCE-AWARENESS. In essence, for an RTLola monitor M, the size of its state Σ^M is always statically determined.

While this formulation allows for resolving temporal dependencies, it lacks two critical components: asynchrony and a relation to real time. Asynchrony mainly models that constituents of an event might be delayed or lost in transmission before reaching the monitor. This property manifests as gaps in the input and output tuple, so the monitor allows for receiving — and is allowed to produce — undefined values:

$$\mathcal{M}\colon \left\langle \mathcal{T}_i^{\downarrow} \dot\cup \{\perp\} \right\rangle_{i \leqslant n^{\downarrow}} \times \Sigma^{\mathcal{M}} \to \left\langle \mathcal{T}_j^{\uparrow} \dot\cup \{\perp\} \right\rangle_{i \leqslant n^{\uparrow}} \times \Sigma^{\mathcal{M}}$$

Lastly, the monitor requires a notion of *real time*. For this, first, each event has a timestamp drawn from $\mathbb{R}^+$ indicating when the event took place. Second, the monitor

Real Time

needs a way to react upon the real time without being triggered by the system. To this end, rather than only issuing a verdict and updating the state, the monitor also produces a relative timestamp indicating when its verdict will change given no further event occurs until then. As a result, the monitor is a piece-wise constant signal that changes when it either receives an event or a relative timestamp passed. This leads to the following definition[15].

Definition 2.17 (Asynchronous State-Based Real-Time Monitor Function)

Def. Asynchronous State-Based Real-Time Monitor

An *asynchronous state-based real-time monitor* function with a state drawn from $\Sigma^{\mathcal{M}}$, $n^{\downarrow}$ inputs of types $\left\langle \mathcal{T}_i^{\downarrow} \right\rangle_{i \leqslant n^{\downarrow}}$ and $n^{\uparrow}$ outputs of types $\left\langle \mathcal{T}_j^{\uparrow} \right\rangle_{j \leqslant n^{\uparrow}}$ is defined as:

$$\mathcal{M}\colon \left\langle \mathcal{T}_i^{\downarrow} \dot{\cup} \{\bot\} \right\rangle_{i \leqslant n^{\downarrow}} \times \mathbb{R}^+ \times \Sigma^{\mathcal{M}} \to \left\langle \mathcal{T}_j^{\uparrow} \dot{\cup} \{\bot\} \right\rangle_{i \leqslant n^{\uparrow}} \times \Sigma^{\mathcal{M}} \times \mathbb{R}^+$$

Note that in the definition, the timestamp of the last input event denotes a notion of "now" since events can be empty.

Example 2.18 (Persistence of verdicts)**.** Suppose a monitor M is in state Σ^M and receives input *ev* at time t, for which it produces $M(ev, t, \Sigma^M) = (v, \Sigma^{M'}, t')$. The monitor may not change its verdict for the next t' time units, so for all empty events $()$ and $\delta < t'$ the following holds:

$$M((), t + \delta, \Sigma^{M'}) = (v, \Sigma^{M'}, t' - \delta)$$

$\triangle$

Remark 2.5 (Non-Monotonic Time)**.** The definition of monitors does not require time to increase monotonically. As a result, events can reach the monitor *out-of-order* for example owing to jitter or re-transmissions. While this is an interesting subject in and of itself, this thesis disregards this problem. Hence, for the remainder of the thesis, assume the monitor automatically rejects an input if the respective timestamp is less than one received previously.

2.4.1. Semantics of RTLola

The semantics of RTLola are defined based on an update function for the state and a slicing function, extracting the verdict from the current state. Both of them hinge on the shape of the monitor state. Intuitively, the state is a collection of stream instances, where each instance is a sequence of values. These values were received by the system for input streams and computed by the monitor for output streams. While for practical purposes,

[15]Note that the definition is adapted from my earlier work [Sch19a] to accurately reflect the state-based and real-time property of the monitor.

it is imperative that the state remains within a constant memory bound, the semantics gloss over this, memorizing every piece of information received and computed over different calls to the monitor. This challenge can be overcome by summarizing relevant and evicting outdated information, as presented in Section 2.7.6 for an interpreter, Section 3.1.4 for a hardware monitor, and Section 3.2.3 for a software monitor.

→Sec. 2.7.6, p. 88
→Sec. 3.1.4, p. 111
→Sec. 3.2.3, p. 133

Definition 2.19 (RTLola Monitor State)

An RTLola monitor $M = M_\Phi$ for a specification Φ has an *internal state* $\Sigma = \Sigma^{M_\Phi}$. The state is a timestamp of its last update plus a collection of *stream instances*. A stream instance $\iota \in SI$ consists of a stream identifier, a creation time, an unbounded sequence of their timestamped values and a tuple of aggregation targets. These five constituents — assuming the implicit destruction of the stream identifier — can be accessed via $\iota.\mathit{name}$, $\iota.\mathit{params}$, $\iota.\mathit{time}$, $\iota.\mathit{vals}$, and $\iota.\mathit{aggr}$, respectively. Hence:

Def. Monitor State
Def. Stream Instance

$$\Sigma \in \mathbb{R}^+ \times 2^{SI} \quad \text{with} \quad SI \in \bigcup_{\sigma \in \mathcal{S}} SId(\sigma) \times (\mathcal{T}^\sigma \times \mathbb{R}^+)^* \times \langle SId(\sigma') \rangle_{\sigma' \in \text{aggr}(\sigma)}$$

Let $\iota \in \Sigma \iff \iota \in \Sigma.2$ and let $\Sigma.\mathit{time} = \Sigma.2$.

Note that $SId(\sigma)$ contains the first three components, i.e., the name, parameters, and time of creation.

The update of the monitor state is a multistep process. Recall that a result of a call to the monitor is three-fold: the updated state, a verdict, and a relative time-offset, called a *deadline*. The deadline states when the verdict of the monitor changes provided it does not receive new events. Hence, when a monitor is called sparsely, the verdict might have changed multiple times between calls. Each of these changes is accompanied by a mutation of the monitor state. Hence, the very first step of the update applies all changes triggered by deadlines that have passed in the interim. After these steps, the monitor finally updates its state with respect to the current event.

Deadline

This process needs information regarding the next deadline of a monitor state.

Definition 2.20 (Next Deadline)

The function nextdl produces an absolute point in time when the *next deadline* is due for a given monitor state Σ as follows:

Def. Next Deadline

$$\text{nextdl}(\psi, \Sigma) = \begin{cases} k\pi & \text{if } \psi = (\pi, \mathit{global}) \wedge k \in \mathbb{N} \wedge k\pi \geqslant \Sigma.\mathit{time} \\ \iota.\mathit{time} + \pi & \text{if } \psi = (\pi, \mathit{local}) \wedge |\iota.\mathit{vals}| = 0 \\ \iota.\mathit{vals}.\text{last}.2 + \pi & \text{if } \psi = (\pi, \mathit{local}) \wedge |\iota.\mathit{vals}| > 0 \end{cases}$$

$$\text{pacings}(\iota) = \{\text{pace}(\iota.\mathit{name}), \text{pace}(\iota.\mathit{name}.\mathit{close}), \text{pace}(\iota.\mathit{name}.\mathit{spawn})\}$$

$$\text{nextdl}(\Sigma) = \min_{\iota \in \Sigma} \bigcup_{\psi \in \text{pacings}(\iota)} \text{nextdl}(\psi, \Sigma)$$

Note that $\text{nextdl}(\Sigma) = \Sigma.\mathit{time}$ is possible both by design.

To understand the nextdl function, recall that each stream is either event-based or periodic. In the former case, a stream is only updated reactively, i.e., with respect to an event. Periodic streams, however, are proactive, meaning they need updates based on the real time and thus independent of event. Hence, the function selects all periodic streams. If their pacing refers to the local clock, their respective deadline is π after their last evaluation if any, or π after time 0 if it does not have a value, yet. If the pacing is global, the next deadline is an integer multiple of their period since the monitor time starts at time 0. Hence, nextdl returns ∞, so the minimum is guaranteed to pick another value.

This allows for defining the monitor update process.

Definition 2.21 (Monitor Update Process)

Def. Monitor Update Process

Let Σ be the state of an RTLola monitor and *ev* be an event received at time t. The *update* of the monitor state is defined as

$$\text{update}(\Sigma_j, \mathit{ev}, t) = \text{update}^1(\text{update}^*(\Sigma_j, t), \mathit{ev}, t)$$

Here, update^1 executes an atomic update step and is defined later. The function update^* iteratively updates the monitor state according to missed deadlines. For this, it computes the point in time of the next deadline for a given monitor state. It then triggers an atomic update at the respective point in time with an empty event.

$$\text{update}^*(\Sigma, t) = \begin{cases} \text{update}^*(\text{update}^1(\Sigma, (\bot)^{n^{\downarrow}}, \text{nextdl}(\Sigma)), t) & \text{if } \text{nextdl}(\Sigma) \leqslant t \\ \Sigma & \text{otherwise} \end{cases}$$

Intuitively, the update function handles the scheduling of updates by ordering them according the timestamp of an event. For this, it repeatedly calls update^1 for each deadline that was missed. The update^1 function itself captures the quantitative rather than temporal part of the RTLola semantics. It is mainly concerned with three challenges: managing instances, determining which ones need updates, and evaluating expressions. Let us first inspect the latter.

Definition 2.22 (Expression Evaluation)

Def. Filtered Evaluation

Let Σ be a monitor state, ζ an expression constituting a semantic type, and *ex* be an expression. The *filtered evaluation* $[\![ex]\!]^{\zeta}_{\Sigma}$ evaluates *ex* and returns its value iff the filter ζ holds.

$$[\![ex]\!]^{\zeta}_{\Sigma} = \begin{cases} [\![ex]\!]_{\Sigma} & \text{if } [\![\zeta]\!]_{\Sigma} \\ \bot & \text{otherwise} \end{cases}$$

The (regular) *expression evaluation* $[\![ex]\!]_\Sigma$ performs a case distinction over the syntax of RTLola expressions. For conventional expressions, it behaves ordinarily:

Def. Expression Evaluation

$$[\![c]\!]_\Sigma = c \qquad \text{(Constants)}$$

$$[\![\circ ex]\!]_\Sigma = \circ([\![ex]\!]_\Sigma) \qquad \text{(Unary Operators)}$$

$$[\![ex_1 \circ ex_2]\!]_\Sigma = [\![ex_1]\!]_\Sigma \circ [\![ex_2]\!]_\Sigma$$

$$[\![\mathit{if}\ ex_1\ \mathit{then}\ ex_2\ \mathit{else}\ ex_3]\!]_\Sigma = \begin{cases} [\![ex_2]\!]_\Sigma & \text{if } [\![ex_1]\!]_\Sigma \\ [\![ex_3]\!]_\Sigma & \text{otherwise} \end{cases} \qquad \text{(Conditionals)}$$

$$[\![ex_1.\mathit{defaults}(\mathit{to}: ex_2)]\!]_\Sigma = \begin{cases} v & \text{if } \mathrm{Some}(v) = [\![ex_1]\!]_\Sigma \\ [\![ex_2]\!]_\Sigma & \text{otherwise} \end{cases} \qquad \text{(Defaults)}$$

For RTLola-specific expressions, offsets access the respective stream instance and shift them by the specified value. Recall that the offset is always negative or 0. Aggregations on the other hand access a sequence of values of the target stream instance. The precise values relevant for the aggregation are determined based on their timestamps rather than a discrete offset. They are then aggregated.

Let σ be a stream, Σ be a monitor state and $e_1, \dots, e_{p^\sigma}$ be a sequence of expressions. Then:

$$\iota_? = \begin{cases} \iota & \text{if } \exists \iota \in \Sigma : \iota.\mathit{name} = \sigma \wedge \iota.\mathit{params} = \langle [\![e_i]\!]_\Sigma \rangle_{i \leqslant p^\sigma} \\ \bot & \text{otherwise} \end{cases}$$

Note that ι is uniquely defined since the monitor states cannot contain multiple instance of the same combination of stream name and parameters. Moreover, let $\vec{v}_? = \iota.\mathit{vals}\iota_?$ and $t_? = \iota_?.\mathit{time}$ if $\iota_?$ is a member of Σ or $\bot$ and ∞ otherwise.

$$[\![\sigma(e_1, \dots, e_{p^\sigma}).\mathit{sync}.\mathit{offset}(\mathit{by}: n)]\!]_\Sigma$$

$$= \begin{cases} \mathrm{Some}(\vec{v}_?\,[|\vec{v}_?| + n]) & \text{if } \iota^? \in \Sigma^M \wedge |\vec{v}_?| > n \\ \mathrm{None} & \text{otherwise} \end{cases}$$

$$[\![\sigma(e_1, \dots, e_{p^\sigma}).\mathit{async}.\mathit{offset}(\mathit{by}: n)]\!]_\Sigma$$

$$= [\![\sigma(e_1, \dots, e_{p^\sigma}).\mathit{sync}.\mathit{offset}(\mathit{by}: n)]\!]_\Sigma$$

$$[\![\sigma(e_1, \dots, e_{p^\sigma}).\mathit{async}.\mathit{aggregation}(\mathit{over\_exactly}: \delta, \mathit{using}: \gamma)]\!]_\Sigma$$

$$= \begin{cases} [\![\sigma(e_1, \dots, e_{p^\sigma}).\mathit{async}.\mathit{aggregation}(\mathit{over}: \delta, \mathit{using}: \gamma)]\!]_\Sigma & \text{if } t - \iota_?.\mathit{time} \geqslant \delta \\ \mathrm{None} & \text{otherwise} \end{cases}$$

$$[\![\sigma(e_1,\dots,e_{p^\sigma}).\mathit{async}.\mathit{aggregation}(\mathit{over}:\delta,\mathit{using}:\gamma)]\!]_\Sigma$$

$$= \begin{cases} \text{Some}(\gamma(\vec{v}_?)) & \text{if } \iota_? \in \Sigma \wedge \forall i\colon \vec{v}_?\,[i]\,.2 \geqslant t-\delta \\ \text{Some}(\gamma(\langle \vec{v}_?\,[i]\rangle_{x\leqslant i\leqslant|\vec{v}_?|})) & \text{if } \iota_? \in \Sigma \wedge x = \min\{i \mid \vec{v}_?\,[i]\,.2 \geqslant t-\delta\} \\ \text{None} & \text{otherwise} \end{cases}$$

Note that this is an exhaustive list of the RTLola-specific expression because synchronous aggregations are not allowed. Note further that the semantics does not distinguish between synchronous and asynchronous accesses as they merely contribute to the type system as explained in Section 2.3. →Sec. 2.3, p. 42

The next step for the update function is to determine when a given stream instance requires an update. For this, recall that every stream σ has a pacing, which is either a real-time period $(\pi, f) = \text{pace}(\sigma)$ for $f \in \{\mathit{local}, \mathit{global}\}$ or an event type $\varepsilon = \text{pace}(\sigma)$. The latter is a positive boolean expression where each literal is an input stream name. Such a literal evaluates to true if the stream received a new value with the current event.

Definition 2.23 (Stream Activation)

Def. Active Stream

An instance ι of a stream σ with creation time t_0 is *active* for a given event *ev* at time t if its pacing type is active. A pacing type is active if it either is periodic and t constitutes a deadline or it is event-based and its type evaluates to true for *ev*.

$$\text{active}((\pi,\mathit{global}),t_0,\mathit{ev},t) \iff \exists k \in \mathbb{N}.\, k>0 \wedge k\pi = t \vee ty = \varepsilon$$

$$\text{active}((\pi,\mathit{local}),t_0,\mathit{ev},t) \iff \exists k \in \mathbb{N}.\, k>0 \wedge t_0 + k\pi = t \vee ty = \varepsilon$$

$$\text{active}(\varepsilon,t_0,\mathit{ev},t) \iff \varepsilon\left[\left\langle \sigma_i^{\downarrow} \mapsto \mathit{ev}\,[i] \neq \bot \right\rangle_{i\leqslant n^{\downarrow}}\right]$$

Note that a consequence of this definition is that periodic streams are inactive at the point in time when they are created.

The last remaining challenge is managing stream instances. Initially the monitor creates a set of stream instances which contains an instance for each input stream and each *static output stream*. These are streams with a semantically empty spawn clause.

Static Output Stream

Example 2.24 (Semantically Empty Spawn Clauses)**.** A stream with semantically empty spawn clause is a stream where the pacing type of the spawn clause represents "*Always*", and the spawn condition is a tautology such as in the following stream declaration:

```
output x()
    spawn always when true with ()
    eval @1Hz when true with ...
    close never
```

For these streams, the parameter tuple will always be 0-ary since parameters would depend on computed values, which are inaccessible statically. △

Definition 2.25 (Initial Monitor State)

The *initial monitor state* Σ_0 is defined as a collection of stream instances for all static streams.

Def. Initial Monitor State

$$\Sigma_0 = \Big\{(\sigma, (), 0, ()) \mid \sigma \in \mathcal{S}^{\downarrow} \lor \sigma \in \mathcal{S}^{\uparrow} \land \mathrm{pace}(\sigma.\mathit{spawn}) = \mathit{filter}(\sigma.\mathit{spawn}) = \mathit{Always}\Big\}$$

The atomic update function simultaneously spawns new stream instances when appropriate and updates active instances — even if they were just spawned — unless their filter prohibits the update. Afterwards, it removes all instances due to be closed.

Definition 2.26 (Atomic State Update Function)

The *atomic update function* updates a monitor state Σ_i into an intermediate state Σ^+_{i+1} before closing instances, reducing it to Σ_{i+1}.

Def. Atomic Monitor State Update

$$\mathrm{update}^1(\Sigma_i, \mathit{ev}, t) = \Sigma^+_{i+1} \setminus \{(\sigma, \vec{p}, \_, \_) \mid \mathrm{close}(\sigma, \vec{p}, \Sigma^+_{i+1})\}$$

Here, $\mathrm{close}(\iota, \Sigma)$ determines whether a stream instance is due to be closed.

$$\mathrm{close}(\sigma, \vec{p}, \Sigma^+_{i+1}) = [\![\mathit{expr}(\sigma.\mathit{close})\left[\langle p_i \mapsto \vec{p}\,[i]\rangle_{i \leqslant p^\sigma}\right]]\!]^{\mathit{filter}(\sigma.\mathit{close})[\langle p_i \mapsto \vec{p}\,[i]\rangle_{i \leqslant p^\sigma}]}_{\Sigma^+_{i+1}}$$

Moreover, $\Sigma^+_{i+1} = X^{\downarrow} \,\dot\cup\, X^{\uparrow} \,\dot\cup\, X^{\uparrow}_{+} \,\dot\cup\, X^*$ consists of several sets containing updated, persisted, and/or newly created streams.

First, $X^{\downarrow}$ contains all input stream instances with potentially updated values.

$$X^{\downarrow} = \bigcup_{\sigma^{\downarrow} \in \mathcal{S}^{\downarrow}} \Big\{\Big(\sigma^{\downarrow}_i.\mathit{name}, \sigma^{\downarrow}_i.\mathit{params}, \sigma^{\downarrow}_i.\mathit{time}, \sigma^{\downarrow}_i.\mathit{vals} \circ \mathit{ev}\,[i]\Big)\Big\}$$

Recall that the $\circ$-operator skips the concatenation if the left operand is $\perp$.

Second, $X^{\uparrow}$ denotes all output stream instances which are inactive and thus not updated.

$$X^{\uparrow} = \Big\{\iota \in \Sigma^+_i \mid \iota.\mathit{name} \in \mathcal{S}^{\uparrow} \land \neg\, \mathrm{active}(\mathrm{pace}(\iota.\mathit{name}), \iota.\mathit{time}, \mathit{ev}, t)\Big\}$$

Third, $X_+^{\uparrow}$ is the set of newly updated output streams.

$$X_+^{\uparrow} = \bigcup_{\iota \in \Sigma_{i+1}^+} \Big\{ (\iota.\mathit{name}, \iota.\mathit{params}, \iota.\mathit{time}, \vec{v}) \mid \iota.\mathit{name} \in \mathcal{S}^{\uparrow}$$

$$\wedge\, \mathrm{active}(\mathrm{pace}(\iota.\mathit{name}), \iota.\mathit{time}, \mathit{ev}, t)$$

$$\wedge\, \vec{v} = \iota.\mathit{vals} \circ \left([\![\mathit{expr}(\sigma^{\uparrow}) \left[\langle p_i \mapsto \vec{p}\,[i]\rangle_{i \leqslant p^{\sigma}}\right]]\!]_{\Sigma_{i+1}^+}^{\mathit{filter}(\sigma)[\langle p_i \mapsto \vec{p}[i]\rangle_{i \leqslant p^{\sigma}}]} \right) \Big\}$$

Last, X^* contains newly created instances.

$$X^* = \Big\{ \left(\sigma^{\uparrow}, \vec{p}, t, ()\right) \mid \sigma^{\uparrow} \in \mathcal{S}^{\uparrow} \wedge \vec{p} = [\![\mathit{expr}(\sigma^{\uparrow}.\mathit{spawn})]\!]_{\Sigma_{i+1}^+}^{\mathit{filter}(\sigma^{\uparrow}.\mathit{spawn})}$$

$$\wedge\, \mathrm{active}(\mathrm{pace}(\sigma^{\uparrow}.\mathit{spawn}), t, \mathit{ev}, t) \Big\}$$

The timestamp of the resulting monitor state is t.

This concludes the update logic for monitor states.

Remark 2.6 (Self-Dependent Semantics)**.** The definition of the updated monitor state Σ_{i+1}^+ refers to itself, so it is not immediately evident why this is well-defined. The resolution to this question requires an intricate analysis of the structure of inter-stream dependencies and will follow in Section 2.5.2.

→ Sec. 2.5.2, p. 71

With the update function in tow, only the slicing function is left to define. The purpose of it is slicing the verdict out of the monitor state. According to Definition 2.17, the monitor produces a sequence of output values. Since the verdict of RTLola is based on triggers, the output is a $n^!$-ary tuple of boolean values. Let $\mathcal{S}^! \subseteq \mathcal{S}^{\uparrow}$ be a multi-set of all output streams marked as triggers. Note that $\mathcal{S}^!$ needs to be a multi-set because several triggers may refer to the same output stream.

→ Def. 2.17, p. 62

Definition 2.27 (Slicing)

Def. Verdict Slicing Function

The *verdict slicing function* determines which output streams are marked as triggers and produced a positive output in a given monitor state Σ.

$$\mathrm{slice}(\Sigma) = \left\langle \exists \iota \in \Sigma : \iota.\mathit{name} = \sigma^! \wedge \iota.\mathit{vals}\,[|\iota.\mathit{vals}|] \right\rangle_{\sigma^! \in \mathcal{S}^!}$$

Note that this definition only considers the *latest* value of the output streams. However, receiving a single event can trigger multiple state updates. This can lead to the verdict not reflecting the fact that a trigger was temporarily true and turned false again before the system queried the monitor. Since it is not universally clear whether a query should take outdated alarms into account, there is no "correct" definition. In any case, Definition 2.27 can easily be adapted to reflect the alternative semantics by inspecting the difference between the former and the new monitor state and aggregating all new values with an existential quantification.

Now, with all necessary definitions in place, the update function assembles all sub-results into the final monitor function.

Definition 2.28 (RTLola Monitor Function)

Def. RTLola Monitor Function

The *RTLola monitor function* M updates a monitor state Σ to $\Sigma' = \text{update}(\Sigma, ev, t)$ and produces a timestamp and verdict for an input event ev at time t as follows:

$$M(ev, t, \Sigma) = (\Sigma', \text{nextdl}(\Sigma'), \text{slice}(\Sigma'))$$

Note that neither the updated state, nor the verdict are necessarily uniquely defined. The next section discusses this topic in detail. However, this first requires clarification of the term *evaluation model*.

(Unique) Evaluation Model

Given an RTLola specification Φ. A sequence of monitor states and verdicts is a valid evaluation model if it can be the output of a successive application of the RTLola monitor function. An RTLola specification has a *unique* evaluation model iff each possible input sequence has exactly one evaluation model. These models can, but do not have to be identical.

2.5. Specification Analysis

The dependency graph analysis is a staple for any variant of Lola since its inception. It is a manifestation of one of the prime advantages of the language: the ability to deeply analyze a specification.

In a nutshell, the analysis constructs the eponymous dependency graph. Here, each node represents a stream. Any stream access within the specification results in a labeled edge. The label encapsulates information regarding the kind of dependency.

The analysis of the graph determines the *monitorability* of a specification and generates several artifacts. These artifacts contain insights into the specification and are thus valuable for further steps such as checking compatibility with the target system, an efficient interpretation, and an easier compilation process (cf. Chapter 3). In summary, the analysis determines:

➔Chap. 3, p. 97

Well-Definedness

Well-Definedness The term *well-defined* describes whether there is a unique evaluation model for a given RTLola specification. This is not a given since the semantics of RTLola by itself neither guarantees the existence nor the uniqueness of valid models.

Evaluation Order

Evaluation Order The *evaluation order* is a partial order on streams. For any pair of streams, if $\sigma \prec \sigma'$, then σ depends on σ'. Thus, when computing a model of the specification, the evaluation of σ needs to precede the one for σ'.

Memory Bound

Memory Bounds In RTLola, streams underlie a strict *memory bound*, i.e., a finite extract of the model suffices to compute consecutive outputs.

2.5.1. Dependency Graph

The dependency graph of a specification lays the foundation for further analysis steps. It consists of finite sets of vertices and edges where each vertex represents a stream. Edges of the graph are labeled with information regarding the kind of dependency, i.e., whether an access takes place in the close clause, spawn clause, semantic filter or stream expression. Hence, this information summarizes the relation between the timelines of streams. It also indicates the temporal nature of an access, i.e., it states whether a stream accesses another's values synchronously or asynchronously, whether it accesses a single value or multiple, and which ones. Hence, if a stream σ accesses σ' in its stream expression *and* in the semantic filter of its close clause, the dependency graph will have at least two distinct edges from σ to σ'.

Definition 2.29 (RTLola Dependency Graph)

Def. Dependency Graph

Def. Dependency Labels

The *dependency graph* $\mathcal{D}_\Phi = (V, E)$ of a desugared RTLola specification Φ is a labeled directed multigraph. The vertices are streams, i.e., $\mathcal{S} = V$ and each edge represents a dependency with $E \subseteq V \times L_\mathcal{D} \times V$. Here, *dependency labels* $L_\mathcal{D}$ enumerate all kinds of possible dependencies.

$$L_{\mathcal{D}} = \overbrace{\{\mathit{Filter}, \mathit{Spawn}, \mathit{Close}, \mathit{Eval}\}}^{\text{Behavioral}}$$

$$\times \overbrace{\Big(\{\mathit{Sync}\} \cup \underbrace{\{\mathit{Async}\}}_{\text{Hold}} \cup (\mathit{Offset} \times \mathbb{N}) \cup \underbrace{(\{\mathit{Async}\} \times \mathcal{F}_H \times \mathbb{R})}_{\text{Aggregation}}\Big)}^{\text{Access Kind}}$$

Here, $\mathcal{F}_H$ denotes the set of list homomorphisms.

The creation of edges happens according to the following rules.

- An input stream $\sigma^{\downarrow}$ does not have outgoing edges.
- For an output stream $\sigma^{\uparrow}$, determine the edges and access kinds of a) $\mathit{expr}(\sigma^{\uparrow}.\mathit{spawn})$, b) $\mathit{filter}(\sigma^{\uparrow}.\mathit{spawn})$ and $\mathit{filter}(\sigma^{\uparrow}.\mathit{close})$, c) $\mathit{filter}(\sigma^{\uparrow})$, and d) $\mathit{expr}(\sigma^{\uparrow})$. Tag them with *Spawn*, *Close*, *Filter*, and *Eval*, respectively.
- For expressions occurring in $\sigma^{\uparrow}$, each synchronous access including offset accesses with offset 0, asynchronous hold access or access with offset $n > 0$ to σ' translates to an edge $(\sigma^{\uparrow}, \_, \sigma') \in E$, with access kind *Sync*, *Async*, or (Offset, n), respectively.
- Each aggregating access to σ' with aggregation function $\gamma \in \mathcal{F}_H$ and duration $d \in \mathbb{R}$ translates to $(\sigma^{\uparrow}, \_, \sigma')$ with access kind $(\mathit{Async}, \gamma, d)$.

The set of the resulting edges is then E.

Example 2.30 (Dependency Graph)**.** Recall the running example from Listing 2.4. Figure 2.10 depicts its dependency graph.

→Listing 2.4, p. 25

△

Remark 2.7 (Stream Instances as Vertices)**.** An alternative formulation is to create a vertex per potential stream instance. This would drastically blow up the graph. In return, it would enable a semantic analysis, so it could for example determine mutually exclusive stream instances. While such checks have merit, they are expensive to the point of infeasibility. So, since the following analyses do not require this information and still yield substantial information renders this definition preferable.

2.5.2. Well-Definedness

The notion of a well-defined specification was first defined for Lola [DAn+05] and translates well to the real-time variant. In either language, a well-defined specification has a unique model. Hence, there are two problematic cases: specifications without a model and specifications with several models. The following specification illustrates both points.

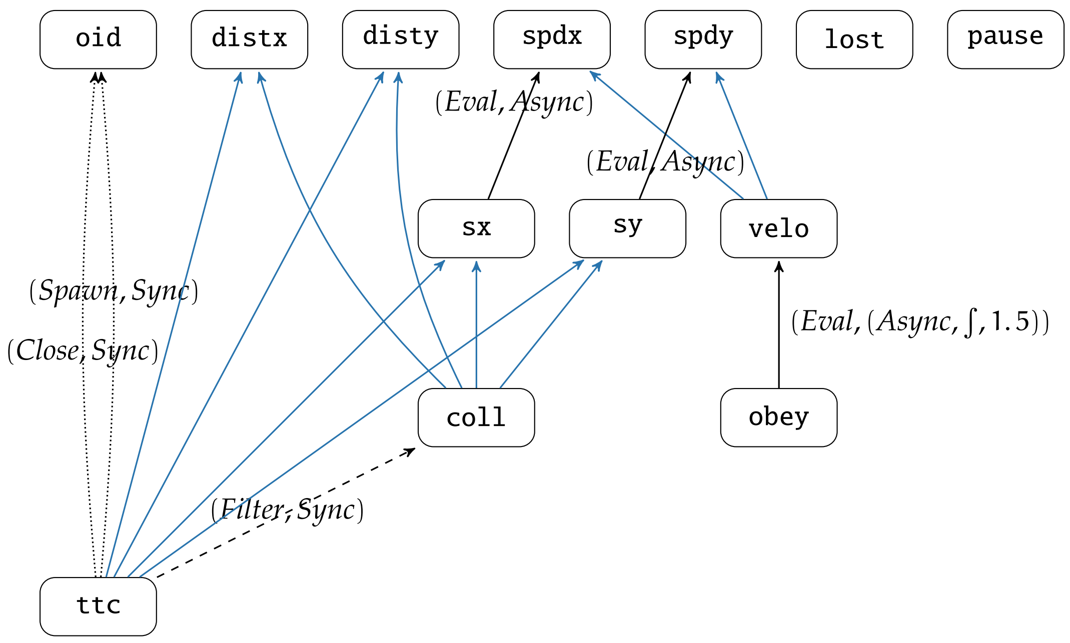


Figure 2.10.: Dependency graph of the running example specification. Blue edges have the label (*Eval*, *Sync*). The two disconnected nodes indicate that both streams are merely timing indicators; their values have to influence on the verdict.

```
output x: Bool := ¬x
output y: UInt64 := y
```

Evidently, the output x constitutes a contradiction whereas output y is tautological. As a result, there is no model for x and 2^{64} models for y.

While determining the lack or abundance of models is trivial in some cases, it is undecidable for unbounded types [DAn+05] and requires an expensive semantic analysis for bounded types. However, there is a simpler, albeit imperfect, syntactic check for Lola. Adapting it to RTLola is possible, yet demands some tweaks.

Well-Formedness

The *well-formedness* check for Lola specifications identifies all cycles occurring in the specification. Here, a chain of streams constitutes a cycle if they access each other, starting and ending in the same stream. Each stream access has a weight, which is 0 for synchronous accesses and $x \in \mathbb{Z}$ for accesses with offset x. Note that in Lola these cover all possible access kinds, however, x can be either positive or negative. A cycle then qualifies as 0-cycle if the sum of its weights amounts to 0. Based on this, well-formedness requires the absence of 0-cycles. It can easily be seen that both streams in the aforementioned specification constitute a 0-cycle by themselves.

The advantage of well-formedness is that it can easily and efficiently be determined based on a dependency graph, and it implies well-definedness [DAn+05]. On the downside, there are well-defined specifications which are not well-formed.

```
output z := z ∨ ¬ z
```

The tautology in the stream expression forces the value of `z` to be `true`. Hence, in spite of the 0-cycle, a unique model for `z` exists. Fortunately, in practice, well-formedness is an adequate criterion since cases like `z` barely ever occur.

Defining well-definedness for RTLola requires a closer look at why 0-cycles pose a problem in Lola and how to adequately lift the notion to take instance handling, asynchrony, and real-time into account. First off, observe that a cycle in the dependency graph indicates that all streams in the cycle influence each other. This is not a problem per se, provided the cycle can be resolved by taking the time axis into consideration. For a cycle with negative weight, successive unrolling of dependencies leads further and further into the past. At some point, this spiral passes the first evaluation of the stream. In this case, an access fails, so the semantics substitutes the default value, ultimately breaking the cycle. An analogous phenomenon happens for cycles with positive weights. The problem only persists for 0-cycles since there is no breaking element. While this is the only possible kind of cycle in Lola, the more fine-grained notion of time in RTLola gives rise to more options. For each of them, well-formedness demands a breaking element.

Legacy Cycle

Cycles on Lola translate to *legacy cycles* in RTLola. These occur when a cycle contains only $(\mathit{Eval}, \mathit{Sync})$-labeled edges. Just as in Lola, they can be broken with a $(\mathit{Eval}, \mathit{Offset}(n))$ edge. The absence of positive offsets renders this sufficient. Whether *Close*, *Filter*, or *Spawn* edges break legacy cycle becomes clearer when first discussing another sources of cycles.

Existential Cycle

An *existential cycle* indicates that the existence of a value depends on the (only *potentially* existent) value itself. An example clarifies the term:

```
input inp: Int32
output x(p: Int32)
    spawn with x
    eval with inp
```

Suppose `inp` produces the value 3, and no instance of `x` exists, yet. If any instance of `x` were to exist, it would produce the value 3, causing `x(3)` to be spawned. The output thus retroactively justifies both itself, and the creation of the outputting instance. On the contrary, if `x` would not produce a value, then there would be no reason for an output or a spawn, justifying the non-existence of both. Hence, multiple evaluation models exist.[16]

The analogous problem arises when a cycle contains a filter dependency, though more subtly. First, consider the following specification:

```
input inp: Bool
output x eval when x with inp
```

[16]Observe that existential cycles generally lead to a small increase of valid models, in this case two, whereas legacy cycles allow for as many models as the size of the domain of values in the cycle.

While there clearly is a cycle in the specification, there is a unique evaluation model. For this, assume inp generates a positive value. In this case, both the output stream and its filter would invariably need to produce the same value. Hence, the filter is satisfied, rendering the generation of a value mandatory. Similarly, if the value of inp were false, there would be only one valid evaluation model: the one in which x would not get extended.

The reason why this cycle can be uniquely resolved is that the point of reference can uniquely be identified: an equivalent specification can just inline the stream expression, breaking the cycle.

```
input inp: Bool
output x eval when inp with inp
```

So, in essence, there is no real cycle because the filter refers to the input rather than the output. This trick no longer works when introducing a stronger reference to x so that inlining is no longer possible.

```
input inp: Bool
input inp2: Bool
output x eval when inp2 ∨ x.last(or: true) with inp
```

Note that the second input is only necessary so that the value of x can be either true or false. In contrast, in the preceding example, the output will always be a potentially empty sequence of true-values.

In any case, this setup constitutes an existential cycle because the resolution of the offset depends on the existence of a new value of the output, which in turn depends on the resolution of the offset. Assume x already has produced several values with a suffix of $\langle true, false \rangle$. Suppose in the next position, both inputs generate a false-value. If x were to be extended, the filter would refer to the last value before the extension, i.e., false. Since inp is false, too, there is no justification for the extension, rendering the model invalid. Hence, let x not be extended. In this case, the offset refers to the second-to-last value of x, which is true. Hence, the filter is satisfied regardless of inp2, justifying an extension. Therefore, this evaluation model is also invalid, so no valid model exists.

The bottom line of these observations is that both filter and spawn dependencies can cause existential cycles. While some filter cycles yield a unique evaluation model, well-formedness for RTLola is an over-approximation and thus prohibits such specifications anyway. So, intuitively, these cases are the existential-cycle equivalents to the legacy cycle of the Lola specification with a tautological expression.

After identification of problematic cycles, the question is how they can be resolved. As mentioned before, for a legacy cycle, an offset suffices. The offset introduces a temporal dissonance between two values since the value at position i depends on one at position $i-1$. Either this value already exists, or there is a default value for it, resolving the dependency. Clearly, an offset is insufficient for existential cycles. Yet, there is another source for temporal dissonance: close-dependencies. Recall Definition 2.26.

→ Def. 2.26, p. 67

Here, the definition first determines all stream updates Σ^+_{i+1} and *afterwards* applies the closing semantics, causing to the dissonance. For an illustration, consider the following specification.

```
input inp
output x: Bool
    eval with inp
    close when x
```

The dependency graph connects the output stream with itself via a synchronous close, and a synchronous evaluation edge. In spite of the cycle, the specification has a unique evaluation model: the evaluation of `x` mirrors input `inp` up to and including the point where `x` becomes true. Only *after* this evaluation step, `x` terminates. Since termination is strictly separated from evaluation, the edge breaks the cycle.

Another source for non-simultaneity is a transition from event-based to periodic streams and vice versa. This is due to the assumption that events do not coincide with periodic deadlines. While in theory this is possible, the assumption is safe in practice due to the high clock frequencies of current hardware.[17] Any edge between an event-based and a periodic stream needs to be asynchronous. Since both streams cannot be evaluated at the same time, the resolution is trivial.

Taking the information about cycle resolution into account gives rise to the *Immediate Dependency Graph*, discarding all edges introducing a temporal dissonance.

Definition 2.31 (Immediate Dependency Graph)

The *immediate dependency graph* $\mathcal{D}^*_\Phi = (V, E^*)$ of a specification Φ is a reduction of the dependency graph $\mathcal{D}_\Phi = (V, E)$ to immediate accesses, i.e.,

Def. Immediate Dependency Graph

$$E^* = \{(\sigma, \ell, \sigma') \in E \mid \ell \neq (\mathit{Close}, \_) \wedge \mathrm{periodic}(\sigma) = \mathrm{periodic}(\sigma')\}$$

Here, $\mathrm{periodic}(\sigma)$ is a boolean flag determining whether σ has a periodic pacing.

This leads to the following definitions of well-formedness for RTLola.

Definition 2.32 (Well-Formedness)

A specification Φ is *well-formed* iff all cycles in the immediate dependency graph $\mathcal{D}^*_\Phi = (V, E^*)$ either contain neither a *Filter* nor a *Spawn*-edge, and an offset edge.

Def. Well-Formedness

$$\forall \{(\sigma_1, \ell_1, \sigma_2), \dots, (\sigma_n, \ell_n, \sigma_1)\} \subseteq E^*: \quad \exists i\colon \ell_i = (\mathit{Offset}, \_) \wedge \forall i\colon \ell_i \neq (\mathit{Filter}, \_)$$

Note that by Definition 2.29, filter edges cover both the filter in the evaluation and spawn clause.

→Def. 2.29, p. 70

Well-definedness is then defined in the same way as for Lola.

[17]On a technical note: theoretically, the probability that an event coincides with a periodic deadline on the real time axis is 0. Practically, since the monitor is a clocked machine, the probability is exceedingly low, yet strictly greater than 0.

Definition 2.33 (Well-Definedness)

Def. Well-Definedness

An RTLola specification is *well-defined* iff it has exactly one evaluation model.

The goal now is to prove:

Theorem 2.1 (*Well-Formedness implies Well-Definedness*)

A well-formed RTLola specification is well-defined.

This first requires a couple of lemmas.

Lemma 2.34 (Well-Defined Arithmetic)**.** *Any expression occurring in an RTLola specification is well-defined provided stream accesses are well-defined.*

Proof Trivial structural induction. Any leaf in the expression tree is a uniquely defined constant or a stream access, which is well-defined by assumption. Any operator in RTLola is total and thus well-defined provided its operands are well-defined. This is the case by induction hypothesis. □

The next lemma first requires the definition of an evaluation order:

Definition 2.35 (Evaluation Order)

Def. Evaluation Order

Let Φ be a well-formed RTLola specification. Let $\mathcal{D}^*_\Phi$ be its immediate dependency graph. An *evaluation order* $\prec \subseteq \mathcal{S} \times \mathcal{S}$ is the least restrictive partial order on streams satisfying the following property: First, for two streams σ and σ', $\sigma \prec \sigma'$ iff there is a path from σ' to σ through $\mathcal{D}^*_\Phi$ without *Offset*-labelled edges. Second, if there are three streams $\sigma_1, \sigma_2, \sigma_3$ with an $(\mathit{Eval}, (\mathit{Offset}, \_))$-labeled edge from σ_1 to σ_2 and a $(\mathit{Filter}, \_)$-labeled edge from σ_2 to σ_3, then $\sigma_3 \prec \sigma_1$.

The latter criterion seems oddly specific and thus warrants an example.

Example 2.36 (Offset-Filter Dependencies)**.** Consider the following sketch of a specification.

```
output a := b.offset(by: -1, or: _)
output b eval when c with _
output c := _
```

The evaluation of a accesses the last value of b. However, resolution of the *last* value requires information on whether b will be extended, which depends on c. Hence, c $\prec$ a. Yet, it does not depend on the actual value of b, so the evaluation order does not impose any particular order between a and b.

Note that the same phenomenon occurs when the filter dependency is part of the spawn clause. This also results in a *Filter*-labeled edge and thus requires no further attention. △

This evaluation order always exists, as captured by the next theorem.

Theorem 2.2 (*Unique Evaluation Order*)

The evaluation order of a well-formed RTLola specification exists and is uniquely defined.

The constructive proof immediately yields the evaluation order.

Proof Observe that — barring offset-edges — the immediate dependency graph is not necessarily a connected graph, but it is directed and acyclic graph. This follows from the Definition 2.32. As such, the reachability relation $\prec^+$ of the reduced graph yields a partial order satisfying the first criterion. For the possibility to refine $\prec^+$ into $\prec$ in accordance with the second criterion, it suffices to show for any affected pair $\sigma \prec \sigma'$ it holds that $\sigma' \not\prec^+ \sigma$.

Proof by contradiction. Suppose $\sigma' \prec^+ \sigma$. In this case, there is a path π from σ' to σ. Concatenating the path π' from σ via the filter to σ' yields a cycle $\pi\pi'$ with at least one edge labeled as filter edge. This contradicts the well-formedness of the specification. □

Clearly, the partial order can be strengthened into a total order.

Corollary 2.37 (Existence of a Strong Evaluation Order). *For a well-formed RTLola specification there is a total order on streams such that it satisfies both criteria of an evaluation order. Such an order is a* strong evaluation order.

Def. Strong Evaluation Order

Proof Any total order agreeing with $\prec$ such as a topological order of $\mathcal{D}^*_\Phi$ without offset edges is a strong evaluation order. □

The next lemma has fundamental implications as its proof constructively states *that and how* a specification can be evaluated.

Lemma 2.38 (Well-Defined Atomic Update). *For a well-formed RTLola specification and a fixed monitor model Σ, an atomic update as per Definition 2.26 without termination is well-defined.*

→ Def. 2.26, p. 67

Proof Induction over the evaluation order $\prec$. This order disregards offset edges, so they will be treated separately. Also note that the absence of side effects allows for skipping the evaluation of edges if their result is irrelevant. Last, let non-immediate edges be edges in $\mathcal{D}_\Phi$ but not in $\mathcal{D}^*_\Phi$.
Induction Base. Let $\sigma \in \mathcal{S}$ such that $\sigma \prec \sigma'$ for any other $\sigma' \in \mathcal{S}$. Hence, it only contains arithmetic expressions, constants, accesses representing non-immediate edges, and offset edges occurring in both dependency graphs.

By Lemma 2.34, the first two kinds of constituents are well-defined in RTLola. For the third kind, note that close edges are irrelevant for atomic updates without termination. For edges connecting two streams with a transition from event-based to periodic or vice versa, recall that their pacing types are mutually exclusive. Hence, the atomic update cannot affect instances of the target stream. Thus, the result of every stream access is uniquely defined.

Last, for offset accesses with edge $(\sigma, (\mathit{Offset}, n), \sigma')$, the evaluation either accesses the n^{th}-to-last value if σ' will be extended in this atomic update, or the $n + 1^{\text{st}}$-to-last value if it will not. This ambiguity can only be resolved by first determining whether σ' will be extended and/or spawned which depends on the evaluation and spawn filters. Recall that by definition of the dependency graph (Definition 2.29), accesses in either of these expressions result in *Filter*-labeled edges.[18] Targets of these edges cannot in turn depend on another stream σ'' because otherwise, by the second requirement on the evaluation order (Definition 2.35) leads to $\sigma'' \prec \sigma$. This would contradict the assumption on σ. Hence, the filter condition of the accessed stream σ' is an expression without outgoing dependencies, so its value is well-defined as argued before. After evaluating the filter of σ'', the offset can be resolved unambiguously. Note that this line of argument also hold for $\sigma = \sigma'$.

→Def. 2.29, p. 70

With all dependencies resolved, the evaluation proceeds as follows: First it evaluates the spawn filter $\mathit{filter}(\sigma.\mathit{spawn})$. If the outcome is positive, it evaluates $\mathit{expr}(\sigma.\mathit{spawn})$, creates a stream instance ι, and adds it to Σ. Note that this evaluation cannot depend on other streams as per assumption on σ. Then, it evaluates its evaluation filter $\mathit{filter}(\sigma)$. If the output is positive, it evaluates $\mathit{expr}(\sigma)$, and adds the result to $\iota.\mathit{vals}$. *Induction Step.* Let $\sigma \in \mathcal{S}$ have outgoing edges in $\mathcal{D}^*_\Phi$. The evaluation of σ proceeds in the same way outlined in the induction base. There are only two differences in the reasoning: it can no longer rely on the assumption that σ is a leaf in $\prec$, and there is a fifth kind of constituent occurring in expressions. For the first difference, the assumption can be replaced with applications of the induction hypothesis without further ado. Any dependent stream is lower in the evaluation order and can thus be resolved unambiguously. The same holds for the fifth kind of expression constituents, which are non-offset edges occurring in both dependency graphs.

This concludes the proof. □

2.5.3. Evaluation Order

The evaluation order is a side-product of the proof for well-definedness. Apart from that, it proved exceedingly useful when realizing a monitor for a specification. Not only does it help to guarantee the uniqueness of an evaluation model, following it also yields the *correct* model. Consider the following specification:

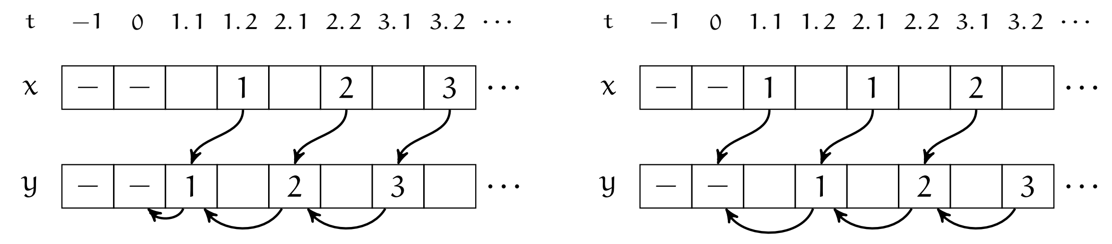
```
output x: Int32 := y + 1
output y: Int32 := y.last(or: 0) + 1
```

By the first condition on the evaluation order: $y \prec x$. Following this order, i.e., evaluating y before a is essential to obtain the correct result as Figure 2.11 illustrates; the syntactic order leads to incorrect values.

Moreover, there are two flavors of the evaluation order, the regular one according to Definition 2.35 and the strong one according to Corollary 2.37. While Chapter 3 covers all

→Chap. 3, p. 97

[18]Note that *Spawn* edges only determine the parameters of newly created instances, hence they are irrelevant here.



(a) The result of evaluating the output streams respecting the evaluation order.

(b) The result of evaluating the output streams in order of their declaration.

Figure 2.11.: Two different evaluations of the output streams a and b, where a accesses b synchronously and b accesses its previous value. Both accesses default to 0 and both a and b increase the obtained value by 1.

the details, the regular order is useful when realizing monitors on a platform that allows for cheap concurrency. Conversely, the strong order is used for sequential evaluations.

Lastly, note that the evaluation order separates a specification into several layers.

Definition 2.39 (Evaluation Layer)

Let $\prec$ be the regular or a strong evaluation order of a well-formed RTLola specification Φ. A stream which is minimal with respect to $\prec$ is in *evaluation layer* 1. Any other stream is one layer above the greatest layer of a stream on which it depends.

Def. Evaluation Layer

$$\text{Layer}_{\prec}(\sigma) = 1 + \max_{\sigma \in \mathcal{S}} \{i \mid \text{Layer}_{\prec}(\sigma') = 1 \land \sigma' \prec \sigma\}$$

Let $\lambda^* = \max_{\sigma \in \mathcal{S}} \{\text{Layer}_{\prec}(\sigma)\}$ denote the greatest evaluation layer.

Omission of the subscript refers to the regular evaluation order of the specification.

2.5.4. Memory Bounds

The next analysis for an RTLola specification is the memory bound analysis. This is in line with MII: RESOURCE-AWARENESS: a monitor realization of a specification is guaranteed to require at most a set, statically determined amount of memory. This information is instrumental when deploying an RTLola monitor in a CPS. It allows for verifying that the available resources suffice for the monitor independent of the dynamic behavior of the system. Another neat — though less critical — advantage of this information is that a can monitor statically allocate all memory it requires. This alleviates the need for costly dynamic memory allocation and de-allocation.[19]

[19]However, note that while dynamic allocation is not necessary, it can still be desirable, in particular when considering instance handling.

The analysis proceeds stream by stream. It identifies three potential sources for memory cost: value persistence, space for expression evaluation, and pre-aggregation.

Value Persistence

Value persistence states how many values of a specific stream need to be persisted. Here, the analysis flatly reserves memory for one value by default. It then determines which incoming edges compel a monitor to store more than one value of the stream. Here, the dependency kind, i.e., *Eval*, *Filter*, *Spawn*, or *Close*, is irrelevant. Moreover, most access kinds incur no additional cost. Synchronous- and hold-accesses refer to the latest value of the stream, which is already covered. The memory cost for aggregations is a separate item. Lastly, an access with offset n demands memory for an additional $n - 1$ values of the target stream. The maximum offset — or 1 if there is none — then yields the *memory bound of the stream*.

Stream Memory Bound

Expression Evaluation

The next source for a memory requirement is the *evaluation of expressions*. For logical and arithmetic expressions, this boils down to the conventional analysis regarding the number of registers required to evaluate an arithmetic expression [Sch71], multiplied with the sizes of the values. Intuitively, the evaluation of an arithmetic expression requires enough space to evaluate each operand sequentially plus storage of the result. As an example, consider the expression $1 + 3 * 5$. The monitor would first evaluate the multiplication, i.e., load the constants 3 and 5, and then multiply them and store the result. It would then load the constant 1 and add both values. Hence, two registers suffice for the evaluation whereas a naive solution requires three. The remaining expressions, i.e., stream accesses and aggregations merely require space to store a single value of the target stream or result type of the aggregation, respectively. Let $\text{cost}(e)$ denote the memory requirement for evaluating an expression e as per Schneider [Sch71].

Pre-Aggregation

Next, the *pre-aggregation* cost corresponds to storing all intermediate values for an aggregation (cf. Appendix A.1.3) at once.

Finally, the analysis also takes parametrization into account. To this end, the size of the domain of the parameters of a stream states how many instances can exist at once. This value needs to be multiplied with the product of the memory bound of the stream and its value size, and with the pre-aggregation cost. Note that the sum of these values yields the memory requirement barring expression evaluation and is universal, i.e., it does not depend on implementation details of the monitor. In contrast, the expression evaluation cost is implementation-dependent. Suppose the monitor works sequentially, i.e., it evaluates one stream instance after another. In this case, it only needs to reserve enough memory to evaluate the most costly expression. For parallel evaluation models, the computation can vary vastly. For example, suppose the evaluation of a monitor proceeds according to the evaluation layers. In this case, the expression evaluation cost for a layer is the sum of the expression cost of each stream in the layer, multiplied by the number of instances the stream can have. The overall expression evaluation cost then amounts to the greatest cost of a single layer.

For this reason, the following definition considers only the sequential evaluation model, as this one is the most relevant for monitoring CPS.

Definition 2.40 (Memory Bound)

Let Φ be a well-formed RTLola specification and $\mathcal{D}_\Phi = (V, E)$ its dependency graph. The *memory bound* $\mu(\sigma)$ of a stream σ is defined as:

Def. Stream Memory Bound

$$\mu(\sigma) = \max\left\{1, \max_{e \in E}\{n \mid e = (\_, (\mathit{Offset}, n), \sigma)\}\right\}$$

Suppose size(σ) produces the number of bytes required to store a single value of σ and dom(σ) the domain of the parameters of σ, i.e., the number of instances the stream can have. This defaults to 1 for non-parametrized streams.

Then, the *total memory bound* for a sequential monitor for Φ is:

Def. Specification Memory Bound

$$\mu(\Phi) = \max_{\sigma^\uparrow \in \mathcal{S}^\uparrow}(\mathrm{cost}(\mathit{expr}(\sigma^\uparrow))) + \sum_{\sigma \in \mathcal{S}} \mathrm{dom}(\sigma)\left(\mu(\sigma)\,\mathrm{size}(\sigma) + \sum_{e \in E} \mu^\omega_\sigma(e)\right)$$

Here, $\mu^\omega_\sigma(e)$ is the memory cost an aggregation edge incurs defined as follows where T_f is the type of the pre-aggregation for a list homomorphism f.

$$\mu^\omega_\sigma(e) = \begin{cases} \lfloor n\pi \rfloor \cdot \mathrm{size}(T_f) & \text{if } e = (\sigma, (\mathit{Async}, f, n), \_) \wedge \pi = \mathrm{pace}(\sigma) \\ 0 & \text{otherwise} \end{cases}$$

2.6. Implementation

After covering the theoretic foundation for RTLola, this section presents the implementation and evaluates it empirically. The implementation is written in Rust, publically available, and divided into several modules. These modules can be separated into a frontend and several backends. Figure 2.12 shows each step in the frontend and available backends.

2.6.1. Frontend

Frontend

The RTLola *frontend* [Bau+20b] takes a specification in text form as input and ultimately generates a data structure that contains all relevant information regarding the specification including analysis results. This process once again is organized into several submodules.

Parser As the name suggests, the RTLola parser parses a specification first into a homogenous tree using the *pest* parser generator [Tis18]. This tree is then transformed into an abstract syntax tree, on which syntactic sugar is removed.

High-Level Intermediate Representation While the Ast is an abstract representation of the syntax, its proximity to the syntax renders analysis steps inconvenient. Hence, the RTLola frontend generates a *high-level intermediate representation* (Hir), specifically designed for convenience and extensibility[20]. This flexibility renders operations on the tree such as accessing details of a stream slow in comparison to rigid data structures.

High-Level Intermediate Representation

Type-State Pattern

Internally, the Hir employs the *type-state pattern*. This is a variation of the behavioral *state pattern* [Gam+95], a design pattern in which an object mimics the behavior of a state machine. The type-state pattern extends this idea by incorporating the current state of the machine into the type of a parametrized type. Hence, the Hir is parametrized by an indicator regarding which analysis steps were already performed and thus which information is readily available. Performing a dependency analysis, for example, uses an `Hir<Typed>` as an input, generates a dependency graph, and incorporates its information into the Hir to yield an `Hir<DepsAnalyzed>`. This unlocks a function for determining the evaluation order of the specification. The two most notable steps are the creation and analysis of the dependency graph via the graph representation and algorithm library *petgraph* [bm15] and the type inference via the type checking library *RustTyC* [Sch19b].

Mid-Level Intermediate Representation After completing all analysis steps, the RTLola frontend remedies the Hir-induced drawbacks by transforming it into a *mid-level intermediate representation* (Mir). This is essentially a copy of the Hir less all flexibility: it stores all information collected during analysis steps at fixed points in

Mid-Level Intermediate Representation

[20]A similar concept can for example be found in Moss et al.'s work [MDM16]

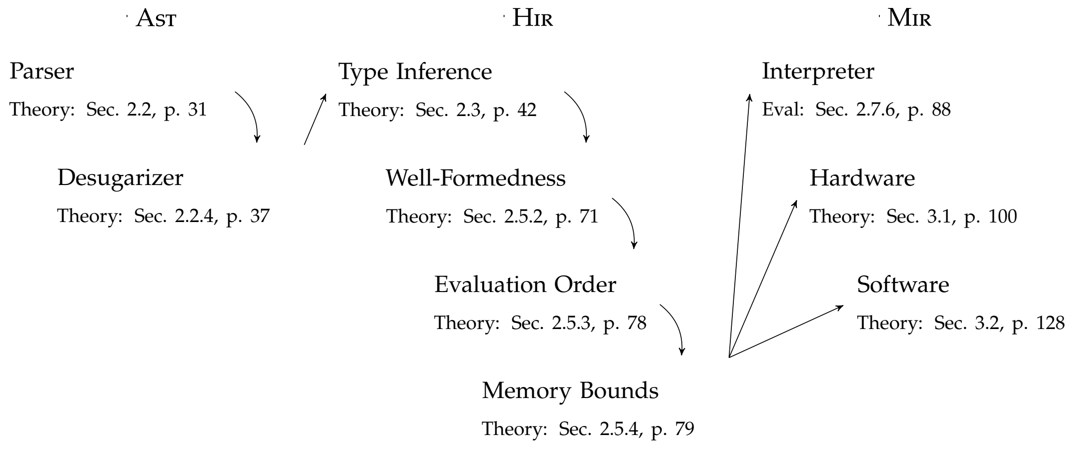


Figure 2.12.: Overview over each significant step in the RTLola frontend and all available backends.

the data structure and removes variable concepts such as the type-state pattern. The resulting Mir is similarly convenient to use, more performant but far less extensible than the Hir.

2.6.2. Backends

RTLola backends build upon the frontend. They take an Mir as input and generate some form of executable monitor. When executing the interpreter backend, for example, it receives inputs successively over the standard input or an input file. During this process, it generates outputs in form of output values or trigger notifications.

Compiler backends such as the VHDL- and Rust-compiler presented in the next chapter instead generate code which can in turn be compiled into an executable monitor. The usual advantages and drawbacks of interpretation versus compilations apply for RTLola as well. In a nutshell, the performance of the interpreter is generally worse than the one of a compiled monitor. However, the time it takes to get the result of the monitor when starting with a specification and an input trace can still be lower when employing interpretation. The reason behind this is that interpretation is a two-step process: analysis via the frontend and then the interpretation. In contrast, compilation requires a pipeline consisting of analysis, compilation to code, compilation to executable, and invocation of the executable. The overhead induced by the compilation steps can exceed the benefit gained for a more performant monitor. This is particularly true when realizing the monitor in hardware (cf. Section 3.1) since synthesis of a hardware description onto a piece of programmable hardware is costly. →Sec. 3.1, p. 100

This factor becomes significant when designing a specification since a prototype should be *validated*. For this, the specifier runs the monitor on a static input file and compares the output against the expected result, similar to end-to-end tests in software engineering. If the outputs match, this is evidence for the adequacy of the specification. Here, reducing the time and effort required to get from a prototype to the output of the monitor enables rapid prototyping. Note that this form of prototyping validation perfectly complements the static analysis of the specification via type checking, well-formedness analysis and the determination of a memory bound.

Clearly, after the specification is fixed, compilation is preferable when deploying the final monitor.

Remark 2.8 (Acknowledgement)**.** The whole RTLola code base grew over several years and would not have been possible without the effort of Jan Baumeister, Stefan Oswald, Malte Schledjewski, Marvin Stenger, and Leander Tentrup. In particular, I want to acknowledge Florian Kohn who recently put significant effort into vastly improving and extending the code base.

Each submodule of the frontend and the interpreter is available on the Rust code publication platform *crates.io* [Tea10].

2.7. Empirical Evaluation

The following is an empirical evaluation coupled with a discussion of the time complexity for each major step from a specification to the generation of an output via interpretation. These major steps are parsing, type inference, well-formedness analysis, determination of memory bounds and interpretation. All experiments were conducted on a machine with an Apple M1 chip and 16 GB of memory.

The results reported and discussed here are all averages over 100 runs with the same input specification of varying size. The average is meaningful since the standard deviation never exceeded 0.5% of the mean. In these experiments, the shape of the input specification has only a negligible effect on the runtime performance of each analysis step except for the type inference. The reason behind this is that the parser is agnostic of dependencies and thus only scales in the length of the specification. Hence, the only varying factors are the size of expressions versus the number of streams. However, an empirical test showed that only the number of tokens was relevant, so scaling either factor is fine. Any other non-type-related analysis visits every node and edge of the graph once, hence it makes no discernible difference whether the former or latter is increased. For the type check, a *chained specification* constitutes the worst case, as detailed below. Thus, this specification is the foundation for all experiments. It is a series of streams chained in a way that the i^{th} stream accesses the $i + 1^{st}$ stream.

Chained Specification

2.7.1. Parser

The RTLola language falls into the category of parsing expression grammars [For04]. The main advantage of this grammar is that parsing has a linear time complexity and there are excellent off-the-shelve parser generators. Hence, RTLola resorts to such a parser generator [Tis18].

Figure 2.13 depicts the time performance of the parser in relation to the well-formedness, evaluation order, and memory bound analysis. The parsing time peaked at 254.03 µs for a specification with 200 streams.

2.7.2. Type Inference

The type inference algorithm passes over the specification multiple times, once for each of the six subtypes, i.e., value type, evaluation filter type etc. Each pass itself carries a context. This allows for accessing information obtained in previous passes. Internally it uses a type inference library [Sch19b] for type systems similar to *Hindley-Milner type systems* [Hin69; Mil78; Dam84]. The library is based on a modified union-find [GF64] backend. Both the implementation of the type inference and the underlying union-find backend are optimized for maintainability rather than performance. Yet, as the empirical evaluation shows, it performs sufficiently well even for large specifications.

Hindley-Milner Type System

Remark 2.9 (Hindley-Milner Type System)**.** The Hindley-Milner type system, also known as the Damas-Hindley-Milner type system, was first formulated by Hindley in 1969 [Hin69], rediscovered by Milner in 1978 [Mil78], and extended by a formal analysis courtesy of Damas' PhD thesis [Dam84] a few years later in 1984.

While it has a strong theoretical background, it performs remarkably well in practice and captures many functional programming languages. Despite the DExpTime-complete complexity of type inference, it often scales linearly in the size of the code.

Out of all specification shapes tested, the chained specification constituted the worst case for the RTLola type inference. It contains the least permitted amount of type annotations and streams are ordered in a way such that the information necessary to perform a union operation in the union-find data structure is placed at the end of the sequence. This delays the operation as much as possible. However, note that this does not relate to the worst case behavior for Hindley-Milner type systems. Nor is it provably the worst case for the RTLola type inference since the precise complexity is an open question.

Typical for Hindley-Milner type system, the type inference is expected to scale linearly. However, as can be seen in Figure 2.14, this is not the case. In part, this is a consequence of a suboptimal implementation of the underlying union-find backend.

In any case, while the running time dominates the one of any other frontend component, its absolute value is low. Specifications with 200 streams require less than $60\,\mathrm{ms}$, all other analysis steps combined take $370.3\,\mu\mathrm{s}$. This still renders the type inference suitable for practical specifications.

2.7.3. Well-Formedness

The well-formedness analysis consists of three steps. First, it creates the immediate dependency graph. This is a simple process running in linear time in the number of streams. It then collects all cycles occurring in the graph. To this end, it performs a depth-first search for back edges, i.e., an edge from a stream to one of its ancestors. Each such back edge constitutes a cycle. The whole search is linear in the size of the graph, which is the number of vertices plus the number of dependencies. Last, it asserts that each cycle contains an offset edge and neither a filter, nor a spawn edge. With appropriate bookkeeping during the search, this process incurs only constant additional overhead. Hence, in total, the running time is linear in the size of the specification. Figure 2.13 is evidence in favor of this proposition. It also shows that the running time barely exceeds $0.1\,\mathrm{ms}$ for the largest specification ($106.31\,\mu\mathrm{s}$).

2.7.4. Evaluation Order

The evaluation requires computing the topological order via a depth-first search. Alternative approaches like Kahn's algorithm [Kah62] would also be possible. Both algorithms

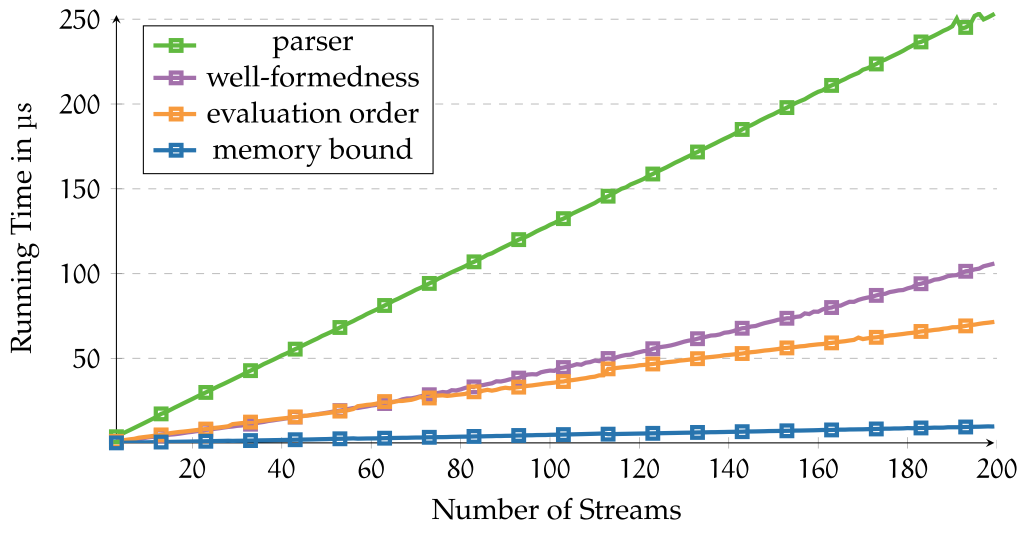


Figure 2.13.: The running time of the parser, well-formedness, evaluation order, and memory bound analysis plotted against the number of streams in the underlying chained specification. Each data point is the average over 100 runs.

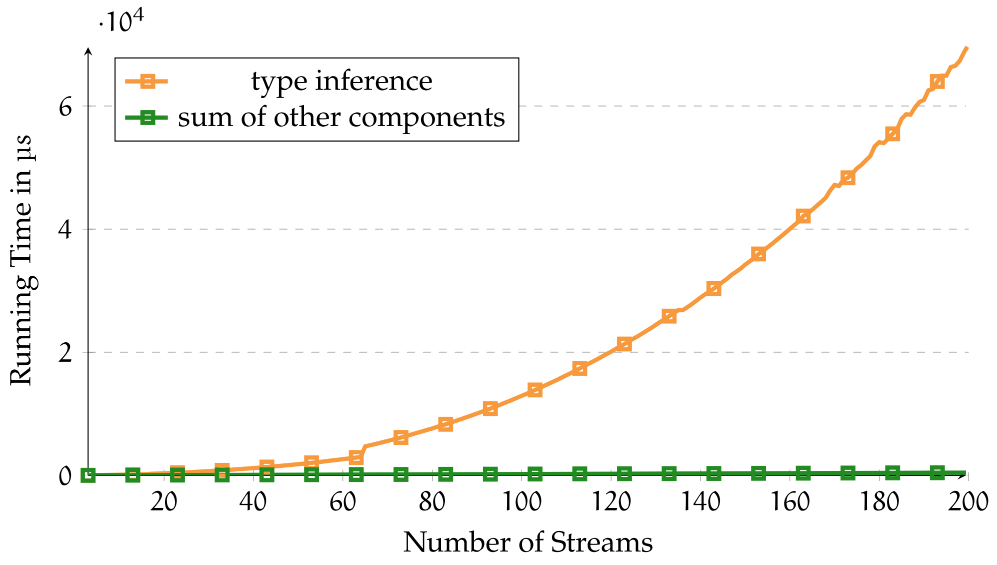


Figure 2.14.: Running time of the type inference in relation to the sum of all other frontend components, eclipsing them by far. Each data point is the average over 100 runs for chained specifications of varying length.

scale linearly in the number of streams plus number of edges. Figure 2.13 shows that the computation requires at most 71.76 µs for a specification with 200 streams.

2.7.5. Memory Bounds

The memory analysis inspects each outgoing edge for each stream. Hence, the worst case complexity is linear in the number of edges. Figure 2.13 validates this conclusion. Here, the running time always remains below 10 µs.

2.7.6. Interpreter

By design, the interpreter is entirely linear: the worst case running time per event is linear in the number of stream instances, linear in the size of the expression, and — since the number of instances is statically bounded — linear in the number of events, including periodic computations, in the input trace. Hence, for the evaluation, realistic specifications are more interesting than synthetic ones.

First, consider the following specification for an autonomous aircraft. As inputs, the specification takes the current time in microseconds, the acceleration in longitudinal and latitudinal direction, and the expected reference speed. It determines the frequency with which the monitor receives input data and validates that it does not fall below a threshold. Moreover, it warns when the actual speed — obtained by integrating the acceleration indefinitely — deviates strongly from the reference value, and counts how often this happens.[21] Lastly, it monitors the frequency of such deviations.

```
input time, accel_x, accel_y, speed: Int32
output count := count.last(or: 0) + 1
output frequency := 1 / (time - time.last(or: 0))
output freq_sum := frequency + freq_sum.last(or: 0)
output freq_avg := freq_sum / count

output speed_x := accel_x.integrate(over: ∞)
output speed_y := accel_y.integrate(over: ∞)
output speed := speed_x*speed_x + speed_y*speed_y

output unchanged := if res_max.last(or: false) then 0 else
    unchanged.last(or: 0) + 1
output velo_dev := abs(velo_r_x - velo_x) + abs(velo_r_y - velo_y)
output strong_velo_dev := velo_dev > 10
output count_devs: Int32 :=
    count_devs.last(or: 0) + if strong_velo_dev then 1 else 0
```

[21]Note that in reality, specifiers would ground the integration via location data obtained from e.g. GNSS [22] modules since long-term integration is subject to drift.

```
trigger freq_avg < 10 "Low input frequency."
trigger strong_velo_dev "Deviation between velocities too high."
trigger count_devs / count > 0.001 "Frequent deviation in velocities"
```

In addition to the specification, the interpreter requires an input trace. For this experiment, a trace of length 433,000 events was generated in the ArduPilot simulator[23], a state-of-the-art simulator for aircraft. Note that the actual values barely have an impact on the runtime performance of the interpreter. Solely optimizations in the evaluation of conditionals can lead to a slightly better or worse performance. In total, the running time for the trace was 664.7 ms. This amounts to an average of 1.535 µs per event.

There are two major factors to consider for this number. First, it does not reflect on the runtime performance of a monitor after deployment of a CPS. The experiment was conducted on a highly performant laptop disregarding any particular concerns towards its weight, cost, or power consumption. This is detached from a realistic scenario. However, the second point is that the interpreter is not supposed to reflect reality. It is a tool for rapid and convenient validation of a specification for a sample trace. In this regard, the running time is perfectly sufficient.

The memory consumption is split into heap memory and stack memory. While the total memory consumption amounts to 16 MB, this includes the in-memory representation of the MIR as well as the code for the interpreter and major parts of the Rust standard prelude. The actual working memory of the interpreter entirely resides on the stack as it does not rely on dynamic allocation. The total stack size lies below 1 kB.

Once again, these two numbers do not translate to embedded monitors without further ado. First, an embedded realization does not import the entire rust prelude.[24] This reduces the memory requirement for code. However, more importantly, the working memory consumption is comparable. A monitor does not require more working memory than the interpreter barring potential padding of values due to a different word size and thus memory layout.

These considerations also hold for the next experiment. For this, consider a network monitor. Its specification fixes the IP of the host and checks network traffic based on the source and destination IP of requests, TCP flags, and the length of the payload. Recall that, in its core, the type `IPv4` is just a 32-bit integer.

The first output stream determines whether an event represents an incoming connection by filtering for `dst = host`. It issues a warning when the host receives over 10,000 connections within a second. Next, it filters the `length` stream to only contain values for incoming push connections. The monitor reports a spike in workload if the sum of bytes received this way grows over 10 MB s^{-1}. Last, the specification declares two streams, one counting events opening a connection with the host, one counting events

[23]`https://ardupilot.org`; last accessed: 02.02.2022
[24]Embedded rust code is `no_std`, i.e., only the bare essentials of the rust prelude is included.

closing connections with it. The latter count exceeding the former indicates some kind of mismatch that has to be addressed.

The resulting specification looks as follows:

```
input src, dst: IPv4
input fin, push, syn: bool
input length: UInt16

output host: IPv4 = ...

output incoming_connection: NoValue
    eval when dst = host
trigger @1Hz incoming_connection.count(over: 0.5s) > 10000
    "Flood of incoming connections"

output data_received @incoming_connection :=
    eval when push with length
trigger @1Hz received.sum(over: 1s) > 10^7 "Workload too high"

output opened @incoming_connection
    eval when syn with opened.last(or: 0) + 1
output closed @incoming_connection
    eval when fin with closed.last(or: 0) + 1
trigger open - closed < 0 "Closed more connection than were open"
```

The input trace stems from the Mid-Atlantic Collegiate Cyber Defense Competition (MACCDC)[25]. Here, each event took an average of 438 ns. This better runtime performance reflects the lower computational complexity of the specification; most expressions are simple additions or integer comparisons. The memory consumption is almost identical to the one for the last experiment owing to the similar code and MIR size, as well as the low stack requirement of less than 1 kB.

[25] https://www.netresec.com/?page=MACCDC; last accessed: 02.02.2022

2.8. RTLola in Practice

The advantages of RTLola render it an interesting choice for practical applications, in particular in avionics. From the start, RTLola was co-developed with engineers of the German Aerospace Center (DLR[26]). This ongoing cooperation resulted in several insights and developments in terms of monitoring autonomous aircraft [Ado+17; Tor+17; Ado+18; STA18; DFS21]. In particular, in the context of this thesis, several specifications were created [Bau+20a]. They contained checks regarding conformance to a geofence, i.e., the system had to remain within a pre-defined area, as well as sensor validation, and sensor cross-validation. The hardware compiler presented in the next chapter (Section 3.1) translated the specification into a hardware description, which was subsequently synthesized onto a programmable hardware board. This board was then deployed onto an unmanned aircraft, the superARTIS, which is part of DLR's Autonomous Rotorcraft Testbed for Intelligent Systems (ARTIS). Over a series of test flights, the monitor observed the system. To validate the effectiveness, the specification was designed in a way that the planned flight path intentionally violated the geofence. Under realistic circumstances, the detection of the violation would result in the termination of the flight or a take-over by a safety pilot. For the test flight, the planned warning was registered and discarded.

→Sec. 3.1, p. 100

In summary, the test flight was an absolute success. It proved the monitor both effective, and sufficiently efficient to be deployed on such a CPS. For this reason, future work aims at shifting further responsibility onto the RTLola monitor.

While the cooperation with DLR continues, a new project[27] started recently. In this project, RTLola plays a major role in the development of monitoring systems for electrical, vertical take-off and landing operations. This work is in cooperation with a leading German manufacturer of electric multicopters for use as air taxis. The project partners also meet with authorities of the European Union Aviation Safety Agency to discuss steps towards integration RTLola in the certification process.

2.8.1. Further Application Areas

Though the avionic use case is most mature, RTLola is also interesting for other domains. Hence, it is part of an Android app for monitoring emission data of cars, and there are first endeavors in preparing RTLola for deployment on medical CPS.

Automotive

RTLola is part into an Android App with the name "RTLola on Board" [Bie+21]. The app allows users to conduct their own real driving emissions (RDE) test drives. These tests are regulated by the European Union [Tut+15; The17] and aim at measuring the

[26]The initialism stands for Deutsches Zentrum für Luft- und Raumfahrt.

[27]`https://cispa.de/en/research/funded-projects-and-collaborations`, project VoloSTreAM; last accessed 20.01.2022.

concentration of pollutants like nitrogen oxides — commonly referred to as NO_x— in the emission of cars under realistic driving circumstances. To this end, a car is equipped with a portable emission measurement system (PEMS). It then has to drive for a certain amount of time in an urban and rural environment, as well as on the motorway. Among other constraints, the ratio of each environment and the driving dynamics have to remain within certain bounds for the test to be valid. Afterwards, the measured emission data reveals whether the car complies with the regulation.

For end-users, it is usually not possible to accurately reproduce the test environment since the price of a PEMS is in the realm of €250,000. However, they can approximate the result by instead using the mandatory [The98] on-board diagnostics interface (OBD). A cheap adapter, costing usually less than €20, transmits diagnostic data produced by the car via Bluetooth. This allows the RTLola on Board app to receive the information and assess whether the reported emission is within the permitted bounds. Moreover, it assists users in conducting the test drive by visualizing the share of each environment according and comparing it against the permitted limits.

Though not perfectly accurate, this enables users to validate the approximate emissions of their cars, such that they no longer have to blindly trust the manufacturer. Past scandals [BBC18; Ril18] have demonstrated that this level of mistrust is appropriate.

Medical Cyber-Physical Systems

Last, and least mature, are medical CPS. These are CPS used for diagnostics or autonomous treatment of patients. As an example, consider an artificial pancreas [Bro19]. This device treats type I diabetes patients by mimicking the behavior of a pancreas, i.e., they release insulin in the blood stream to lower blood glucose levels. A major challenge for these devices is tweaking certain parameters to match each individual patient. Historically, patients themselves had to hand-tune them according to their body's response. Recent innovation adapts them according to measured blood glucose levels. Yet, even in this automated setup, it is unavoidable to subject some patients to inadequate parameter values. As a remedy, deploying an independent monitor can catch a mismatch between parameter values and responses of the individual patients, and thus reduce the number of critical situations. Yet, since artificial pancreata are affixed to the body via adhesive patches, or implanted, the computational resources at their disposal are severely limited. For these reasons, RTLola constitutes an interesting addition. First conceptual work [FSS20; Fin+21] went into developing specification for such artificial pancreata and pacemakers in the context of this thesis, though these endeavors are in their early stages.

2.9. Related Work

The first step in the direction of generating monitors is selecting an input language. Especially in the early days of monitoring, temporal logics such as LTL [Pnu77] or PSL were a popular choice. These logical formulas were then translated into monitoring automata [Dru00; HR02; FS04; Dah+05; RH05; BLS11], e.g. finite state machines or alternating automata. The appeal of these logics is that specifications are concise and declarative, and the languages have only few base operations. This renders formal arguments about them effortless compared to full-fledged programming languages. However, these logics are also limited in terms of expressiveness. In particular, they cannot express real-time constraints, nor do they have language primitives for quantitative properties. While it is possible to represent bounded integer types[28] and their arithmetic operations in LTL, this is cumbersome and leads to a blowup in the size of the specification.

There are a variety of approaches tackling either challenge. For the first one, there are monitoring algorithms for real-time logics, most prominently STL [DFM13b; WS20] and various dialects of MTL [TR05], for example MTL with time series [Dru03], or metric first-order temporal logic [Bas+15; Sch+19]. For the second one, there are two directions. One can enrich the language such as the meta event definition language [Lee+99], which is similar to PSL but with arithmetic capabilities. Alternatively, one can enrich the verdict, lifting it from a ternary/quaternary [BLS07; BLS11] domain to a tridecimal one [Mas+20]. Though, ultimately, the specification language needs to compensate for both shortcomings. Hence, there are quantitative extensions for real-time logics such as STL [DM10; DFM13a; Des+17; ZJP21] or MTL [FP06; FP09; BKT17; Alq+18; Jak+18; CM20].

The resulting languages are sufficiently expressive for use in CPS while still preserving the advantages of logics outlined above, however, the additional expressiveness comes at a heavy price. First, while monitors for LTL were finite state machines and thus inherently space-bounded, the memory requirement of an MTL monitor grows linearly in the number of events even without a quantitative extension. This can be seen when considering the property $p \implies F_{[0,3]} q$. It states that an event q needs to follow within at most 3 second after an event p. This forces the monitor to memorize the timestamps of each p to ensure that a q followed in time. Yet, the number of ps in 3 time units is unbounded, and so is the memory consumption. Second, logics usually have a low number of base operations, which is great for arguing about them. It is also appropriate when expressing relatively short and simple specifications ("Every three seconds, the altitude of the system has to rise"), or a modular series thereof such as a conjunction of several such properties. Nonetheless, when properties grow more complex like when computing sliding window aggregations or nesting properties, formulas become unwieldy and convoluted. As a result, specifiers can no longer grasp their semantics at first glance, and neither can other members of the development team, let alone certification authorities.

[28] This is usually sufficient for CPS.

If these problems render logics inappropriate for the system at hand, there is another solution: languages specifically designed for runtime monitoring. Lola is a pioneer in this category. It is a stream-based specification language for synchronous monitors. As such, it is a monitoring-specific variant of the widely used synchronous programming languages like Lustre [Hal+91; Hal05] or Esterel [BG92]. Yet, while it natively supports arithmetic operations, it lacks a concept of real-time and — as expected from a synchronous language — asynchrony.

Recent work tied Lola into the Haskell ecosystem with HLola [CGS20; GS21]. This paves the way for specifiers to incorporate the Haskell standard library and to easily integrate HLola into existing Haskell-based systems. The different choices in venue for HLola (Haskell) and RTLola (Rust) is indicative of different target audiences. While Haskell is used for industrial purposes, it is widely adopted in teaching and academia as a highly abstract, functional language. In contrast, Rust is tightly related to the world of C and considered a safer alternative for imperative programming in embedded systems, though not yet widely adopted.

Apart from HLola, there are two more extensions for Lola, similar in concept to RTLola: TeSSLa [Con+18a; Leu+18] and Striver [GS18]. The former puts heavy emphasis on the temporal aspect, i.e., it grants specifiers fine-grained control over the timing of the monitor. This level of control even exceeds the one provided in RTLola. In both languages, specifiers can delay a computation by some time. The difference is, in RTLola this value is part of the specification and thus statically determined, whereas in TeSSLa it depends on stream values. Hence, while providing more expressiveness in general, it also allows specifiers to unintentionally create Zeno behavior, i.e., the monitor attempts to process an infinite amount of endogenously created events in a finite amount of time. Since RTLola focuses on safety of the specification, trading off expressiveness for more safety is considered a necessary evil. Another difference between the two languages is that TeSSLa's type system heavily relies on optional values rather than differentiating between synchronous versus asynchronous accesses and event-based versus periodic streams. As a result, when defining TeSSLa functions which access more than one stream, the specifier has to handle the possibilities that only the first, latter, or both streams received a new value. The type system does not assist specifiers when they forget to cover a case or attempt to access a non-extant value. Lastly, TeSSLa has no capability to express dynamic stream creation, nor to access previous values of a stream. Syntactically, it closely resembles function programming languages like Haskell or Scala.

Striver is similar in nature to TeSSLa. It characterizes streams based on their temporal behavior, which the authors call ticks, and their quantitative behavior. These concepts correspond to the timeline of a stream barring spawn and close conditions, and the value type in RTLola, respectively. Striver fixes several shortcomings of TeSSLa by enabling offset accesses and explicitly excluding Zeno behavior. Its syntax is operator-heavy, rendering the desugared syntax concise, yet at times cryptic for laypeople. However, there are standard function declarations and syntactic sugar hiding some of these aspects such that `x(<t, d)`, which desugars into `if (x << t) == outside then d else x(<t)` and expresses an offset operation, i.e., `x.last(or: d)`. With this syntactic sugar, Striver is syntactically similar to Lola and can be categorized between logics and RTLola.

2.9.1. Previous Versions of RTLola

The version of RTLola presented in this work is the product of several revisions.

First, Lola [DAn+05] pioneered synchronous stream-based runtime monitoring. Conceptually, RTLola stayed true to its origins, i.e., a Lola specification is a collection of input streams, output streams, and triggers. However, a Lola expression may refer to the current, past or *future* values of streams. Here, the monitor evaluates expressions as much as possible and kept a store of unresolved expressions. Their resolution depends on future data and is thus delayed until it becomes available. Details on this concept follow in the software compilation in Section 3.2.

→Sec. 3.2, p. 128

After Lola [DAn+05] laid the foundation, it was extended to allow for dynamic stream creation in Lola 2.0 [Fay+16]. Soon after, Faymonville [Fay19] presented the first version of RTLola as a real-time, asynchronous extension of Lola 2.0.

Syntactically, this version closely resembled both Lola and Lola 2.0. Thus, it is more concise than the recent version. Apart from that, there are two notable differences. The syntax for offsets and aggregations was `s[-1, 0]` rather than `s.last(or: 0)` and `s[1h, count, 0]` rather than `s.count(over: 1h)`. Moreover, a parametrized stream has a spawn expression, and extension condition and a termination condition. The first one corresponds to the spawn expression in the current RTLola version, the latter two to the semantic filters of a stream and its termination condition.

Another substantial difference is the underlying model of time. This version of RTLola does not classify *streams* as periodic or event-based but evaluation cycles. Variable-rate computations are triggered when the monitor receives an event. Fixed-rate computations take place periodically according to a globally defined monitor frequency. Every output stream can be affected by both computations, however, sliding windows only get updated in fixed-rate computations to preserve memory bounds.

This timing model has several advantages and drawbacks. Evidently, the specifier has coarser control over the timing of each stream, so they cannot define periodic streams ticking at different static rates. Moreover, all stream accesses constitute 0-order holds, which require a default value and specifiers have to determine by themselves whether the value was guaranteed to be updated in the same evaluation cycle. Yet, this alleviates the need for a complex type system, allowing Faymonville's type system to consist of only value types. Moreover, streams can be computed both in regular intervals *and* after arrival of an event.

Faymonville's RTLola inspired and heavily influenced further work, in particular, my Master's thesis [Sch19a]. This work first revised the syntax to something similar to the desugared syntax, which this thesis will introduce in the next section. Moreover, it introduced the distinction between periodic and event-based streams and presented a suitable type system differentiating between synchronous and asynchronous accesses. Yet, its event-based types are sets of input streams rather than boolean formulas over them, plus it does not feature semantic filters, nor dynamic stream creation. This renders its type system, semantics, and static analyses simpler compared to the RTLola version presented in this thesis.

Chapter 3

Monitor Realizations

The usability of a specification language hinges on proper ways to *realize* and *integrate* monitors into a concrete system. On an abstract level, the choices for the realization boil down to software or hardware solutions. Software solutions require generation — either manual or automatic — of monitor code in a high-level programming language, which is compiled into binary code. This code can then be executed on any suitable general-purpose hardware. The large variety of programming languages and hardware options renders this process quite flexible. Moreover, tools for software development are cheap or even free, which keeps development costs low. On the other hand, hardware solutions require generation of a hardware description for example in VHDL or Verilog. A synthesizer analyzes the description, generates artifacts such as a netlist and configures programmable hardware like a field-programmable gate array (FPGA) or a complex programmable logic device (CPLD) to realize the monitor. This process generally takes much longer than compilation of software code, usually in the realm of minutes to an hour as opposed to seconds. Hence, prior validation of the specification is imperative to keep development time and cost low. The major advantage of hardware solutions is the extraordinary performance of the monitor, in particular in terms of space, time and power consumption. Also, parallelization comes at almost no overhead in hardware solutions. This results in an immense performance boost in terms of throughput provided the monitor can be parallelized. For RTLola, this is the case, partially as a result of MIV: MODULARITY.

Since neither hardware, nor software compilations are universally better, this chapter presents a compilation for either target.

For the hardware realization, Section 3.1 presents a mathematical formulation of a pipelined hardware description for a given RTLola specification. The mathematical formulation is easy to grasp than a hardware description language, yet can easily be translated into one. This was done in a prototype implementation [Bau20], which generates VHDL code and thus enables an empirical evaluation of the process. For this,

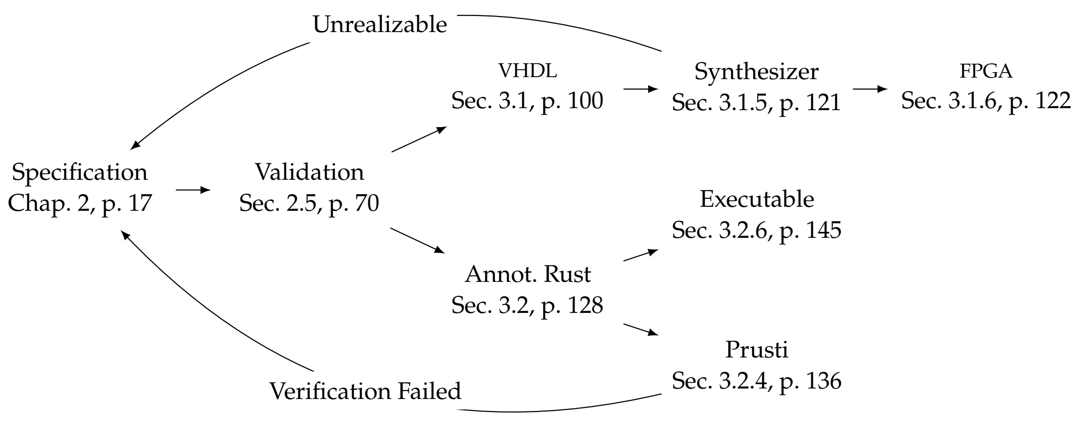


Figure 3.1.: Overview over the workflow from an RTLola specification to an executable monitor via software or hardware realization. Both paths contain validation steps.

an off-the-shelve synthesizer realizes a monitor onto an FPGA. During this process, the synthesizer analyzes the hardware description and provides insights regarding the maximum clock rate, peak and idle power consumption, well as the consumption of logic gates and memory cells. These insights allow specifiers to decide before deployment whether the available resources for the monitor suffice.

The software solution presented in Section 3.2 does not solely aim at generating a monitor realization: it is a proof-of-concept of how to increase trust into the translation process. For this, it translates Lola [DAn+05], a synchronous and discrete variant of RTLola allowing specifiers to refer to future values of streams, into Rust code. Similar to the hardware translation, the software translation allows for generating a monitor evaluating streams in parallel rather than sequentially. Moreover, the translation injects verification annotations into the Rust code to enable translation validation. A third-party tool such as Prusti [Ast+19] can then analyze the code and verify statically that it satisfies the assumptions stated in the annotations. The static verification itself already increases confidence in the correctness of the monitor. Moreover, the chain of trust no longer relies on the translation but on the static verifier. Since the latter is interchangeable, any static verifier commonly used and accredited for the respective application domain can be used, greatly benefitting the certification process.

Figure 3.1 shows the workflow for both solutions. As discussed in the last chapter, after designing a specification, the validation and static analysis provides feedback, which enables informed improvement of the specification. Afterwards, the RTLola compiler generates either VHDL or Rust code. In the former case, the synthesizer realizes the monitor onto an FPGA and produces a resource report. In contrast to the static analysis of the validation, this information is specific to the hardware board at hand. If this report prohibits deployment as planned, specifiers can adapt the specification or system accordingly. On the software side, the Rust code contains verification annotations which will be checked by Prusti. On an abstract level, there are three possible outcomes. First, the verification succeeds. In this case, the code can be compiled into an executable using any Rust compiler. Second, the verification reports a mismatch between specification and monitor code. This indicates a bug in the compilation process, requiring fixes to the compiler, a responsibility that lies outside the hands of the specifier. Third, the verification points at an arithmetic error such as a potential division by zero. This insight allows specifiers to fix the specification such that it properly handles such scenarios and restart the workflow.

Both compilations utilize the domain-specific nature of RTLola to generate heavily optimized code. An empirical evaluation reveals that the compiled software monitors perform significantly better than the interpreter, reducing the running time by over 95%. A similar comparison between hardware and software solutions is hardly possible due to the vast difference in performance of the software platform, a state-of-the-art general purpose computer, versus the hardware platform, a low-resource FPGA. However, the evaluation also showed a significant running time reduction of roughly 91% when comparing a concurrent hardware evaluation versus a sequential one. Naturally, these results are not reproducible to this extend for the software solutions since parallelism in software comes with a substantial overhead and the number of cores is limited. Hence, in this setting, the benefit strongly depends on the details of the specification.

3.1. Hardware Compilation

This section presents an automatic translation of an RTLola specification into a mathematical description of a circuit realizing its semantics. This circuit is organized in a pipeline architecture and closely resembles the semantics of RTLola. For this, it is separated into two logical components connected via a queue. The first component, the *timing manager* (TM), is responsible for handling incoming data and preparing it for the evaluation. Since this process requires very little logic, the TM can be clocked exceedingly fast. This increases the rate in which the monitor can receive input events and hence the level of asynchrony between the system and the monitor. The preparation includes transforming both incoming events and deadlines of periodic streams into a unified schema. As a result, the second component called the *evaluation manager* (EM) can just carry out the evaluation of streams agnostic of the source of the update. The entire monitor is *pipelined*, allowing it to process several data packets in parallel. This becomes particularly relevant in the stream evaluation where all streams in the same evaluation layer are evaluated concurrently.

Timing Manager

Evaluation Manager

This split in responsibility between the TM and EM is rooted in the semantics of RTLola: Definitions 2.20, 2.21 and 2.23 show that all information on *when* an evaluation has to take place and *which* streams are affected can be determined without carrying out any evaluation — barring semantic filters. Definition 2.26 on the other hand show that the EM merely requires the current timestamp for sliding window updates, the input event for non-periodic updates, and information on which streams are potentially active.

→ Defs. 2.20, 2.21, 2.23, p. 63, 64 and 66

→ Def. 2.26, p. 67

After the translation, the last part of this section showcases the efficacy and efficiency of the translation. For this, it presents specifications for monitoring drones and networks. The former demonstrates the impact of computationally heavy monitor tasks whereas the latter quantifies throughput. The results show that the implementation is highly efficient since the monitors fit on small boards, provide high throughput and require less than $2\,W$ of power to operate under peak pressure.

3.1.1. Preliminaries and Notation

First, some hardware-specific concepts and notation needs introduction.

A circuit consists of registers, wires, and gates, which store, transport, and transform information, respectively. The circuit operates based on a system clock χ, which ticks periodically. Each tick activates edge-triggered components and increases the cycle count by one. The time between clock ticks allows signals and values to stabilize.

Wires carry data and connect registers and gates with each other. The terms "wire" and "signal" can mostly be used interchangeably. However, wires represent the physical component whereas signals represent the data carried by wires. A signal s at time x is written as s^x. *Registers* are edge-triggered components which store values until they get updated. They realize a simple update logic in the sense that an n bit register gets an n bit input signal and a 1 bit update signal. On a clock tick, the input will be written into the register provided the update bit is on. Otherwise, the stored value remains unchanged. $\mathbf{R}^x$ denotes the value of register $\mathbf{R}$ in cycle x. Lastly, *gates* realize logic, i.e., they realize a total $\mathbb{B}^n \to \mathbb{B}^m$ function for arbitrary $n, m \in \mathbb{N}$.

The translation algorithm refrains from explicitly defining gates and instead shifts the logic into the definition of register and signal values. Here, owing to the timing model, the definition of a signal at time t may not refer to the value of a register at time t since this value is not yet stored in it. Instead, it can only refer to the value of the register at $t-1$. However, within a pipeline stage, the definition may refer to the *input* of the register at time t. This input will be the value of the register at time $t+1$. As a result, it circumvents the delay of one cycle. Notationally, the definition then refers to $\mathbf{R}.\mathrm{in}$ rather than just $\mathbf{R}$.

The notation for bit strings overlaps with the notation for vectors. Hence, for a bit string $\vec{x}$ of width n, $\vec{x}\,[i]$ denotes the i^{th} value of $\vec{x}$ for $i \leqslant n$. Given two bit strings $\vec{x} \in \mathbb{B}^n$ and $\vec{y} \in \mathbb{B}^m$, the $\circ$ operator denotes the bit concatenation, so $\vec{x} \circ \vec{y}$ is an $n+m$ bits wide string. Moreover, $\vec{x}\,[i \ldots j]$ is the substring $\vec{x}\,[i] \circ \vec{x}\,[i+1] \circ \ldots \circ \vec{x}\,[j]$ provided $j \geqslant i$. Last, b^n denotes an n-fold repetition of the single bit $b \in \mathbb{B}$.

3.1.2. Structure

The hardware realization has two entry points for information and one outlet. The entry points are a clock providing the current time and an interface to the system for reception of new events. These entry points correspond to the split between periodic and event-based streams. This information is unified in the Timing Manager (TM) and flows to the Evaluation Manager (EM). This potentially triggers the dissemination of a warning to the system. Figure 3.2 illustrates this setup.

The TM receives event data over a $\sum_{\sigma^{\downarrow} \in \mathcal{S}^{\downarrow}} (\mathit{bits}(\sigma^{\downarrow}) + 1)$ bit wide wire. Here, $\mathit{bits}(\sigma^{\downarrow})$ is the bit size of a single value of stream $\sigma^{\downarrow}$. Hence, the wire can transport a single value of every input stream at once plus a bit per stream to indicate whether a new value is present or not. This is necessary to distinguish an input value of $0^{\mathit{bits}(\sigma^{\downarrow})}$ from a non-extant value. Note that in the following, the limits of the sum are omitted when describing data lines carrying events. The second input comes from the clock and is $|\mathit{ts}|$ wide, i.e., the number of bits required to store a single timestamp.

The connection between TM and EM consists of a buffered update channel and an unbuffered feedback channel. Here, the buffer is a simple first in – first out queue. Interfacing with such a queue requires two input signals: the data q_{in}, and a 1-bit push signal. A data packet amounts to the space required to transfer a single event with an additional bit per output stream. This bit indicates whether the respective stream is active at the point in time of the event. This information suffices for specifying both updates due to events and deadlines, hence the message format is universal. The feedback channel consists of registers rather than a dedicated queue. Its information consists of $(1 + |\mathit{ts}|)\,\mathrm{dom}(\sigma^{\uparrow})$ bits per parametrized output stream $\sigma^{\uparrow}$. Here, $\mathrm{dom}(\sigma^{\uparrow})$ denotes the number of instances permitted for the stream. This can either be the domain of the

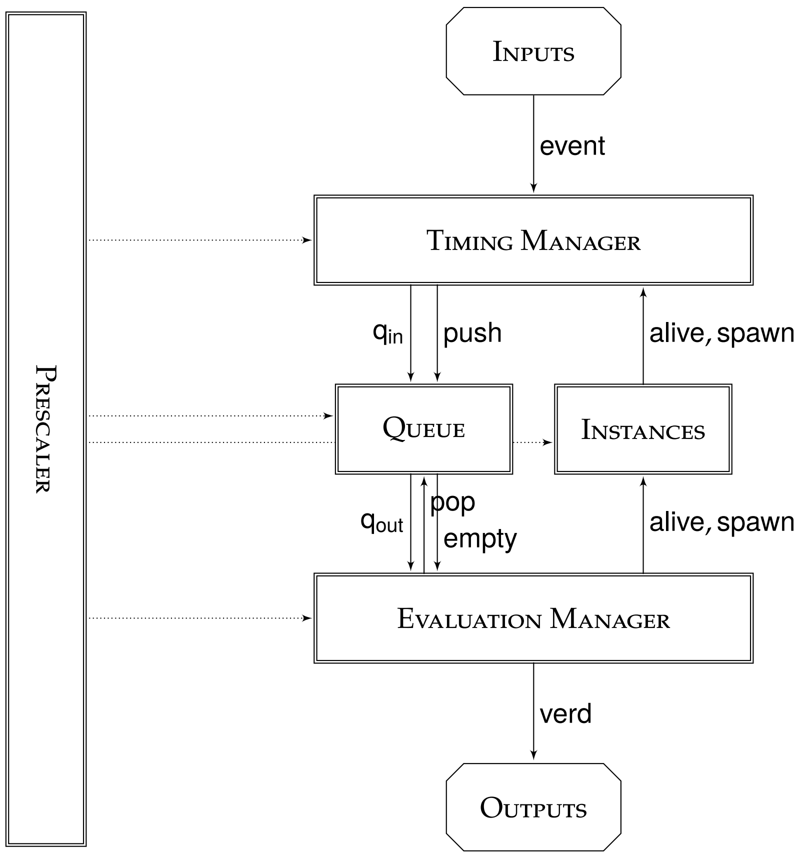


Figure 3.2.: Schematic of an RTLola monitor comprising two modules with access to an instance store and connected via a queue. The *Timing Manager* manages the order in which periodic and event-based streams have to be evaluated. The *Evaluation Manager* controls the evaluation process of all affected streams. Limits of sums are omitted for a cleaner display.

parameter type or a specific, lower value. The channel allows the EM to inform the TM about which instance of a stream are currently in existence and when they were created. This is necessary because each instance induces a timeline and thereby deadlines.

Lastly, the EM pops data out of the queue and manages the evaluation process. To this end, its interface to the queue consists of a 1 bit pop signal controlled by the EM, plus the 1 bit empty signal and the q_{out} signal controlled by the queue. They respectively convey the information whether data is present and — one cycle after popping — the information itself. Internally, the EM is a state machine which idles until reception of information from the TM. It then first updates the memory of input streams and iteratively evaluates output streams layer by layer. At the end, it writes the verdict in an output register and returns to its idle state.

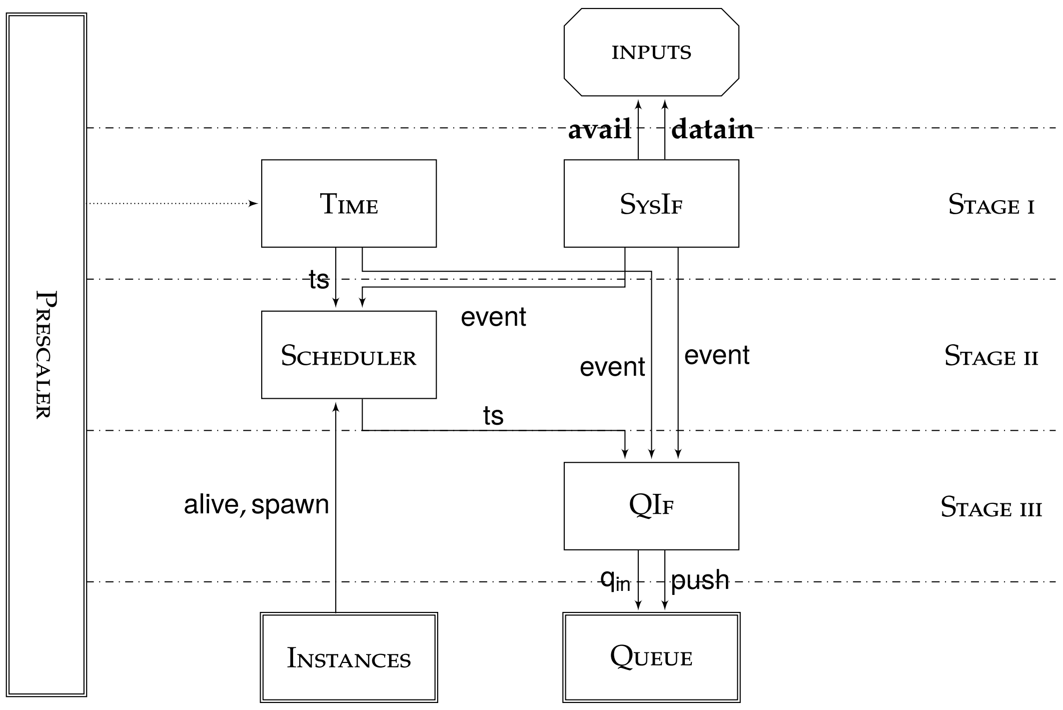


Figure 3.3.: Schematic of the Timing Manager receiving external events, managing periodic deadlines, and preparing data for the Low-level Controller.

Remark 3.1 (Dual Purpose of the Queue)**.** Due to the difference in complexity of the TM and EM, the former can process information significantly faster than the latter. This enables the TM to be clocked at a rapid pace, which in turn allows it to cope with bursts of data. These bursts can be a result of a sudden spike in events or several deadlines almost coinciding. In such cases, the queue acts as a buffer between the TM and EM, temporarily relieving it of some stress while preventing loss of data to some extent. Moreover, the queue allows for decoupling the frequency of the TM and EM entirely. To this end, the clock of the queue has to be compatible to both components. Due to the minimal logic engraved in the queue, this constraint is easy to satisfy.

3.1.3. Timing Manager

In a nutshell, the timing manager receives external events and manages periodic evaluations. Its schematic, outlined in Figure 3.3, reflects this dual responsibility. The left half relates to deadlines. Information stems from the prescaler, which provides a common clock for all components. Most importantly, it allows the Time-component to keep track of the current system time by counting the number of ticks and multiplying it by the static period of ζ_{tm}. The current time flows to the scheduler, enabling it to decide which deadlines are due. This decision depends on a static and a dynamic schedule.

On the event-based side of the timing manager, the component named System Interface (SysIf) manages communication with the system via two registers. The system writes event data into the **din** register and sets the **avail**-bit. Upon detection of new data, SysIf copies and resets the **avail**-bit, signaling readiness to accept further events. This can happen in each clock tick of the TM even though the information of the previous event has not reached the queue, yet, thanks to the pipelined architecture. Information regarding the event then flows to the *Queue Interface* (QIf). This component unifies three sources of information: the event data, the timestamp in the moment of event reception, and the affected stream instances. Since the sources propagate their information immediately, a delay mechanism is required to align them temporally, as is usual when working with pipeline stages.

The following describes each component in details and provides a mathematical description.

Remark 3.2 (Simplification for Clarity)**.** Figure 3.3 outlines the schematic of the TM. Evidently, it omits some information: the prescaler provides a clock signal for all components of the TM rather than just the Time-component. Generally, all further illustrations omit clock lines unless to emphasize particularities. Moreover, signals require a mechanism to detect whether it carries meaningful data or is "empty". For this, every data line d between two components has an implicit 1-bit valid_d signal indicating presence of data. Illustrations generally omit this detail while mathematical definitions take it into account.

Prescaler

This component scales the system clock ζ_{sys} down by a constant factor to two TM-internal clocks ζ_{tm} and ζ_{2tm}. The former is the main clock of the TM, so it drives the majority of components. QIf is the mere exception as it is clocked by a combination of ζ_{tm} and ζ_{2tm} ticking twice as fast. The reason behind this becomes apparent when discussing the details of QIf.

System Interface

Inputs: **avail** $\in \mathbb{B}$, **datain** $\in \mathbb{B}^{\sum(bits(\sigma^{\downarrow})+1)}$,
Outputs: event $\in \mathbb{B}^{\sum(bits(\sigma^{\downarrow})+1)}$, valid_event $\in \mathbb{B}$

This component handles the communication with the system. For this, it waits on the **avail** bit. If on, it copies the event data over to the event signal and activates the validation bit. Otherwise, the signal is meaningless, hence its value is DC ("don't care"). Here, DC means the actual value is never used in any computation, so any concrete realization for this value is fine. Since processing an event only takes a single cycle, the component always clears the **avail** register and the valid_event bit mirrors **avail**.

$$\text{event}^t = \begin{cases} 0^{\sum(bits(\sigma^{\downarrow})+1)} & \text{if } t = 0 \\ \mathbf{datain}^{t-1} & \text{if } \mathbf{avail}^{t-1} \\ \textsc{dc} & \text{otherwise} \end{cases}$$

$$\mathbf{avail}^t = 0$$

$$\text{valid\_event}^t = \begin{cases} 0 & \text{if } t = 0 \\ \mathbf{avail}^{t-1} & \text{otherwise} \end{cases}$$

Time

Outputs: ts $\in \mathbb{B}^{|ts|}$

Internal Registers: **ts** $\in \mathbb{B}^{|ts|}$

The components waits on the system clock and updates its internal timestamp register **ts** by repeatedly adding the statically determined clock period. The input of the register is also the output of the component.

$$\mathbf{ts}^t = \begin{cases} 0^{|ts|} & \text{if } t = 0 \\ \mathbf{ts}^{t-1} + \zeta_{\text{tm}} & \text{otherwise} \end{cases}$$

$$\text{ts}^t = \mathbf{ts}^t.\text{in}$$

Remark 3.3 (Alternative Realization)**.** For this component, there is a functionally equivalent realization in which it counts the number of clock ticks rather than summing the periods up. The current time is then the tick count multiplied by the clock period. Neither option is definitively better: multiplication is generally expensive in terms of logic gates and delay, however, it does require less memory. Ultimately, the decision is case-specific.

Scheduler

Inputs: ts $\in \mathbb{B}^{|ts|}$, event $\in \mathbb{B}^{\sum(bits(\sigma^{\downarrow})+1)}$

Outputs: due $\in \mathbb{B}^{n^{\uparrow}_{\text{static}}+n_{\pi}+n^{\uparrow}_{\text{ev}}}$

For a given point in time and event it has to decide which stream instances need to be evaluated. This requires three distinct kinds of information. First, it needs the current time, which the Time component provides. Second, the points of evaluation of dynamically created streams depends on when they were spawned. Hence, the Scheduler accesses information provided by the Instances component regarding which stream instances are active and when they became so. Last, it needs to take the event-based and periodic types of streams into account. Since this information is static for a given specification, it is hard-wired into the circuitry.

For periodic streams, this is in the shape of lookup tables. These tables are essentially two arrays of static memory cells: $\mathbf{staticsched}^{\infty}$ and $\mathbf{affected}^{\infty}$. The former is a static schedule covering one lookup table, i.e., it contains timestamps at which at least one static stream needs to be evaluated. Each timestamp is relative to the hyperperiod Π of all static streams. The latter array contains the IDs of streams which are affected by a deadline in unary encoding. This means, at time t with $t = \mathbf{staticsched}^{\infty}[i] \mod \Pi$, the k^{th} static output stream needs to be evaluated if $\mathbf{affected}^{\infty}[i][k]$ is true.

In the following, let $\pi_{\sigma^{\uparrow}}$ be the periodic type of $\sigma^{\uparrow}$. For event-based streams, let $\varepsilon_{\sigma}^{\uparrow}$ be the event-based type of $\sigma^{\uparrow}$. Since an event-based type is a positive boolean formula over input streams, it naturally translates into a simple, state-less circuit. The input for this formula is the information which input streams received a new value. Moreover, let n_{dl} be the number of static deadlines, i.e., the number of entries of $\mathbf{staticsched}^{\infty}$, and let $n_{\text{static}}^{\uparrow}$ denote the number of static streams, i.e., output streams with spawn timeline *Always* and close timeline *Never*. Further, let $n_{\text{ev}}^{\uparrow}$ be the number of output streams with an event-based type. Last, let n_{π} and n_{ε} denote the total number of potential instances of periodic streams, and event-based streams, respectively.

Static Periodic Streams

Static: $\mathbf{staticsched}^{\infty} \in \mathbb{B}^{|ts| \times n_{\text{dl}}}$, $\mathbf{affected}^{\infty} \in \mathbb{B}^{n_{\text{static}}^{\uparrow} \times n_{\text{dl}}}$

Inputs: $\mathsf{ts} \in \mathbb{B}^{|ts|}$

Outputs: $\mathsf{staticdue} \in \mathbb{B}^{n_{\text{static}}^{\uparrow}}$

Internal Registers: $\mathbf{poteffts} \in \mathbb{B}^{|ts|}$, $\mathbf{activedl} \in \mathbb{B}^{n_{\text{dl}}}$

Internal Signals: $\mathsf{poteffts} \in \mathbb{B}^{|ts|}$, $\mathsf{effts} \in \mathbb{B}^{|ts|}$, $\mathsf{potdue} \in \mathbb{B}^{n_{\text{dl}}}$

The construction starts by determining the statically due streams. It first aligns the time representation of ts, which is an absolute time, and the representation in $\mathbf{staticsched}^{\infty}$, which is relative to the last passage of the hyperperiod. To this end, it keeps track of an **offset** that is the absolute timestamp of the beginning of the current hyperperiod. It then subtracts the offset from the current time to obtain the *potentially effective timestamp* (poteffts). This value can be greater than the hyperperiod, in which case the Scheduler increases the offset for the next cycle. In this case it also subtracts an additional hyperperiod from the poteffts to obtain the *actual* effective timestamp (effts).

$$\mathbf{offset}^t = \begin{cases} 0 & \text{if } t = 0 \\ \mathbf{offset}^{t-1} & \text{if } \mathsf{poteffts}^t \leqslant \Pi \\ \mathbf{offset}^{t-1} + \Pi & \text{otherwise} \end{cases}$$

$$\mathsf{poteffts}^t = \mathsf{ts}^t - \mathbf{offset}^{t-1}$$

$$\mathsf{effts}^t = \begin{cases} \mathsf{poteffts}^t - \Pi & \text{if } \mathsf{poteffts}^t > \Pi \\ \mathsf{poteffts}^t & \text{otherwise} \end{cases}$$

For the next step, the scheduler utilizes the inherently parallel nature of hardware by concurrently comparing each deadline in **staticsched**$^\infty$ against the effective timestamp. The result is a bit array of potentially due (potdue) deadlines starting with a prefix of 1s, followed by 0s. The currently active deadline (activedl) is now the right-most 1 provided it has not been active in the clock cycle before as well. This check requires the scheduler to persist the last active deadline in a separate register.

$$\mathsf{potdue}^t\,[i] \equiv \mathbf{staticsched}^\infty\,[i] \leqslant \mathsf{effts}^t$$

$$\mathsf{activedl}^t\,[i] = \begin{cases} 0^{n_{\mathrm{dl}}} & \text{if } t = 0 \\ \mathsf{potdue}^t\,[i] \wedge \neg \mathsf{potdue}^t\,[i+1] \wedge \neg \mathbf{lastactive}^{t-1}\,[i] & \text{if } i < n^{\uparrow}_{\mathrm{static}} \\ \mathsf{potdue}^t\,[i] \wedge \neg \mathbf{lastactive}^{t-1}\,[i] & \text{if } i = n^{\uparrow}_{\mathrm{static}} \end{cases}$$

$$\mathbf{lastactive}^t\,[i] = \begin{cases} \mathbf{lastactive}^{t-1}\,[i] & \text{if } \mathsf{activedl}^t = \mathbf{lastactive}^{t-1} \\ \mathsf{potdue}^t\,[i] \wedge \neg \mathsf{potdue}^t\,[i+1] & \text{if } i < n^{\uparrow}_{\mathrm{static}} \\ \mathsf{potdue}^t\,[i] & \text{if } i = n^{\uparrow}_{\mathrm{static}} \end{cases}$$

Clearly, activedl is a bit string with at most one active bit. The SCHEDULER transforms this information into a unary encoding of active static streams where each bit represents a static stream. If a bit is on, the respective stream needs to be evaluated. This transformation again exploits parallel computation of $\mathbb{B}^1 \times \mathbb{B}^n \to \mathbb{B}^n$ conjunctions.

$$\mathsf{staticdue}^t = \bigvee_{i=1}^{n_{\mathrm{dl}}} \mathbf{affected}^\infty\,[i] \wedge \mathsf{activedl}^t\,[i]$$

The crux in the design of the static side is to utilize static information as much as possible. This entails pre-computing the full static schedule such that during runtime, the component only needs to identify where it is operating within the hyperperiod. With this in place, the only costly operation that needs to be duplicated for each deadline are comparisons, yet they are still relatively cheap component.

Dynamic Periodic Streams

Inputs: ts $\in \mathbb{B}^{|ts|}$, alive $\in \mathbb{B}^{n_\pi}$, spawn $\in \mathbb{B}^{|ts| \times n_\pi}$

Outputs: dyndue $\in \mathbb{B}^{n_\pi}$

Internal Registers: **lastalive** $\in \mathbb{B}^{n_\pi}$, **lastdue** $\in \mathbb{B}^{|ts| \times n_\pi}$

Internal Signals: nextdue $\in \mathbb{B}^{|ts| \times n_\pi}$

Computing the dynamic schedule is significantly more costly for two reasons: there is less static information since dynamic streams have no fixed spawn time, and these spawn times are neither necessarily aligned for different streams, nor for instances of the same stream. As a result, the component needs to inspect each stream instance separately to determine whether they are due.[1] This requires duplication of the determination logic for each potential stream instance. Hence, instantiation incurs enormous costs and should thus be used sparsely for a hardware translation, e.g. by parametrizing only over data types with small domains such that the number of potential instances n_π is low.

The logic itself uses the current timestamp as well as information from the Instances component, which is an alive bit for each instance and its spawn timestamp. Based on this information, the Scheduler keeps track of the last due time of each instance in the **lastdue** registers. Adding the period of the stream yields the nextdue signal. If this value is less than or equal to the current timestamp, the respective instance is due. Formally, the following describes the dynamic scheduling logic for a dynamic stream $\sigma^\uparrow$ and possible parameters p:

$$\text{nextdue}^t\left[\sigma^\uparrow, p\right] = \textbf{lastdue}^{t-1}\left[\sigma^\uparrow, p\right] + \pi_{\sigma^\uparrow}$$

$$\textbf{lastalive}^t\left[\sigma^\uparrow, p\right] = \begin{cases} 0 & \text{if } t = 0 \\ \text{alive}^t\left[\sigma^\uparrow, p\right] & \text{otherwise} \end{cases}$$

$$\textbf{lastdue}^t\left[\sigma^\uparrow, p\right] = \begin{cases} 0^{|ts|} & \text{if } t = 0 \\ \text{spawn}^t\left[\sigma^\uparrow, p\right] & \text{if alive}^t\left[\sigma^\uparrow, p\right] \wedge \neg\textbf{lastalive}^{t-1}\left[\sigma^\uparrow, p\right] \\ \text{nextdue}^t\left[\sigma^\uparrow, p\right] & \text{if dyndue}^t\left[\sigma^\uparrow, p\right] \\ \textbf{lastdue}^{t-1}\left[\sigma^\uparrow, p\right] & \text{otherwise} \end{cases}$$

[1]This can be improved by grouping streams with identical spawn and close conditions.

$$\mathsf{dyndue}^t\left[\sigma^\uparrow, p\right] \equiv \mathsf{nextdue}^t\left[\sigma^\uparrow, p\right] \leqslant \mathsf{ts}^t \wedge \mathsf{alive}^t\left[\sigma^\uparrow, p\right]$$

Note that access operations such as $\mathsf{nextdue}^t\left[\sigma^\uparrow, p\right]$ denote access into a memory matrix. In a concrete realization, the matrix would be linearized and the indices for the accesses statically pre-computed.

While the logic looks fairly simple, its costliness stems from the addition and comparison in the definition of nextdue and ddue. These adders and comparators need to be present for each *potential* stream instance. Since even small domains allow for representing a large amount of values, the cost blows up quick. If a specification only contains a single stream parametrized by a `UInt8`, the Scheduler needs an additional 256 adders for values of width $|ts|$ bits. A remedy is domain specific value types, for example, a system might only be capable of observing up to 20 different external objects. Hence, respective streams should be parametrized over a domain with 20 entries rather than `UInt8`.

Event-Based Streams

Inputs: event $\in \mathbb{B}^{\sum(bits(\sigma^\downarrow)+1)}$

Outputs: eventdue $\in \mathbb{B}^{n^\uparrow_{\text{ev}}}$

For event-based streams, the Scheduler first computes whether a stream can be affected by an event based on its type $\varepsilon^\uparrow_\sigma$. For this, it transforms the type into its natural circuit representation. It then extracts the bits indicating whether an input received an update and connects them to the respective inputs of the circuit. The result of the circuit is then an indicator whether an event-based stream is affected by an event and thus due. Note that if the event is empty, due to lack of negation in event-based types, all bits of evaff will be off.

$$\mathsf{eventdue}^t\left[\sigma^\uparrow\right] = \mathsf{event}^t \models \varepsilon^\uparrow_\sigma$$

Here, the alive bit from the InstanceStore is irrelevant since the alive status can change based on data currently processed in the EM. Hence, this consideration is part of the EM.

Composition

Inputs: staticdue $\in \mathbb{B}^{n^\uparrow_{\text{static}}}$, dyndue $\in \mathbb{B}^{n_\pi}$, eventdue $\in \mathbb{B}^{n^\uparrow_{\text{ev}}}$

Outputs: due $\in \mathbb{B}^{n^\uparrow_{\text{static}}+n_\pi+n^\uparrow_{\text{ev}}}$

Lastly, the Scheduler composes the sub-results into a single bit array.

$$\mathsf{due}^t = \mathsf{staticdue}^t \circ \mathsf{dyndue}^t \circ \mathsf{eventdue}^t \tag{3.1}$$

Note that the eventdue and by proxy the due signal does not consider the alive status for event-based streams. This is because this status can change based on data currently processed in the EM, rendering eventdue void. Hence, this consideration is part of the EM instead and the TM only relays information which event-based *streams* rather than instances are affected. This can only happen for event-based streams since periodic ones always have to wait until their period has passed at least once. Here, it is a reasonable assumption on the hardware monitor and specification that the period of every stream is greater than the time the monitor needs for a full evaluation of the queue, rendering the same treatment for periodic streams unnecessary.

Queue Interface

Inputs: due $\in \mathbb{B}^{n^{\uparrow}_{\text{static}}+n_\pi+n^{\uparrow}_{\text{ev}}}$, event $\in \mathbb{B}^{\sum(bits(\sigma^{\downarrow})+1)}$, ts $\in \mathbb{B}^{|ts|}$

Outputs: q_in $\in \mathbb{B}^{n^{\uparrow}_{\text{static}}+n_\pi+n^{\uparrow}_{\text{ev}}+|ts|+\sum(bits(\sigma^{\downarrow})+1)}$, push $\in \mathbb{B}$

Internal Registers: **event** $\in \mathbb{B}^{\sum(bits(\sigma^{\downarrow})+1)}$, **ts** $\in \mathbb{B}^{|ts|}$

This component accepts data from three sources, aligns them temporally[2], composes them, and commits them to the queue. For the alignment, the QIf stores both the current event and the respective internal timestamp into a register for precisely one cycle. At the same time, it concatenates the outputs of the registers, which is the values of the previous cycle, plus the current input sent from the Scheduler. The reason for this is that the pipelined architecture results in a one-cycle mismatch between values from the first stage, i.e., Time and SysIf, and the second stage, i.e., Scheduler. As a result, the short delay introduced by the registers suffices for alignment. A push into the queue takes place if either there was an event or at least one periodic stream requires an update.

$$\mathbf{event}^t = \begin{cases} 0^{|ts|} & \text{if } t = 0 \\ \mathsf{event}^t & \text{otherwise} \end{cases}$$

$$\mathbf{ts}^t = \begin{cases} 0^{|ts|} & \text{if } t = 0 \\ \mathsf{ts}^t & \text{otherwise} \end{cases}$$

$$\mathsf{push}^t = \mathsf{valid\_event}^t \vee \bigwedge_{i=1}^{n^{\uparrow}_{\text{static}}+n_\pi} \mathsf{due}[i]$$

$$\mathsf{q\_in}^t = \mathsf{due}^t \circ \mathbf{ts}^{t-1} \circ \mathbf{event}^{t-1}$$

[2]This alignment is a standard procedure for pipelining and often left implicit; here, it is spelled out explicitly.

Note that the event bits of q_in potentially carry meaningless data. In this case, the respective valid-bits for each input are 0, though. Hence, the EM will not update the streams, ignoring the values.

3.1.4. Evaluation Manager

The evaluation manager essentially transforms data from the TM into verdicts. For this, it pops data off the queue and starts an evaluation process based on it. Along the way, it persists updates in the INSTANCES store. Upon completion of the evaluation, it potentially emits trigger warnings to the system. The entire process is clocked by the EM specific clock ζ_{em}.

Figure 3.4 shows the schematic of the EM. The queue interface (QIF) is responsible for the communication with the queue. However, the main logic lies within the control component (CTRL). It indirectly communicates with the queue, and directly updates the instance store. Most notably, it manages the evaluation process of all streams. This evaluation is pipelined much like the TM. Here, each stage represents an evaluation layer of the specification. Each layer contains a set of streams and windows. Stream nodes consist of logic realizing the stream expression, and storage containing the most recent values of the stream. Similarly, window nodes contain logic computing the sliding window aggregation, and memory storing the intermediate values. Pushing window handling in separate nodes allows for better utilization of the pipeline architecture and parallel evaluation of nodes. Suppose a stream σ in layer 3 aggregates values of a stream σ' in layer 1. A realization without the split places the logic of the aggregation and the stream evaluation in the same stage. However, the stream evaluation needs to wait until the aggregation finished. With the split, the window aggregation takes place in the second stage, significantly reducing the depth of the third stage.

The pipeline architecture enables the EM to start processing a new event before the preceding one was completed. However, this requires non-consecutive layers to be independent. In RTLola, this is not necessarily the case as a stream $\sigma^{\uparrow}$ can depend on one or several lower layers. Hence, these lower layers may not be overwritten by another evaluation until $\sigma^{\uparrow}$ was evaluated. One way to solve this problem is by storing additional values and shifting the access index to account for the delay of the access. The necessary logic requires information from multiple components over several consecutive evaluation steps. As a result, the wiring becomes intricate and correctness is hard to assess. For this reason, the EM implements a simpler stalling mechanism manifesting the dependencies between different streams.[3] This mechanism is *coarse*, i.e., a stream stalls the entire evaluation layer of the stream on which it depends to reduce complexity.

[3]The index shift solution is part of the software compilation as neither pipelining nor wiring are a concern there.

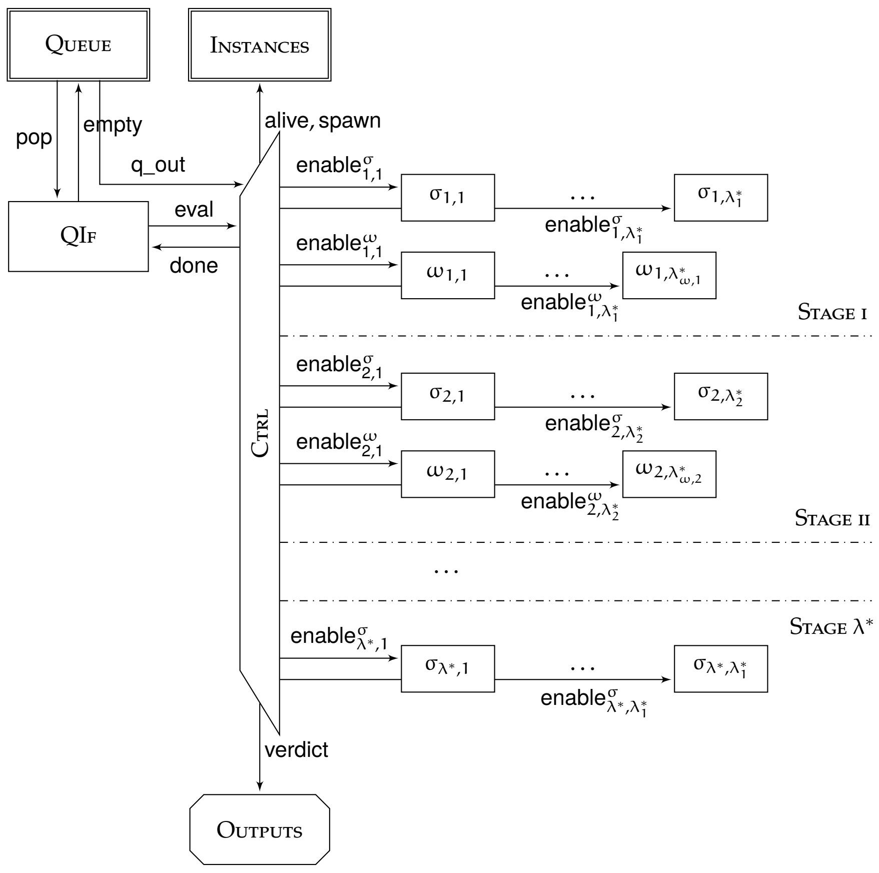


Figure 3.4.: Schematic of the Evaluation Manager. It consists of a QIf responsible for the communication with the Queue and the Evaluator, which manages internal memory, evaluates stream expressions, and updates the Instances store.

Lastly, Ctrl collects information regarding newly spawned and closed stream instances as well as triggers. The former information updates the Instances store, the latter is forwarded to the system.

Note that the schematic lacks some wires for a less cluttered presentation. In particular, it lacks feedback channels from each stream to the control component, and wires from streams to windows and other streams.

The following example provides more intuition on how stalling works in this architecture before proceeding with the formal details.

Example 3.1 (Stalling I)**.** Consider a specification with three streams where the second stream depends on the first and the third stream depends on both the first and second. Its dependency graph is depicted in Figure 3.5. There are three layers and thus three pipeline stages. The right-hand side of the figure shows how data flows through the stages over time. In this process, stalling ensures that required data is always present.

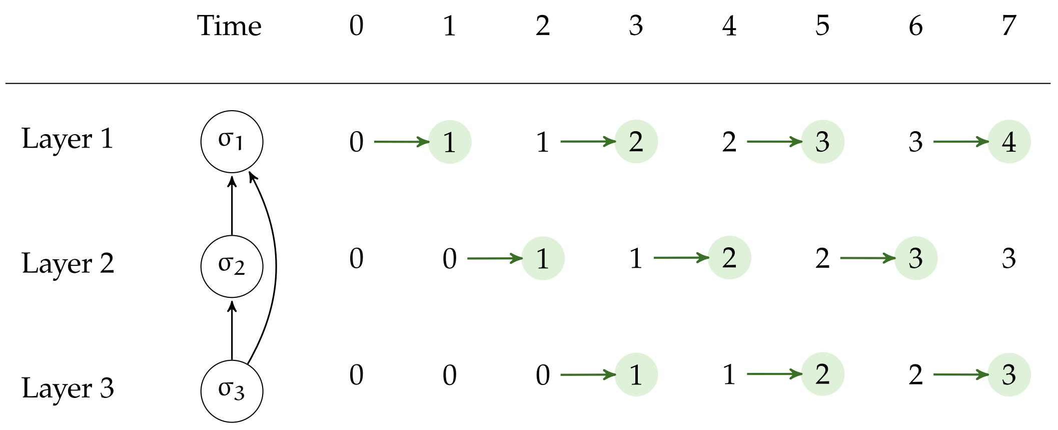


Figure 3.5.: Dependency graph of a specification with three streams. Numbers on the right show how events are propagated through the pipeline stages over time. Green markings indicate which layers were updated.

When transitioning from time 1 to 2, for example, layer 1 already processed event 0 whereas layer 2 is just in the process of doing so. Since layer 3 depends on both preceding layers, it cannot process event 0, yet. Moreover, this causes layer 1 to be stalled to prevent it from overwriting data layer 3 still needs.

In the next step, i.e., from time 2 to 3, layer 3 can proceed. Since this process dissipates the stall, layer 1 can process event 1 in parallel. Only layer 2 has to idle because its next computation depends on the output of layer 1 for event 1, which is not ready, yet.

Without the dependency from layer 3 to layer 1, the pipelined architecture allows for persistently clocking every stage. Thus, every clock cycle, another event would be processed to completion. △

Queue Interface

The queue interface (QIf) handles the communication between the EM and the Queue. Internally, it is a simple three-state automaton depicted in Figure 3.6. The *idle* state constitutes the starting point of the evaluation. Whenever the queue contains at least one entry, the QIf transitions into the *pop* state. Here, the component simultaneously sends a pop signal to the Queue and an eval signal to the Ctrl. This prompts the queue to send its latest entry via the q_{out} signal directly to the controller. At the same time, the eval signal serves as enable-bit for a register in the Ctrl component, so it automatically receives and persists the output of the queue.

The QIf unconditionally leaves the *pop* state after one cycle in favor of the *eval* state. In the process, it lowers the pop signal. Here, it resides until it receives the done signal from the Ctrl indicating that it can receive another input. In this case, the QIf transitions

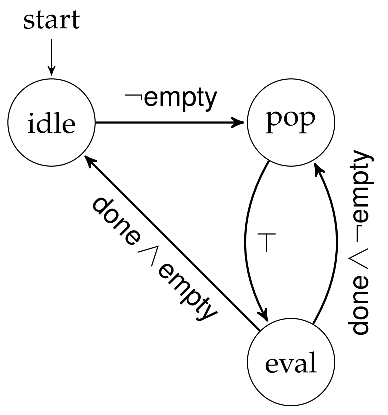


Figure 3.6.: The LLQInterface handles the communication with the queue in *pop*, and waits in *eval* until the evaluation finished.

either back into the *pop* state, repeating the process, or in the *idle* state if the queue is empty. This automaton can trivially be translated into a hardware description.

Control

The Ctrl component is the heart and soul of the EM. Its task is to manage stream instances and start the evaluation of active, non-stalled streams.

Stalling Intuitively, a layer is stalled if an evaluation would override data that another layer still needs. To this end, the stalling mechanism keeps track of which data is currently in each stage via a **count** register. This register is initially 0 and increases whenever the layer is enabled.

$$\mathbf{count}_\lambda^t \equiv \mathbf{count}_\lambda^{t-1} + \mathsf{enable}_\lambda^t$$

Note that the enable signal, the stall signal, and the count register are mutually dependent. This can be resolved since the stall signal only uses the enable signal of greater stages and the count in the last time step. The intuitive difference between the stall and the enable signal is that the former indicates there is an external reason prohibiting a stage to progress. In contrast, the enable signal is only set if there is neither an external reason preventing the execution, nor an internal one such as a lack of data the layer depends on. Hence, if the enable signal is set, the evaluation will definitely take place whereas when there is no stall, the evaluation is still not guaranteed.

With this intuition and the count register, the following definition states three criteria for when a layer $\lambda' > \lambda$ stalls the layer λ. These signals are computed successively, starting with the greatest layer propagating the results downwards.

$$\mathsf{stall}^{t}_{\lambda'\to\lambda} \equiv \Big(\bigvee_{\eta}\big(\underbrace{\mathrm{dep}(\sigma_{\lambda',\eta},\lambda)}_{\text{static}} \wedge \bigvee_{p}\underbrace{\mathsf{tmactive}^{t}(\sigma^{p}_{\lambda',\eta})}_{\text{dynamic}}\big)\Big) \qquad \text{(Dependency)}$$

$$\wedge\neg(\mathsf{enable}^{t}_{\lambda'} \wedge \mathbf{count}^{t-1}_{\lambda} = \mathbf{count}^{t-1}_{\lambda'} + 1) \qquad \text{(Parallel Computation)}$$

$$\wedge\,\mathbf{count}^{t-1}_{\lambda} \neq \mathbf{count}^{t-1}_{\lambda'} \qquad \text{(Recovery)}$$

Here, the notation $\sigma_{\lambda,\eta}$, refers to the η^{th} stream in layer λ.

First, stalling only becomes a possibility if there is a dependency relation between the two layers. This is the case if there is a stream instance in λ' that accesses a stream in λ and is active. For this, the static check $\mathrm{dep}(\sigma_{\lambda',\eta},\lambda)$ is true iff there is an edge in the dependency graph from stream $\sigma_{\lambda',\eta}$ to a stream $\sigma_{\lambda,\eta'}$ in layer λ. The dynamic check refers to whether $\sigma_{\lambda',\eta}$ is active according to all non-semantic criteria. For this, recall that $\mathsf{q}_{\mathsf{out}}$ comprises the timestamp, the event data, and indicators which periodic stream instances and event-based streams are affected, defined in Equation (3.1). Thanks to this, the check $\mathsf{tmactive}^{t}(\sigma^{p}_{\lambda',\eta})$ simply accesses the respective bit for stream $\sigma_{\lambda',\eta}$ from the q_out signal for periodic streams. For event-based streams, the check accesses the respective bits from the eventdue portion of q_out and conjoining it with the alive bit of the instance. → Eq. (3.1), p. 109

To have access to the q_out signal, each stage additionally contains a register **curr_ev** persisting and propagating it through the stages. The information is then piped forward whenever a layer is enabled.

$$\mathbf{curr\_ev}^{t}_{\lambda} = \begin{cases} \mathsf{q\_out}^{t} & \text{if } \lambda = 1 \wedge \mathsf{eval}^{t} \\ \mathbf{curr\_ev}^{t-1}_{\lambda-1} & \text{if } \lambda > 1 \wedge \mathsf{enable}^{t}_{\lambda} \\ \mathbf{curr\_ev}^{t-1}_{\lambda} & \text{otherwise} \end{cases} \tag{3.2}$$

Second, stalling is unnecessary despite active dependencies if λ' is enabled and its computation requires exactly the results stored in the current layer. Note that the enable signal for a layer is only set if all its dependencies are satisfied. Yet, this check in the definition of stall is necessary since stage λ' might be enabled for an older computation, i.e., $\mathbf{count}_{\lambda'}$ might be $\mathbf{count}_{\lambda} - 2$. Hence, if λ' is enabled and the counts are off by exactly one, then λ does not need to be stalled as the computations of both layers happen in parallel.

Third, λ' does not stall λ if their count is equal. Intuitively, this is the case if λ' was stalled until λ caught up. In this case, λ can continue working despite active dependencies from λ' to λ since these dependencies refer to yet-to-be-computed values. Here, λ' will not become active until after these values were computed as per definition of the enable signal.

Finally, a stage is stalled if any other stage causes the stall as described above or there is a congestion, i.e., the stage immediately above is neither enabled nor empty. The stall signal is then:

$$\mathsf{stall}^t_\lambda \equiv \underbrace{\mathbf{empty}^{t-1}_{\lambda+1} \wedge \neg \mathsf{enable}^t_{\lambda+1}}_{\text{congestion}} \vee \bigvee_{\lambda' > \lambda} \mathsf{stall}^t_{\lambda' \to \lambda}$$

Stream Enable There are three kinds of enable signals: one for the layer as a whole, and another one for each stream instance and window component within the layer. They rely on an empty bit indicating whether the pipeline stage contains meaningful data. This bit is initially 1.

$$\mathbf{empty}^t_\lambda = \begin{cases} \neg \mathsf{eval}^t & \text{if } \lambda = 1 \\ \mathbf{empty}^{t-1}_{\lambda-1} & \text{if } \lambda > 1 \wedge \mathsf{enable}^t_\lambda \\ \mathbf{empty}^{t-1}_\lambda & \text{otherwise} \end{cases}$$

$$\mathsf{enable}^t_\lambda = \neg \mathbf{empty}^{t-1}_\lambda \wedge \neg \mathsf{stall}^t_\lambda \wedge \bigwedge_{\lambda' < \lambda} \bigwedge_\eta (\mathrm{dep}(\sigma_{\lambda',\eta}, \lambda) \implies \mathbf{count}^t_{\lambda'} = \mathbf{count}^t_\lambda + 1)$$

Hence, a layer is enabled if it contains data, is not stalled and all its dependencies are ready, i.e., all layers on which it depends contain the result relevant for the current computation.

Enabling single instances and windows depends on whether they are *active*. An instance $\sigma_{\lambda,\eta}$ is active if its non-semantic criteria are satisfied for the current event and its filter is on. Formally:

$$\mathsf{emactive}^t_{\sigma^p_{\lambda,\eta}} = \mathsf{tmactive}^t(\sigma^p_{\lambda,\eta}) \wedge \mathrm{filter}(\sigma^p_{\lambda,\eta})^t$$

Here, $\mathrm{filter}(\sigma^p_{\lambda,\eta})^t$ refers to the latest value in the storage of the stream which constitutes the filter condition of $\sigma^p_{\lambda,\eta}$. Hence, it is a simple static wiring without further logic. This works because of three reasons: First, recall that in the evaluation order (Definition 2.39), any dependency in the filter expression is resolved in an earlier layer. Hence, the entire filter expression can be manifested in an earlier layer than the stream itself. As a result, resolving the filter boils down to a synchronous stream access.

→ Def. 2.39, p. 79

Now, a stream instance $\sigma^p_{\lambda,\eta}$ is enabled if its layer is enabled and the instance itself is active. These enable signals are sent to the respective stream instance components.

$$\mathsf{enable}^t_{\sigma^p_{\lambda,\eta}} = \mathsf{enable}^t_\lambda \wedge \mathsf{emactive}^t_{\sigma^p_{\lambda,\eta}}$$

Enabling window components works in much the same way. Since they do not conventionally have an evaluation layer, this requires a minor tweak to the input of the evaluation layer analysis. For this, the dependency graph gets a new node for each sliding window occurring in the specification. The stream, whose expression features the aggregation, accesses the new node synchronously, and the new node accesses the target of the evaluation synchronously. Evidently, this does not affect the correctness of the analysis. It merely tricks the evaluation layer analysis to assign the new nodes a layer in-between the layer of its origin and target. If no such layer exists, i.e., target and origin layers are consecutive, it necessarily increases the layer of the origin stream by one to make place for the new node. The window now behaves just like a stream, which yields stalling and enable signals for windows without further ado.

Example 3.2 (Stalling II)**.** Recall the setup from Example 3.1. The following table shows the valuations of the **empty** (`E`), **count** (`C`), stall (`S`), and enable (`N`) over time assuming the queue never runs empty.

Time	1	2	3	4	5	6	7	8
	E C S N	E C S N	E C S N	E C S N	E C S N	E C S N	E C S N	E C S N
$\lambda = 1$	0 0 0 1	0 1 1 0	0 1 0 1	0 2 1 0	0 2 0 1	0 3 1 0	0 3 0 1	0 4 1 0
$\lambda = 2$	1 0 0 0	0 0 0 1	0 1 0 0	0 1 0 1	0 2 0 0	0 2 0 1	0 3 0 0	0 3 0 1
$\lambda = 3$	1 0 0 0	1 0 0 0	0 0 0 1	0 1 0 0	0 1 0 1	0 2 0 0	0 2 0 1	0 3 0 0

The table also shows the impact of each stall criterion except for congestion. The *dependency* clause takes effect at each even timestamp for layer 1 such that layer 3 has an opportunity to compute its value. *Parallel evaluation* takes effect whenever several enable bits are true at the same time, i.e., at each odd timestamp after 3. Lastly, the *recovery* criterion allows the second layer to become active early in even timestamps starting at 4. Without this criterion, the dependency relation would disallow the layer to progress. △

Signaling Readiness The Ctrl needs to inform the QIf when it is ready to receive information regarding the next evaluation. Thanks to the enabling mechanism, this is simple: as long as the first layer is enabled, there are capacities to start the next evaluation. Hence:

$$\mathsf{done}^t = \mathsf{enable}_1^t$$

Conveying Alarms Some streams represent triggers. These streams have a feedback channel to the Ctrl. This wire is active-low, meaning that it carries a 0 unless the stream writes a 1 on it. As a result, the Ctrl waits on a rising edge of the signal, which prompts it to write the respective bit in the output register connected to the system. It is the responsibility of the system to reset the bit again after it has received the information.

Instance Handling The last responsibility of the Ctrl is to manage creation and termination of instances. In particular, this includes updating the instance store. For this, recall that it consists of an **alive** bit and a **spawn** register containing a timestamp for each potential instance. Since the Ctrl already has access to the current timestamps of each stage via **curr_ev**, it only needs to pass this information on whenever a stream is spawned or closed. To this end, for a stream σ with spawn expression ex_s and close expression ex_c, there are feedback channels from the components of ex_s and ex_c to Ctrl similar to the one for triggers. This feedback channel is only one bit wide, indicating whether the respective instance was spawned/closed. Since there is a single, fixed component per *possible* instance, the parameter values can be hard coded. When the spawn and close bits are raised, Ctrl set or resets the alive bit, respectively, and updates the spawn timestamp in case of a spawn.

Another effect of a spawn is that the new instance might have to be evaluated in a lower stage of the same evaluation process. For this, recall that a periodic stream is only evaluated after its period has passed at least once. Hence, by assumption, a spawn cannot affect any stream instance in the pipeline. For event-based streams, recall that the $\mathsf{tmactive}^t(\sigma^p_{\lambda,\eta})$ signals separately check the alive status of their instances every step. Hence, no need for further action.

This is not true for termination. While termination only takes effect *after* full evaluation of an event or deadline (cf. Definition 2.26), the pipeline might already contain the next one. Hence, the Ctrl has to reset the respective bits in **curr_ev**. This update process is integrated into the propagation mechanism for **curr_ev**. To this end, Ctrl computes an additional signal for each periodic stream instance. Here, $\mathsf{close}^t_\lambda\left[\sigma_{\lambda,\eta}, p\right]$ is the output of the feedback channel of the close expression. → Def. 2.26, p. 67

$$\mathsf{closemask}^t_\lambda\left[\sigma_{\lambda,\eta}, p\right] = \left(\neg\mathsf{close}^t_\lambda\left[\sigma_{\lambda,\eta}, p\right] \gg n^{\uparrow}_{\text{static}}\right) \circ 1^{n^{\uparrow}_{\text{ev}}+|ts|+\sum(bits(\sigma^{\downarrow})+1)}$$

Here, the shift operation appends 1s from the left to create a proper mask for **curr_ev**. That means, closemask is a string as wide as **curr_ev**, consists of 1s and a 0 for each closed periodic stream instance. With this, Equation (3.2) is replaced by: → Eq. (3.2), p. 115

$$\mathsf{closeupdate}^t_\lambda = \bigvee_{\lambda'>\lambda}\left(\mathsf{closemask}^t_{\lambda'} \wedge \mathbf{count}^{t-1}_{\lambda'} \neq \mathbf{count}^{t-1}_{\lambda} \wedge \mathsf{enable}^t_{\lambda'}\right)$$

$$\mathbf{curr\_ev}^t_\lambda = \begin{cases} \mathsf{q\_out}^t \wedge \mathsf{closeupdate}^t_{\lambda'} & \text{if } \lambda = 1 \wedge \mathsf{en}^{t-1}_1 \\ \mathbf{curr\_ev}^{t-1}_{\lambda-1} \wedge \mathsf{closeupdate}^t_{\lambda'} & \text{if } \lambda > 1 \wedge \mathsf{en}^{t-1}_\lambda \\ \mathbf{curr\_ev}^{t-1}_{\lambda} \wedge \mathsf{closeupdate}^t_{\lambda'} & \text{otherwise} \end{cases}$$

As a result, the close update only has an effect if a stream instance was closed in a greater layer processing an earlier event.

Stream Nodes

Stream nodes consist of a logic and a memory part. The logic part is a straight-forward translation of the stream expression with one exception: Sliding window aggregations are translated into accesses to the respective window nodes. The memory part provides an array of registers with space to store enough values of the stream according to the memory bound defined in Definition 2.40. Whenever the node is enabled, it triggers its evaluation circuit, shifts each register one to the right, evicting the oldest value, and commits the new value to the first register. Any access to the memory of a stream node by another stream handles indices accordingly. → Def. 2.40, p. 81

Suppose stream $\sigma_{\lambda,\eta}$ has a memory bound of μ. For an instance $\sigma^p_{\lambda,\eta}$ of this stream, let res be the output of its expression circuit. Its internal memory is then a register array **mem** of length μ.

$$\mathbf{mem}^t[i] = \begin{cases} \mathbf{mem}^{t-1}[i] & \text{if } \neg\mathsf{enable}^t_{\sigma^p_{\lambda,\eta}} \\ \mathbf{mem}^{t-1}[i-1] & \text{if } \mathsf{enable}^t_{\sigma^p_{\lambda,\eta}} \wedge i > 1 \\ \mathsf{res}^t & \text{if } \mathsf{enable}^t_{\sigma^p_{\lambda,\eta}} \wedge i = 1 \end{cases}$$

Note that input streams operate in much the same way except that it does not have a logic part, thus res contains its value in the current event obtained by the CTRL.

Lastly, if a stream node is target of a window aggregation ω, it has to notify the respective window upon computation or reception of a new value. For this, it raises an update-enable signal (upden_ω) and transmits its latest value over winin_ω.

$$\mathsf{upden}^t_\omega = \mathsf{enable}^t_{\sigma^p_{\lambda,\eta}}$$

$$\mathsf{winin}^t_\omega = \mathsf{res}^t$$

Window Nodes

Window accesses only refer to the latest value of the window. Hence, it only needs to store one result value. In addition to this, it has to manage all intermediate values as detailed in Appendix A.1.3. While the window updates its internal state when receiving an update enable, it only updates its result register when it is enabled. This prevents overwriting the result prematurely, which would render potential accesses from later pipeline stages incorrect. → Appx. A.1.3, p. 211

Upon reception of an update enable signal, the window invalidates all pre-aggregation results ("buckets") that became obsolete between now, and the result of the last evaluation. It also determines the currently active bucket. Afterwards, it applies the map function on the input and reduces the result with the content of the active bucket. This result is then committed to the bucket. For this, the translation of the map and reduce functions are straight-forward as they are stateless, arithmetic functions.

$$\begin{aligned}
\mathsf{passed}^t &= \mathsf{ts}^t - \mathbf{last\_ts}^{t-1} \\
\mathsf{inv'}^t[i] &= \mathsf{passed}^t > \delta \cdot i \\
\mathsf{inv}^t &= \mathrm{csr}(\mathsf{inv'}^t, \mathbf{cix}^t.\,in) \\
\mathsf{tix}^t &= \begin{cases} \mathrm{csr}(\mathrm{last}(\mathsf{inv'}^t), \mathbf{cix}^t) & \text{if } \mathsf{inv}^t \neq 0 \\ \mathbf{cix}^{t-1}.\,in & \text{otherwise} \end{cases} \\
\mathbf{bucket}^t[i] &= \begin{cases} \mathbf{bucket}^{t-1}[i] \oplus \mathrm{map}(\mathsf{winin}^t) & \text{if } \mathsf{upden}^t \wedge \mathsf{tix}^t[i] \wedge \neg\mathsf{inv}[i] \\ \varepsilon \oplus \mathrm{map}(\mathsf{winin}^t) & \text{if } \mathsf{upden}^t \wedge \mathsf{tix}^t[i] \wedge \mathsf{inv}[i] \\ \varepsilon & \text{if } \mathsf{upden}^t \wedge \neg\mathsf{tix}^t[i] \wedge \mathsf{inv}[i] \\ \mathbf{bucket}^{t-1}[i] & \text{if } \neg\mathsf{upden}^t \vee \mathsf{tix}^t[i] \wedge \neg\mathsf{inv}[i] \end{cases} \\
\mathbf{cix}^t &= \begin{cases} \mathsf{tix} & \text{if } \mathsf{upden}^t \\ \mathbf{cix}^{t-1} & \text{otherwise} \end{cases} \\
\mathbf{last\_ts}^t &= \begin{cases} (\mathsf{ts}^t \operatorname{div} \delta) \cdot \delta & \text{if } \mathsf{upden}^t \\ \mathbf{last\_ts}^{t-1} & \text{otherwise} \end{cases} \\
&\approx \begin{cases} \mathsf{last\_ts}^t + \mathrm{popcnt}(\mathsf{inv'}^t) \cdot \delta & \text{if } \mathsf{upden}^t \\ \mathbf{last\_ts}^{t-1} & \text{otherwise} \end{cases}
\end{aligned}$$

Here, δ is the time a single bucket represents, i.e., the duration of the window divided by the period of the stream accessing the window. Moreover, ts is the timestamp extracted from **curr_ev** of the respective layer.

These formulas use and maintain

- the **cix** register, containing the index of the currently active, i.e., "most recent" bucket in unary encoding,
- the **last_ts** register containing the timestamp representing the first point in time covered by the currently active bucket. It starts with 0 and progresses in increments of multiples of δ, and
- the array of **bucket** registers containing the intermediate results of each bucket.

The algorithm first checks how much time has passed since the starting time of the bucket that was active in the last update. Next, it concurrently computes as many bits as it has buckets. Each bit with value 1 indicates that a bucket is outdated and thus needs to be invalidated. This bit-string is then cyclically shifted to the right by the index of the active bucket in the last update. As a result, the bit at index i corresponds to the i^{th} bucket. Next, it determines the index of the target bucket, i.e., the bucket in which the new value needs to be placed. This is the current index if no bucket needs to be invalidated. Otherwise, it resets every set bit in the inv' signal except the last one. This operation can be efficiently realized with a parallel prefix circuit. The result also needs to be shifted by **cix**.

With this information, the node updates its buckets. Here, there are four cases. The first two cover the currently active bucket provided it needs to be updated. If the bucket does not have to be invalidated, the node accesses the old bucket value, reduces it together with the mapped new value, commits the result in the bucket. Otherwise, it uses the neural element of the aggregation rather than the last bucket value. The third line states that a bucket that is not targeted and needs to be invalidated will simply receive the neural element. Lastly, buckets retain their value if there is no update, or if they are neither targeted nor get invalidated.

At the same time, **cix** gets the value of tix unless there is no update. Lastly, when updated, **last_ts** becomes the current timestamp rounded down to the next δ. While this is possible by dividing and the multiplying by δ, such operations are expensive. There is a cheaper solution available under the condition that there will be at least one update per duration of the window Δ. Even though events cannot guarantee this, it suffices if there is a static, periodic stream with a period less than Δ. In this case, adding the population count[4] of inv' multiplied by δ to **last_ts** yields the same result.

Evaluation Since most of the heavy lifting takes place in the update mechanism, the retrieval of the result is fairly simple. Note that $\Delta\delta^{-1}$ is a compile-time constant.

$$\mathbf{res}^t = \begin{cases} \mathrm{fin}(\mathbf{bucket}^{t-1}[1] \oplus \ldots \oplus \mathbf{bucket}^{t-1}\left[\Delta\delta^{-1}\right]) & \text{if } \mathsf{en}_\omega^t \\ \mathbf{res}^{t-1} & \text{otherwise} \end{cases}$$

Since $\oplus$ is an associative function, the reduction can be realized efficiently in a binary tree.

3.1.5. Synthesizer

The mathematical descriptions can be translated into VHDL code. This code is then synthesizable onto an FPGA, for example via the Vivado Design Suite developed by Xilinx. The tool also analyzes the description and provides insight into the realization. This includes the resource consumption in terms of logical gates, memory cells, peak and idle power consumption as well as the slack time.

[4]The population count, hemming weight, or horizontal sum of a bit-string denotes the number of 1s occurring in it.

3.1.6. Performance

The validation of the hardware realization is based on a prototype compiler from RTLola to VHDL implementing the theory presented in this chapter [Bau20; Bau+20a]. It encompasses three case studies. The first two are concerned with monitoring a network and an aircraft. On the contrary, the third one is purely synthetic and designed to emphasize the benefits of the parallel and pipelined evaluation structure. All specifications were compiled into VHDL code and then synthesized on a Zynq-Z-7010 ARM/FPGA SoC Trainer Board[5], which is logic-equivalent to an Artix-7 FPGA. The board features 4,400 logic slices, each with four 6-bit lookup tables (LUT) and 8 flip-flops (FF).

Aircraft Monitoring

Consider the following specification for a drone.

```
input lat, lon, spd: Int32
input sats: UInt8

output gnss_freq @1Hz: bool := lat.count(over: 1s, or: 10) < 9
trigger gnss_freq "GNSS frequency less than 9 Hz"

output fast := spd > 700
trigger fast.last(or: false) ∧ ¬fast "Slowing down"

output gnss_dist := sqrt(δ(lon)^2 + δ(lat)^2)
output gnss_dps := gnss_dist / δ(time)
trigger abs(gnss_spd - spd) > 10 "Sensor deviation"

output hovering @1Hz := velo.integrate(over: 5s, or: 5) < 1
trigger hovering "Little distance covered"
```

Input events consist of positional values, i.e., longitude and latitude, the airspeed and the number of GNSS satellites in range. The GNSS module is supposed to send values with a frequency of 10 Hz. The output stream `gnss_freq` counts the number of samples received within a second and checks if it falls below 9. In this case, the first trigger reports the unexpectedly low sample frequency. The second trigger reports a warning when the drone's airspeed drops below 700, provided that the speed was greater than 700 before that. For the third trigger, the specification uses a simplified reconstruction of the distance the drone traveled using the Pythagorean theorem. This is only a rough approximation for illustration, the haversine function for example would yield more accurate results. The square root computation is realized using the constant-time function proposed by Li

[5]https://reference.digilentinc.com/reference/programmable-logic/zybo/reference-manual?_ga=2.102758273.1814454663.1555084001-1980681841.1546416239; last accessed: 02.02.2022

Table 3.7.: Results when realizing a specification for aircraft onto a Zynq-Z-7010 FPGA. Power consumption amounted to 0.121 W when idle and peaked at 1.620 W.

Component	FF	LUT	MUX	CA	MULT
Total	3,036	3,685	26	656	18
TM	901	156	0	22	0
QUEUE	543	442	0	43	0
EM	1,281	2,820	0	576	18

and Chu [LC96]. Differentiating the distance discretely computes the speed according to the GNSS module. This allows for cross-validating sensor values by comparing the sensed input speed with the computed one. If the deviation between the two values reaches a critical point, the monitor raises an alarm. Lastly, the specification detects hover phases by integrating both speed values and checking whether it lies below a threshold.

Table 3.7 presents the resource consumption in terms of required flip-flops (FF), look-up tables (LUT), multiplexers (MUX), adders (CA), and multipliers (MULT) for each component. Note that the realization purged the instance store due to a lack of parametrized streams.

Note further that the amount of resources like flip-flops of the entire monitor is not equal to the sum of the resources of all components. The difference is required for internal tasks such as signal management. One can see that most flip-flops reside in the EM because it manages the persisted values of all streams. Yet, the difference is relatively small, with the TM requiring 70% as much memory as the EM. This can be contributed to the fact that the TM uses several internal registers and that the greatest offset in the specification is only 1, which keep the memory requirement of the EM low. The overwhelming majority of logic gates, in particular look-up tables, adders, and multipliers, reside in the EM which was expected given that this component implements the evaluation logic. The 18 multipliers are mainly required for squaring the δ-values and computing the integral window. The power consumption amounted to 0.121 W when idle and 1.620 W under peak pressure.

The monitor was tested against data generated with the ArduPilot[6] Copter[7] drone simulator. It simulated a multicopter flying over the campus of Saarland University. Sensor information was piped to the monitor over a serial port. Evaluating events and periodic deadlines took an average of 428 system clock cycles with a frequency $\zeta_{\mathsf{sys}} = 100\,\mathrm{MHz}$. Thus, processing an event took 4.28 µs on average. Here, the worst slack amounted to 1.653 ns.

[6]`http://ardupilot.org/`; last accessed: 02.02.2022

[7]`http://ardupilot.org/copter/index.html`; last accessed: 02.02.2022

```
input src, dst: IPv4
input fin, push, syn: bool
input length: UInt16

output host: IPv4 = ...

output incoming_connection: NoValue
    eval when dst = host

trigger @1Hz incoming_connection.count(over: 0.5s) > 10000
  "Many incoming connections"

output data_received @incoming_connection :=
    eval when push with length
trigger @1Hz received.sum(over: 1s) > 10^7 "Workload too high"

output opened @incoming_connection
    eval when syn with opened.last(or: 0) + 1
output closed @incoming_connection
    eval when fin with closed.last(or: 0) + 1
trigger open - closed < 0 "Closed more connection than were open"
```

Listing 3.1: RTLola specification for monitoring network traffic

Network Monitoring

Network monitoring differs from aircraft monitoring in several major points. It relies less on heavy arithmetic computations and more on simple counting and comparison-based checks rendering the workload per event rather low. To make up for that, events occur significantly more often; the sheer amount of network traffic that arises exerts an immense pressure on the monitor. Here, inputs usually arrive at no discernible frequency — as opposed to most inputs of an aircraft, which stem from sensors with a fixed sampling frequency.

With these differences in mind, recall the specification from the interpreter evaluation in Section 2.7.6 depicted in Listing 3.1. This specification allows for pressure testing the generated FPGA monitor.

→Sec. 2.7.6, p. 88

Realizing the specification reveals that the resource consumption is lower compared to the avionics example, mostly due to the lower arithmetic complexity of the specification. The number of look-up tables decreases by around 60%, adders by 65% and multipliers by 100%. The number of flip-flops only decreases by around 38%. While there is no significant difference in the number of sliding windows and lookup expressions in the

Table 3.8.: Results when realizing a specification for aircraft onto a Zynq-Z-7010 FPGA. Power consumption amounted to 0.120 W when idle and peaked at 1.570 W.

Component	FF	LUT	MUX	CA	MULT
Total	1905	1533	23	226	0
TM	550	161	0	37	0
QUEUE	330	342	0	28	0
EM	895	927	0	161	0

two specifications, integral windows require 5-times as much memory as summation and count windows. Neither the idle, nor the peak power consumption showed significant differences. Table 3.8 displays the full statistics.

The empirical evaluation replays data from the Mid-Atlantic Collegiate Cyber Defense Competition (MACCDC)[8] in real time. While the evaluation process is simpler, the TM remains mostly the same. Thus, the amount of system clock cycles required per event only decreases by around 25%, the response time for a single event is 3.2 µs on average. The worst slack time, however, increased by 150% to 4.0 ns. This allows for safely increasing the system clock frequency by up to 200 MHz. The reason for this is that the square root computation in the avionics specification has a significantly greater depth than all operation performed while monitoring the network. Since this computation is part of a single pipelining stage, it must be completed in a single cycle, so the slack time decreases significantly when the clock frequency remains unchanged.

Parallelization

The third case study is designed to showcase the benefit of realizing the monitor in hardware. The following scalable specification forms the basis for the evaluation. It imitates a command-response structure, i.e., depending on the value of the `command` input it performs some simple, meaningless arithmetic checks. Every trigger is event-based and ticks for every possible input event.

```
input command: Int16
input x, y, z, ...: Int32

trigger @⊤ ϕ₁ ∧ cmd = 1

    ...
```

[8]https://www.netresec.com/?page=MACCDC; last accessed: 02.02.2022

```
trigger @T ϕ512 ∧ cmd = 512
```

The empirical evaluation uses two different EM realizations. The first conforms to Section 3.1.4. The second one injects spurious dependencies between consecutive triggers, forcing each one into a separate pipeline stage. This prevents any form of parallel evaluation. A stress-test successively increases the input data rate until the EM can no longer process events in time. For this, the companion processor on the board sends events to the FPGA and measures the time it takes until a verdict is available. This setup produces more robust results than connecting the board with an external device using serial communication.

→Sec. 3.1.4, p. 111

Unsurprisingly, neither the size of the realization, nor the power consumption varied noticeably between the realizations. On the contrary, the execution time varies immensely. While the sequential execution takes 43.83 µs, the speed of the parallel execution exceeds the capabilities of the processor of up to 866 MHz, which amounts to 3.77 µs between sending an event and attempting to read the output. Practically, this means that if the processor sends events to the FPGA with its maximum frequency, the parallel realization can process all events, whereas the sequential one misses 89% of the events.

These results render the hardware compilation particularly efficient when the specification contains a multitude of independent streams. While such specifications seem artificial, they are commonplace. Not only are command-response patterns crucial to check compliance of the system to a control entity, independent computations are typical for modular systems like CPS. On the one hand, since the monitor has several data sources, each (sensor-) input needs separate validation. On the other hand, checks for geofence violations, for example, require to compare the trajectory of the system with several faces of the fence at once. This closely resembles the structure of the example specification. So, in conclusion, the presented realization of parallel structures pays off for a large variety of realistic specifications.

3.1.7. Related Work

Early work in the translation of runtime monitoring specification into hardware used the assertion-based verification language PSL (property specification language). The ToCs [Dah+05] tool, developed by IBM Haifa translated PSL specifications to VHDL/Verilog code, which was successfully deployed in an industrial system-on-chip project. Similarly, P2V [LF07] compiled sPSL assertions to Verilog, and more recently MBAC [BZ08] synthesizes automata-based monitors from PSL assertions. Apart from PSL, there is work based on bounded future LTL [FK09], and past-time LTL, for which BusMOP [Pel+08] synthesized circuits for monitoring PCI bus traffic.

Transitioning to real-time properties, Jaksic et al. [Jak+15] realize STL specifications onto an FPGA. Further, the R2U2 tool [MRS17] generates an external FPGA monitor for MTL specification with future dependencies.

All of these approaches use logic-based input languages. However, realistic CPS often require quantitative stream-based languages to express properties beyond yes-or-no verdicts. There is significant work on synchronous programming languages: Lustre [Hal+91; Hal05] and Esterel [BG92] are widely used for the development of digital circuits [Ber16]. Moreover, there is a compiler for Lola specification onto FPGA via VHDL presented by Maltry [Mal17].

Lastly, for one other asynchronous extension of Lola, TeSSLa [Con+18a], there is also a hardware realization [Con+18b]. The realization divides the monitor into several easy-to-replace chunks. This modularization requires instantiations of the blocks to conform to a certain schema, slightly reducing performance. On the plus side, this significantly reduces deployment time, benefitting rapid prototyping. This approach is contrary to RTLola, where specification development is intended to take place on the software side via the interpreter. Integration is then independent of the specification due to the decoupling of monitor and system. Plus, there is no intention to switch out parts of the monitor post-deployment. Hence, for the RTLola framework, low deployment time of the hardware monitor is not considered an issue. Ultimately, the question of which framework works better in terms of hardware realization depends on the details of the system under development.

3.2. Software Compilation

The software compilation is the second form of monitor realization. This compiler is a verifying one, i.e., it injects verification annotations into the generated Rust code. These annotations enable the Viper [MSS16] toolkit with the Prusti [Ast+19] frontend, to verify the correctness of the generated code. Here, correctness encompasses that monitor verdicts correspond to the formal semantics of the specification language, and the termination of each event processing step.

The input language for the translation is the synchronous, discrete-time origin of RTLola, i.e. Lola. Hence, this section first outlines the semantic differences between the languages while sticking to the syntax of RTLola for a more uniform presentation. An analysis of the specification identifies a pre- and postfix of the monitor execution. Treating these stages differently creates for a more efficient implementation. The section then presents the translation into rust code as well as generation of verification annotations. Here, the code generation can either produce a sequential or a parallel monitor. While parallelization is always beneficial in hardware, this is not the case in software since concurrency incurs significant runtime overhead.

An empirical evaluation reveals that the generated sequential monitor requires only 1.4% to 3.1% as much time than the interpreter. However, the benefit or penalty of the concurrent one is not universal since it highly depends on the specification at hand. Here, a wide dependency graph with expensive stream expressions is optimal, resulting in an additional speedup of up to 60%.

A performance evaluation of the compilation and verification showed that the running time and memory consumption of the former is entirely subsumed by the latter. Nearly the entire time is spent in the underlying SMT solver Z3 [MB08]. Unfortunately, its running time is highly erratic resulting in inconsistent — yet non-contradictory — results for complex specifications. On the positive side, attempting to verify a large specification in the domain of avionics found in the literature, the verification reported a potential division by zero. This turned out to be a true positive. Only after fixing it did the verification succeed.

3.2.1. Stripping the RT off RTLola

The Lola language [DAn+05] was conceived in 2005 and is the “grandparent” of RTLola as presented in this thesis. As the lack of the “real-time (RT-)” prefix already indicates, Lola is fully discrete. This results in an absence of periodic streams and sliding window aggregations. Moreover, the language is fully synchronous without dynamic stream creation, so an event always covers all input streams and effects all outputs, rendering hold-accesses futile. As a result, the type system of Lola reduces to the mere value type lattice. Though, this timing model allows for an efficient handling of offsets targeting future values of streams rather than past ones.

The Lola syntax presented here stays true to the original Lola syntax except for the shape of stream expressions and trigger messages. So a specification is a sequence of stream declarations. Input stream declarations are of the form `input` $\sigma_j^\downarrow$`:` T_j, where $\sigma_j^\downarrow$ is an input stream and T_j is its (value) type. Output streams have the shape $\sigma_j^\uparrow$: $T_j = e_j(i_1, \dots, i_m, s_1, \dots, s_n)$, where $\sigma_1^\downarrow, \dots, \sigma_m^\downarrow$ are input streams, $\sigma_1^\uparrow, \dots, \sigma_n^\uparrow$ are output streams, and e_j are stream expressions. These consists of constant values, arithmetic and logic functions $f(e_1, \dots, e_k)$, conditional expressions `if` c `then` e_1 `else` e_2, and stream accesses σ`.offset(by:` k`, or:` c`)`, where σ is a stream, k is the *offset*, and c is the constant default value. The offset may be any integer number including 0, in which case the `.offset(by:` k`, or:` c`)` suffix may be omitted.

Example 3.3 (Running Example)**.** The following specification will serve as a running example throughout this section.

```
input altitude: Int32
output tooLow: Bool :=
  altitude.offset(by: -1, or: 0) < 200
  ∧ altitude < 200
  ∧ altitude.offset(by: 1, or: 0) < 200
output tooHigh: Bool :=
   altitude.offset(by: -1, or: 0) > 600
   ∧ altitude > 600
   ∧ altitude.offset(by: 1, or: 0) > 600
trigger tooLow "Flying below minimum altitude."
trigger tooHigh "Flying above maximum altitude."
```

The specification monitors the altitude of a drone. For this, the output stream `tooLow` checks whether the altitude is lower than `200` in the last, current, and next step. Analogously, `tooHigh` checks whether these values exceed 600. In either case, one of the triggers would go off and issue an alarm. As for RTLola, the evaluations of `tooHigh` and `tooLow` try to access the penultimate value of `altitude` as well as the last and the next one. When `altitude` does not have at least two values, the accesses with offset -1 fail and the monitor substitutes the default values, in this case `0`. Conversely, when `altitude` ceases to produce values upon completion of the mission, the accesses with offset $+1$ will fail. Hence, in contrast to negative offsets, the default value for accesses with positive offset become relevant at the end of the execution. △

Semantics

The semantics of Lola is defined in terms of *evaluation models*. Intuitively, an evaluation model contains the values and evaluations of each input and output stream of the specification. The evaluation is equivalent to the one for RTLola in Section 2.4 when →Sec. 2.4, p. 61

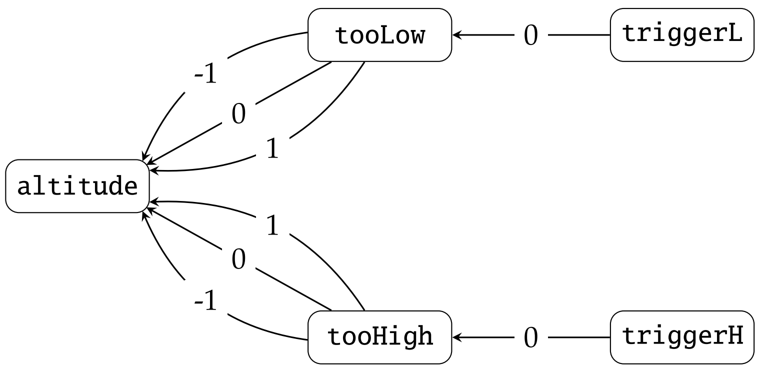


Figure 3.9.: The dependency graph D_Φ induced by the running example specification.

striping off real-time, asynchrony, filtering and dynamic stream management. It can also be found in d'Angelo et al.'s work [DAn+05] in its original syntax.

Definition 3.4 (Evaluation Model [DAn+05])

Let Φ be a Lola specification over input streams $\sigma_1^\downarrow, \dots, \sigma_{n^\downarrow}^\downarrow$ and output streams $\sigma_1^\uparrow, \dots, \sigma_{n^\uparrow}^\uparrow$. The tuple $M = (m_1, \dots, m_{n^\downarrow + n^\uparrow})$ of sequences of length N is called an *evaluation model* if for each equation $\sigma_j^\uparrow = e_j(\sigma_1^\downarrow, \dots, \sigma_{n^\downarrow}^\downarrow, \sigma_1^\uparrow, \dots, \sigma_{n^\uparrow}^\uparrow)$ in Φ, $(m_{n^\downarrow+1}, \dots, m_{n^\downarrow+n^\uparrow})$ satisfies $m_{n^\downarrow+j}[k] = [\![e_j]\!]_{M[\dots k]}$ for $0 \leqslant k \leqslant N$, where $[\![e_j]\!]_{M[\dots k]}$ evaluates the stream expression e_j under M up to position k.

The resulting dependency graph is defined as:

Definition 3.5 (Dependency Graph [DAn+05])

Def. Lola Dependency Graph

The *dependency graph* $D_\Phi = (V, E)$ of a Lola specification Φ is a weighted directed multigraph with $V = \mathcal{S}^\downarrow \dot\cup \mathcal{S}^\uparrow \dot\cup \mathcal{S}^!$. Each edge represents an access operation. Thus, $(\sigma_1, w, \sigma_2) \in E$ iff $w \in \mathbb{Z}$ and the stream expression of σ_1 contains an access operation to σ_2 with offset w.

Example 3.6 (Dependency Graph)**.** Figure 3.9 depicts the dependency graph of the running example. It consists of five nodes representing the input stream `altitude`, the output streams `tooLow` and `tooHigh`, as well as the two triggers. The edges represent the stream accesses with their labels being the corresponding offsets. While the triggers only access the output streams with an offset of 0, both of them access the input stream `altitude` with offsets -1, 0, and 1. △

Based on the dependency graph, d'Angelo et al. define the *shift* of a stream [DAn+05]. Intuitively, the shift of σ indicates how many steps the evaluation of its expression needs to be delayed. For instance, suppose the shift of σ is n. Then, the value of σ for time t can be computed at time $t + n$. Formally:

Definition 3.7 (Shift [Osw20])

For a Lola specification Φ, the *shift* $\Delta(\sigma)$ of a stream σ is the greatest weight of any path through D_Φ originating in σ: **Def.** Shift

$$\Delta(\sigma) = \max\{0\} \cup \{w + \Delta(\sigma') \mid (\sigma, w, \sigma') \in E\}$$

The shift enables the definition of the modified evaluation order via *synchronized edges* E^*. Here, the weight of a synchronized edge $(\sigma, m, \sigma') \in E^*$ represents the relative time difference of when values of σ and σ' are computed. A value of 0 indicates that σ attempts to access a value of σ' at the same time at which it is computed. So, $E^* = \{(s, \Delta(s) - w - \Delta(s'), s') \mid (s, w, s') \in E\}$. Synchronized Edge

Definition 3.8 (Evaluation Order)

The *evaluation order* $\prec$ is a partial order on the output streams of a Lola specification Φ. Let $D_\Phi = (V, E)$ be the dependency graph of Φ with synchronized edges E^*. The evaluation order is the transitive closure of a relation $\prec$ with $\sigma \prec \sigma'$ iff $(\sigma, 0, \sigma') \in E^*$. **Def.** Lola Evaluation Order

Note that the definition of evaluation layers seamlessly translates from RTLola.

Definition 3.9 (Lola Evaluation Layer)

Let Φ be a Lola specification and let $\prec$ be the evaluation order induced by its dependency graph. A stream σ is in *layer* λ, written $\text{Layer}(\sigma) = \lambda$, then there is a strictly decreasing sequence of λ streams w.r.t. $\prec$ starting in σ. **Def.** Evaluation Layer

Example 3.10 (Evaluation Order)**.** The output streams and triggers are pairwise incomparable w.r.t. $\prec$. The order puts inputs least in layer 1, followed by the outputs in layer 2, and the triggers in layer 3. △

While the concurrent hardware realization utilized the partial order, sequential software solutions require a total order instead. Hence, the *total evaluation order* $\prec^+$ of a specification is obtained by strengthening $\prec$ arbitrarily. **Def.** Total Evaluation Order

The evaluation order resolves issues arising from accesses with an offset of 0 for Lola in the same way as it does for RTLola. Additionally, specifications where the dependency graph has no positive cycles are called *efficiently monitorable*. Such specifications can be monitored with constant memory, and an output value can always be produced after a constant delay [DAn+05]. All example specifications considered in this section are efficiently monitorable. Efficiently Monitorable

3.2.2. Specification Analysis

After determining the evaluation order and inter-stream dependencies, the compilation performs an additional analysis. Here, it identifies the pre- and postfix of the evaluation and the overall memory consumption.

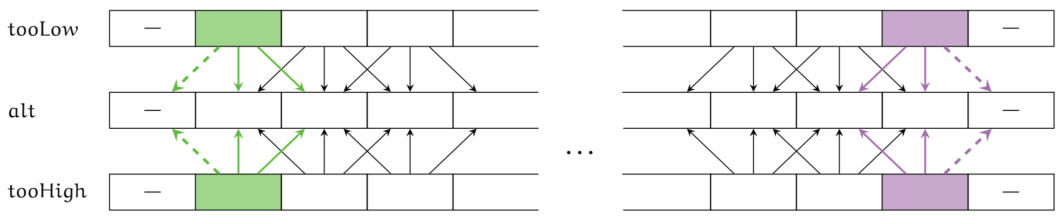


Figure 3.10.: Illustration of stream accesses in different phases of the execution of a monitor for the running example. The output streams access the input stream with offsets -1, 0, and $+1$. Dotted arrows indicate past accesses in the prefix and future accesses in the postfix. These accesses need to be replaced by their default values.

Execution Pre- and Postfix.

Refer back to the running example. Here, the accesses to `altitude` have an offset of at least -1. Hence, as soon this stream generated at least two values, they will always succeed. Thanks to the synchronous nature of Lola, this is the case starting at time $t = 2$.

Generally, suppose an output stream σ accesses another stream σ' with an offset of n. If n is strictly negative, then accesses fail until $t = \Delta(\sigma) - n - \Delta(\sigma')$. If n is 0 or strictly positive, they always succeed. However, in the latter case, the evaluation of σ needs to be delayed by $\Delta(\sigma) - n$, i.e., until σ' received the respective value. With this delay, all accesses to σ' continue to succeed until σ' ceases to produce new values. As soon as this is the case, the monitor needs to evaluate σ for $\Delta(\sigma) - n$ more times to compensate for the delay. In these evaluations, these accesses definitely fail, since they refer to non-extant future values of σ'.

Prefix
Monitor Loop
Postfix

This behavior induces the structure of the monitor execution: it starts with a *prefix* where past accesses can fail, *loops* in the regular execution where all accesses always succeed, and ends in a *postfix* where future accesses always fail. Figure 3.10 illustrates the different phases for the running example.

Memory Requirement

While the shift mainly concerns time, it can also be used to compute the memory requirement of a stream, i.e., the number of values of a single stream that can be relevant at the same time. If a stream σ of type T has a memory requirement $\mu(\sigma) = i$, the monitor needs to reserve $i \cdot \text{size}(T)$ bytes of memory for σ.

Definition 3.11 (Memory Requirement)

Def. Memory Requirement

The *memory requirement of a dependency* $(\sigma', w, \sigma) \in E$ is determined by the shifts of the streams σ, σ' as well as the weight w of the dependency, i.e., the offset of the stream access: $\Delta(\sigma) - \Delta(\sigma') - w$. The memory requirement $\mu(\sigma) \in \mathbb{N}$ of a *stream* σ is thus the maximum memory requirement of any ingoing dependency:

$$\mu(\sigma) = \max(\{1\} \cup \{\Delta(\sigma) - \Delta(\sigma') - w \mid (\sigma', w, \sigma) \in E\}).$$

As a result, the requirement defaults to 1 if there are no incoming dependencies.

Example 3.12 (Memory Requirement)**.** Recall the running example and its dependency graph shown in Figure 3.9. The memory requirement of `altitude` is 2 because its shift is 0, while the shift of both `tooLow` and `tooHigh` is -1. Both output streams access `altitude` with offset -1, so $\mu(\texttt{alt}) = 0 - (-1) - (-1) = 2$. The memory requirement of `tooLow`, `tooHigh`, and both triggers is 0. △

Based on the shift as well as on the memory requirement of the streams, the compilation determines three key values for each specification.

Definition 3.13 (Prefix and Postfix Length, Memory Consumption)

Let Φ be a Lola specification. The *prefix length* $\eta^{\leftarrow}_{\Phi}$, *postfix length* $\eta^{\rightarrow}_{\Phi}$, and the *memory consumption* μ^*_{Φ} of φ are defined as follows:

Def. Prefix and Postfix Length

Def. Memory Consumption

$$\eta^{\leftarrow}_{\Phi} = \max_{s \in \Phi} \{\Delta(s) + \mu(s)\}$$

$$\eta^{\rightarrow}_{\Phi} = \max_{s \in \Phi} \{\Delta(s)\}$$

$$\mu^*_{\Phi} = \sum_{s \in \Phi} \{\mu(s) \cdot \text{size}(T_s)\}$$

Example 3.14 (Prefix and Postfix Length)**.** The prefix length of the running example is $\eta^{\leftarrow}_{\Phi} = 2$ since the shift of the input stream is 0 and its memory requirement is 2. The postfix length is $\eta^{\rightarrow}_{\Phi} = 1$ as both outputs have a shift of 1. △

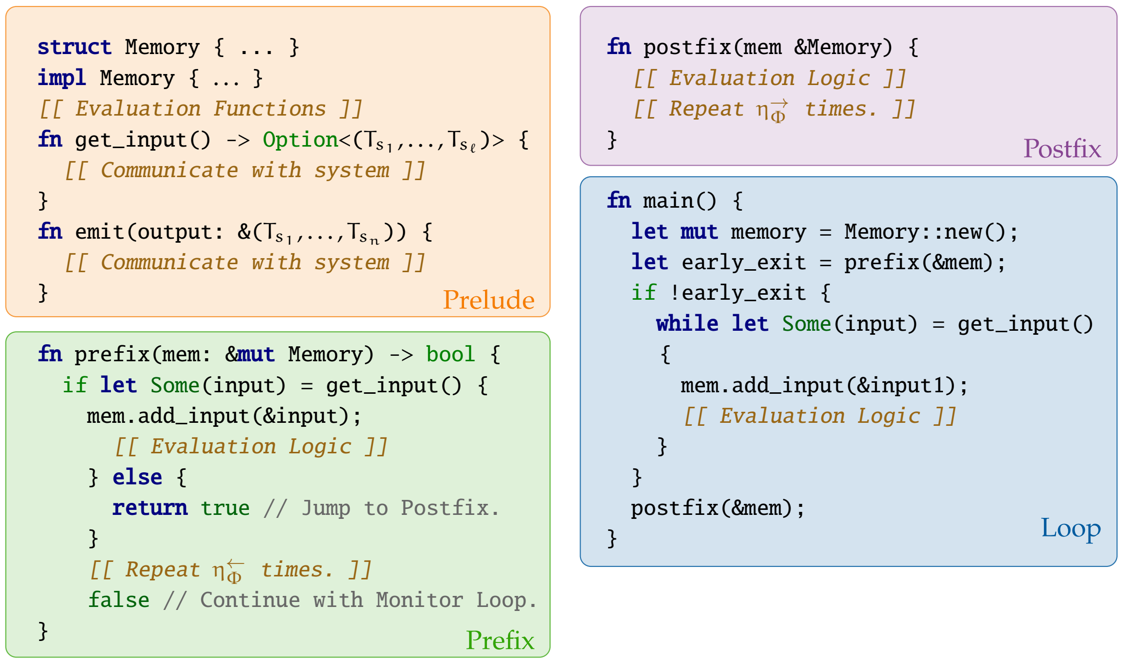
3.2.3. Code Generation

Figure 3.11 illustrates the general structure of the monitor. The remainder of this subsection detail each component. Here, the `main` function constitutes the entry point. It allocates static memory and initiates the execution of the prefix, monitor loop, and postfix.

Prelude

The monitor code starts with a *prelude*. It declares and allocates the working memory plus provides several helper functions required throughout the monitoring. The first two functions are I/O functions handling the communication between monitor and system. The `get_input` function models the reception of input data:

Prelude



```
struct Memory { ... }
impl Memory { ... }
[[ Evaluation Functions ]]
fn get_input() -> Option<(T_{s_1},...,T_{s_ℓ})> {
  [[ Communicate with system ]]
}
fn emit(output: &(T_{s_1},...,T_{s_n})) {
  [[ Communicate with system ]]
}
```
Prelude

```
fn prefix(mem: &mut Memory) -> bool {
  if let Some(input) = get_input() {
    mem.add_input(&input);
      [[ Evaluation Logic ]]
    } else {
      return true // Jump to Postfix.
    }
    [[ Repeat η_Φ^← times. ]]
    false // Continue with Monitor Loop.
}
```
Prefix

```
fn postfix(mem &Memory) {
  [[ Evaluation Logic ]]
  [[ Repeat η_Φ^→ times. ]]
}
```
Postfix

```
fn main() {
  let mut memory = Memory::new();
  let early_exit = prefix(&mem);
  if !early_exit {
    while let Some(input) = get_input()
    {
      mem.add_input(&input1);
      [[ Evaluation Logic ]]
    }
  }
  postfix(&mem);
}
```
Loop

Figure 3.11.: Overall structure of the generated Rust code.

`get_input() -> Option<(`$T_{\sigma^{\downarrow}_1},\dots,T_{\sigma^{\downarrow}_{n^{\downarrow}}}$`)>,`

Here, $T_{\sigma^{\downarrow}_1},\dots,T_{\sigma^{\downarrow}_{n^{\downarrow}}}$ are the types of all input streams. The function produces `None` if the execution of the system under scrutiny terminated, or `Some(v)`, where v is an $n^{\downarrow}$-tuple containing the latest input values.

The counterpart responsible for dispensing information is the `emit` function.

`emit(&(`$T_{\sigma_{n^{\downarrow}+1}},\dots,T_{\sigma_{n^{\downarrow}+n^{\uparrow}+n^{!}}}$`))`

It conveys the output values, including triggers, to the system.

Additionally, there are evaluation functions for each stream in several variants depending on whether the functions will be called in the prefix, the monitor loop, or the postfix. The implementations of the function variants differ only in the logic accessing other streams. The static analysis of the specification reveals which accesses fail or succeed in which phase of the execution. This alleviates the need for dynamic checks when providing several implementations. Hence, in lieu of a realization with a smaller code size, the compilation opts for more efficient realizations of the monitor logic. The translations themselves are straight forward since Lola expressions are syntactically and semantically similar to Rust expressions, barring stream accesses. These translate naturally to accesses to the working memory.

This working memory is a struct aptly named `Memory`. It consists of a static array for each stream in the specification and reads as follows:

```
struct Memory { $\sigma_1$: [$T_{\sigma_1}$; $\mu\sigma_1$], ... , $\sigma_{n^\downarrow+n^\uparrow}$: [$T_{\sigma_{n^\downarrow+n^\uparrow}}$; $\mu\sigma_{n^\downarrow+n^\uparrow}$] }
```

Note that this does not reserve memory for triggers since these cannot be accessed by other streams. The monitor allocates `Memory` once in its `main` function, keeps it on the stack, and grants read access to functions evaluating stream expressions. These functions also return their results rather than committing it into memory. Leaving this task to the main function makes no difference when executing the monitor sequentially but already prepares the concurrent evaluation presented later.

Execution Prefix

Execution Prefix

The *prefix* consists of $\eta_{\Phi}^{\leftarrow}$ conditional blocks, each processing an input event of the system under scrutiny. If the system terminates before the prefix concludes, the function returns `true`, indicating an early termination. This prompts the `main` function to initiate the postfix. Otherwise, the input is added to the working memory and, evaluation layer by evaluation layer, each output stream is evaluated in a dedicated function. For this, suppose the specification has λ^* evaluation layers and λ_i^* denotes the number of streams within evaluation layer $i \leqslant \lambda^*$, i.e.,

$$\lambda^* = \max\{\mathrm{Layer}(\sigma)\}$$

$$\lambda_i^* = |\{\sigma \mid \mathrm{Layer}(\sigma) = i\}|$$

Lastly, let $\sigma_{i,j} \prec^+ \sigma_{i,j+1}$ with $\mathrm{Layer}(\sigma_{i,j}) = \mathrm{Layer}(\sigma_{i,j+1}) = i$. Then, the code for the evaluation looks as follows:

```
let v_$\sigma_{1,1}$ = eval_pre_1_$\sigma_{1,1}$(&memory);
...
let v_$\sigma_{1,\lambda_1^*}$ = eval_pre_1_$\sigma_{1,\lambda_1^*}$(&memory);
memory.write_layer_1(v_$\sigma_{1,1}$,...,v_$\sigma_{1,\lambda_1^*}$)
...
let v_$\sigma_{\lambda^*,1}$ = eval_pre_$\sigma_{\lambda^*,1}$(&memory);
...
let v_$\sigma_{\lambda^*,\lambda_{\lambda^*}^*}$ = eval_pre_$\sigma_{\lambda^*,\lambda_{\lambda^*}^*}$(&memory);
Memory.write_layer_$\lambda^*$(v_$\sigma_{\lambda^*,1}$,...,v_$\sigma_{\lambda^*,\lambda_{\lambda^*}^*}$);
if v_$\sigma_{t_1}$ { emit($m_{t_1}$) } // Emit a warning when a trigger is active.
```

Note that, as indicated in the prelude, each conditional block calls a different set of evaluation functions. This allows for a fine-grained treatment of stream accesses, improving the overall performance at the cost of greater code size. Also, the call passes a single argument to the evaluation function: an immutable reference to `Memory`. As a result, the Rust type system guarantees that the evaluation does not mutate its state. The function returns a value that is committed to memory after fully evaluating the current layer. The bodies of these functions are straight-forward translations of stream expressions: each arithmetic and logical expression has a counterpart in Rust. Stream lookups simply read the respective entries in the working memory.

The functions `write_layer_i` used in the code snippet writes computed stream values to `memory`. After $\mu(s)$ iterations, the memory evicts the oldest data point for stream s, thus constituting a ring buffer.

Monitor Loop

Monitor Loop

The main difference between the *monitor loop* and the prefix is that, as the name indicates, the former consists of a loop rather than a sequence of conditionals. The monitor loop terminates as soon as the system ceases to produce new inputs. At this point, it transitions to the execution postfix.

Within the loop, the monitor proceeds just as in the prefix except that the evaluation functions are agnostic to the current iteration number. In the evaluation, all stream accesses are guaranteed to succeed rendering it free of conditionals except when the stream expression itself contains one. This is a performance boon since conditionals are expensive compared to arithmetic operations.

Execution Postfix

Execution Postfix

The structure of the *execution postfix* closely resembles the prefix with few differences: First, rather than a sequence of conditional evaluations, the postfix executes the evaluation functions unconditionally with no option for early termination. Second, it calls a different set of evaluation functions specifically tailored to the postfix.

Code Characteristics

The generated code exhibits two favorable characteristics. First, accepting an increase in code size and compilation time by quasi-duplicating the evaluation functions leads to an excellent runtime performance because it avoids conditional statements. Moreover, the functions require few arguments and utilize data locality. This is further emphasized by the lack of dynamic memory allocation and usage of native data types. Second, the clear code structure, especially w.r.t. memory accesses, drastically simplifies reasoning about the correctness of the code, as will come in handy in the next step.

3.2.4. Verification

The goal is to automatically prove that the verdicts produced by the monitor correspond to the formal semantics of Lola. Here, the main challenge is that the evaluation model refers to unbounded data sequences, agnostic of memory concerns. In contrast, the implementation allocates and manages a finite amount of memory for the monitoring process. As a result, the Lola semantics argues about data values long after they have been discarded in the implementation. This renders the exact relation between the

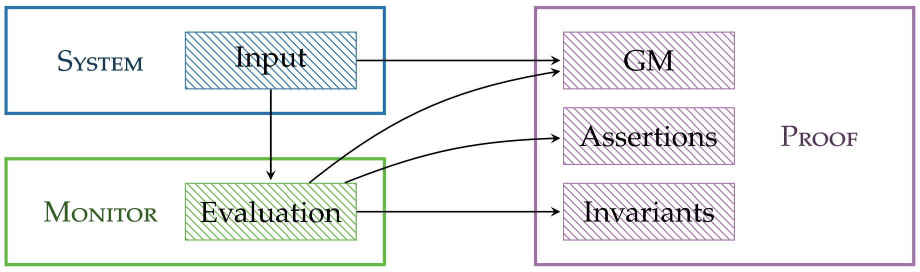


Figure 3.12.: Information flow between monitor and ghost memory.

memory content and the evaluation model, and thus the correctness of the computation, no longer immediately apparent.

The compilation solves this problem with the classic proof technique of introducing *ghost memory*. To this end, it introduces another data structure named `GhostMemory` (GM). Internally, the GM is realized as a wrapper for Rust vectors, i.e., dynamically growing sequences of data. Whenever the monitor receives or computes any data, it commits it to the ghost memory. The ghost memory's size thus obviously exceeds any bound, voiding the memory guarantees. However, the sole purpose of this act is to aid the verification without affecting the monitoring process: Information flows from the program into the ghost memory and into the proof, but remains strictly separated from the stream evaluation. This allows for removing the ghost memory after successful verification of the correctness of the monitor without altering its behavior. Figure 3.12 illustrates the flow of information between the monitor and the ghost memory. Clearly, removal of proof artifacts does not affect the monitor.

Ghost Memory

The correctness proof has two major obligations: proving compliance between values in the ghost memory and the working memory, and proving the correctness of the trigger evaluations w.r.t. the ghost memory. These obligations are encoded as verification annotations, such that the Viper [MSS16] toolkit verifies them automatically. The generation of additional annotation allows for guiding the verification processes, increasing feasibility.

Viper Annotation Generation

Viper annotations fall into the following categories:

Function Contracts. An annotation in front of a function `f` consists of preconditions as well as guarantees. Viper imposes constraints on both the function caller and the function body itself. Each call to `f` is replaced by an assertion of its preconditions, prompting Viper to prove their validity, and an assumption of the guarantees. In a separate step, Viper assumes that the preconditions are met and verifies that the guarantees hold after executing the function body. Note that the Rust type system already ensures that references passed to the function `f` are accessible and cannot be modified or freed unless they are explicitly declared mutable.

Loop Invariants. Viper analyzes while-loops, similarly to functions, in three steps. First, it verifies that the code leading to the loop satisfies the invariants of the while-loop. Second, it assumes that both the loop invariant and the loop condition hold and verifies that the invariant again holds after the execution of the body, while it may be violated in the interim. Lastly, Viper assumes that the invariant and the negation of the loop condition hold for the code following the loop.

Inline Assertions. Function contracts and loop invariants impose implicit assertions on the code. Viper also allows for supplementing explicit assertions using the Rust macro `assert!`. Usually, the macro checks the respective condition during runtime.[9] However, Viper eliminates the need for this dynamic check as it verifies the correctness statically and transforms it into an assumption for the remainder of the verification. Thus, the assertions serve a similar function as the ghost memory: they are a proof construct and do not influence the monitor per se (cf. Figure 3.12).

The compilation inserts annotations at several key locations. First, as an example for function annotations, consider a function that retrieves a value of the stream σ from the working memory. The function takes the relative index of the retrieved value as single argument. An index of 1, for instance, accesses the second-to-latest value. The annotation requires that the index must not exceed the memory reserved for σ. Syntactically, this results in the following annotation in front of the function head:

```
#[requires="index < μ(σ)"]
```

Moreover, the function needs to guarantee that the return value corresponds to the respective value stored in memory. This is expressed by the following annotation for each $0 \leqslant i \leqslant \mu(s)$.

```
#[ensures="index == i ==> result == self.s[i]"]
```

Here, `self` refers to the memory struct. The remaining function annotations follow a similar pattern, i.e., they require valid arguments and ensure correct outputs as well as the absence of undesired changes.

Note that the ghost memory is essentially a wrapper for Rust vectors as they represent a growing list of values. Thus, functions concerning the ghost memory carry the standard annotation ensuring correctness of the vector as presented in the Viper examples.[10]

Next, the loop has several entry checks that are expressed as inline assertions. These ensure that the iteration count is $\eta_\Phi^{\leftarrow}$ and that the length of the ghost memory for a stream σ is $\eta_\Phi^{\leftarrow} - \Delta(\sigma)$. This is necessary because the loop invariant asserts equivalence between an excerpt of the ghost memory and the working memory, so the excerpt needs

[9]Note that in some languages, assert statements are no longer present after compilation in the release configuration. For Rust, this is not the case; compilation merely purges `debug_assert!` macros.

[10]See for example the verified solution for the Knight's Tour Problem: `https://github.com/viperproject/prusti-dev/blob/master/prusti-tests/tests/verify/pass/rosetta/Knights_tour.rs`; last accessed: 02.02.2022.

to be present and of the proper length. While this is guaranteed for the working memory due to the static allocation, it needs to be asserted for the dynamically growing ghost memory. Hence, the compilation adds the entry checks.

In terms of memory equivalence, it remains to be shown that all values in the working memory correspond to the respective entry in the ghost memory. Formally, let m be the working memory and let g be the ghost memory where index 0 marks the latest value. Furthermore, let η be the current iteration count. Then, the invariant checks:

$$\forall \sigma \colon \forall i \colon (0 \leqslant i < \mu(\sigma) \implies m_\sigma[i] = g_\sigma[i]). \tag{3.3}$$

At loop entry, $\mu(\sigma) = \eta_{\Phi}^{\leftarrow} - \Delta(\sigma) = \eta - \Delta(\sigma)$ is the number of iterations in which a value for σ was computed. In each further iteration of the loop, the invariant checks that the former $\mu(\sigma) - 1$ entries remained the same and that the new values in the ghost memory g and the working memory m are equal. The first of these checks is not strictly necessary for the proof because it immediately follows from the function contracts of the helper functions. However, after completing one loop iteration, Viper invalidates prior knowledge about all variables that were mutated in the loop. Further reasoning about these variables is thus solely based on the loop invariants.

To express Equation (3.3) in Viper, the compilation needs to statically resolve the universal quantification over the streams. Thus, for each stream σ, the compilation generates the following annotation:

```
#[invariant="forall i: usize :: (0 <= i && i < μ(σ)) ==> mem.get_σ(i) == gm.get_σ(iter − 1)"]
```

Here, `iter` is a variable denoting the current iteration, `mem` is the working memory, and `gm` is the ghost memory. Viper can handle the remaining universal quantification over i. However, the compilation reduces the verification effort further by unrolling it. This is possible since the memory requirement $\mu(s)$ of a stream s is determined statically.

Lastly, the compilation introduces inline assertions after the evaluation of stream expressions, i.e., in the prefix, loop body, and postfix. These annotations show that computed values are correct when assuming that the values retrieved from the working memory are correct as well. This argument is well-founded because the compilation substitutes failing stream accesses with their respective default values. Thus, any value that is retrieved from `Memory` was computed in an earlier iteration or layer and is therefore proven correct by Viper.

It only remains to be shown that the stream expression is properly evaluated. Expressions consist of arithmetic or logical functions, constants, and stream accesses. The former two can be trivially represented in Viper. Since the memory is assumed to be correct and failing accesses are substituted by constants when possible, accesses also translate naturally into Viper.

Conclusion: Annotation Generation. The validity of the assertions after performing the evaluation logic shows that newly computed values are correct if the values in the working memory m and the ghost memory g coincide. This fact is guaranteed by the loop invariant. Furthermore, the inductive argument of the loop invariants shows that $m_\sigma[i] = g_\sigma[i]$ holds for all streams σ and for all $i \leqslant \eta$ if m were to never discard values. Thus, m is a real subsequence of g, which is a perfect reflection of the evaluation model. As a result, any trigger violation detected by the monitor realization corresponds to a violation in the evaluation model for the same sequence of input values; The realization is verifiably correct.

3.2.5. Concurrent Stream Evaluation

The hardware realization proved that a concurrent evaluation of independent stream has the potential to significantly boost performance of the monitor. The following section devises an analysis of Lola specifications that enables safe parallelization.

To this end, there are two notable characteristics of Lola: the computation of a stream expression can only *read* memory of other streams, and inter-stream dependencies are determined statically. The evaluation layers are a manifestation of the second observation as they group streams incomparable according to the evaluation order. Combined with the first observation it follows that all streams within one layer may be computed in parallel. Thus, the compilation spawns a new thread for each stream within the layer with read access to the global memory. Adding further annotations to the code enables Viper to verify the correctness of the parallel execution as well.

The compilation capitalizes on Rust's concurrency capabilities by evaluating different output streams in parallel. Here, a major advantage of Rust is that its ownership model enforces a strict separation of mutable and immutable data. Any data point has exactly one owner who can transfer ownership for good or let other functions borrow the data. Borrowing data is again either mutable or immutable. If a function mutably borrows data, it — and it alone — can read or write this data. Conversely, if a function immutably borrows data, other functions and the owner can read, but not write it. A consequence of this fine-grained access management with static enforcement is that enabling concurrency becomes rather easy when compared to languages like C.

Evaluation Logic

Enabling the concurrent evaluation of streams requires slight changes in the code generation. First, in the setup of the monitor, it creates two *one-way, multi-producer, single consumer channels*. These channels allow multiple entities to send data through a channel in a non-blocking fashion. Even though it sounds somewhat counter-intuitive, the channel also allows several consumers to blockingly wait on it the same time. However, each message sent will reach exactly one randomly selected consumer.

MPSC Channels

These channels lay the foundation for a thread pool, so the monitor then creates Ξ threads with $\Xi = \max\{\lambda_i^* \mid i \leqslant \lambda^*\}$, i.e., the greatest width of a single layer. A consequence of this choice is that when evaluating streams layer by layer, each thread generally has to evaluate only one stream expression.[11] However, note that scheduling can distort this outcome if one thread completes its task before another one receives one. In any case, each thread gets access to the receiving end of one of the channels (called rx_c for "receive command") and the sending end of the other (called tx_r for "transmit result"). The setup phase thus gets the following addition:

```
let (tx_c, rx_c) = crossbeam::channel();
let (tx_r, rx_r) = crossbeam::channel();
(0..ζ).for_each(||
  let recv = rx_c.clone();
  let send = tx_r.clone();
  thread::spawn(move || eval_thread(recv, send);
)
```

Upon creation, each thread immediately starts to listen to rx_c, blocking themselves until the monitor starts receiving input data.

When an event finally reaches the monitor, the main thread initiates the evaluation through the thread pool, which runs layer by layer. For this, it sends commands through tx_c ("transmit command"), where each command is the ID of a stream of the current layer. An arbitrary thread receives the ID of a stream σ, prompting it to evaluate its stream expression, resulting in a value v. It then sends the pair of v and the ID of σ over tx_r ("transmit result"), and returns to waiting on rx_c. This results in the following code for the `eval_thread` function:

```
fn eval_thread(rx_c: Sender<Id>, tx_r: Receiver<(Id, Value)>) {
  rx_c.for_each(|id|
    let v = /* evaluation logic */
    tx_r.send((id, v));
  );
}
```

After sending all commands, the main thread starts to listen to rx_r. Whenever receiving (v, i), it updates its working memory by writing v for the stream with ID i, potentially evicting other data. Note that this write-operation does not affect the concurrent evaluation of other threads since, by definition of layers, the respective computations operate on independent slices of the working memory. When all results are received and committed to memory, the process continues with the next layer. Hence, the main monitor function requires the following code for each layer i:

```
(0..λ_i).for_each(|ix| tx_c.send(ids[ix])
(0..λ_i).for_each(|_|
    let (id, v) = rx_r.recv();
    mem.write(id, v);
)
```

Note that in all code snippets, the error handling and slicing of memory is omitted for illustration. The full code is available in Appendix A.2.

→Appx. A.2, p. 214

[11]This value can be reduced arbitrarily without compromising correctness. In particular, it should not exceed the number of cores available on the machine.

Remark 3.4 (Crossbeam)**.** The generated code uses the Rust library crossbeam, the de facto standard for concurrency. A similar result can be achieved without external code by moving the global memory to the heap and using the standard Rust thread logic. Diving deeper into the technicalities of Rust, its type system requires the programmer to guarantee that the global memory will not be dropped until all threads terminate. Thus, the memory needs to be wrapped into an *Atomically Reference Counted (Arc)* pointer. This has two disadvantages: all accesses to memory require generally slower heap access and the evaluation suffers from the overhead accompanying atomic reference counting. Hence, crossbeam is preferred.

Verification

The verification of the new evaluation logic requires all evaluation functions to be annotated with `#[pure]`. This indicates that a function mutates naught but its local stack portion.

The correctness of this approach is an immediate consequence of the correctness of the evaluation order and memory locality of streams. In particular, the independence of streams within the same evaluation layer and the pureness of the functions are crucial. The latter ensures that the function does not mutate anything outside its local stack. The former ensures that using pure evaluation functions within the same layer is indeed possible. Thus, the order of execution cannot change the outcome of the function, enabling the concurrent evaluation.

3.2.6. Experimental Evaluation

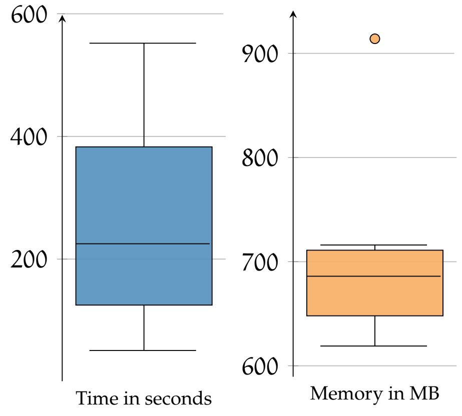
→Sec. 2.6, p. 82

The compiler uses the RTLola frontend implementation (Section 2.6). The verification of the output uses the Rust-frontend of the Viper [MSS16] framework called Prusti [Ast+19]. Prusti translates the Rust program into the Viper intermediate verification language, followed by a translation into an SMT model, which is checked by the SMT solver Z3 [MB08]. In combination, the toolchain enables completely automatic proof checking.

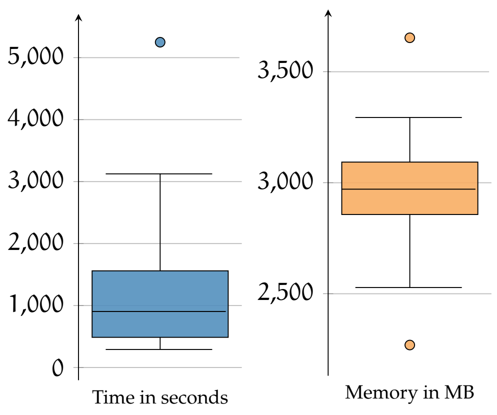
The experiments validate the approach with respect to three success indicators: First, compiling Lola specifications of varying sizes from literature and verifying the generated annotated rust code determines the scalability with respect to the size and complexity of the specification. Second, comparing the performance in terms of running time of the generated monitors against the RTLola interpreter measures the efficiency of the monitor. Third, comparing the running time performance of a sequential and a concurrent monitor for the same specification indicates the impact of the concurrent evaluation.

Monitor Generation and Verification

The experiments were conducted on a 3.1 GHz Dual-Core Intel i5 machine with 16 GB of RAM. In all experiments, the compilation including the parsing and analysis of



(a) Altitude Monitor.



(b) Network Traffic Monitor.

Figure 3.13.: Results of 20 verification runs for the running example specification (➔ Example 3.3, p. 129) and a network monitoring specification (➔ Listing 3.1, p. 124). Blue marks the running time in seconds, orange the memory consumption in MB.

the specification via the RTLola frontend has a negligible running time of under ten milliseconds and total memory consumption of less than 4 MB. As expected, the verification of the annotated rust code using Prusti and the Viper toolkit requires significant time and memory. While the translation into the SMT model is deterministic and can be parallelized, the verification with Z3 exhibits generally high and unpredictable running time.

Out of the three specifications of varying size, the compilation and verification worked flawlessly on two. The third one occasionally ran into timeouts and inconclusive verification results.

First, consider the running example specification from Example 3.3. The results both in terms of running time on the left and memory consumption on the right for 20 runs are depicted in Figure 3.13a. The plot shows that the running time never exceeds 600 s with a median of 225 s. The memory consumption is noticeably more stable ranging between 648 MB and 711 MB with one outlier of 914 MB. ➔Ex. 3.3, p. 129

Evidently, the first specification is short and illustrative, so the second one is more practically relevant. It is a Lola adaptation of the network traffic specification from Listing 3.1. Recall that it monitors the network traffic of a server based on the source and destination IP of requests, TCP flags, and the length of the payload. The full, adapted specification can be found in Listing A.1. Figure 3.13b shows the results both in terms of running time on the left and memory consumption on the right for 20 runs. The increase in resource consumption clearly reflects the increase in complexity and size of the input specification. While the longest run took nearly 90 min, most of the runs took ➔Listing 3.1, p. 124 ➔Listing A.1, p. 220

less than 25 min with a median of roughly 15 min. Like before, the memory consumption is relatively stable ranging around 3 GB.

The last specification showcases the limitations of the approach. It is a Lola adaptation of a drone flight phases detection presented in [Ado+17]. The monitor is supposed to raise an alarm if actual velocity and a reference velocity provided by the flight controller deviate strongly. The details of the specification are not relevant except for one line.

```
input time_s, time_micros, velo_x, velo_y, velo_r_x, velo_r_y: Int32

output time := time_s + time_micros / 1000000
output count := count.last(or: 0) + 1
output frequency := 1 / δ(time)
output freq_sum := frequency + freq_sum.last(or: 0)
output freq_avg := freq_sum / count
output velo: Int32 := sqrt(vel_x*vel_x + vel_y*vel_y)
output velo_max: Int32 :=
    if res_max.last(or: false) then velo else max(velo_max.last(or: 0), velo)
output velo_min: Int32 :=
    if res_max.last(or: false) then velo else min(velo_min.last(or: 0), velo)
output res_max: Bool := (velo_max − velo_min) > 1
output unchanged: Int32 :=
    if res_max.last(or: false) then 0 else unchanged.last(or: 0) + 1
output velo_dev: Int32 := velo_r_x − velo_x + velo_r_y − velo_y
output worst_dev: Int32 :=
    if unchanged > 15 then velo_dev else max(velo_dev, worst_dev.last(or: −10))

trigger freq_avg < 10 "Low input frequency."
trigger velo_dev > 10 "Deviation between velocities too high."
trigger worst_dev > 20 "Worst velocity deviation too high."
```

While the specification looks alright and the compilation worked without difficulties, the verification revealed arithmetic errors in the original specification. They arose from division in which the denominator was an input stream access: `δ(time)`. Recall that this desugars into a subtraction of the last time from the current one. Since neither Viper, nor the system can guarantee that this value is necessarily non-zero, Viper reported a potential violation of a verification annotation. Hence, the approach was able to detect flaws in specifications stemming from implicit assumptions on the system. These assumptions might not hold during runtime, causing the monitor to fail.

When modifying the specification to work without an unsafe division, still only four of the 10 runs terminated successfully. Here, a successful run shows that the approach succeeded in verifying the monitor realizations of a large and arithmetically challenging Lola specifications. The running time varies between 6 min and 16 min with a memory consumption of between 1.38 GB and 1.66 GB. Another two runs did not terminate within three hours. The reason lies within the underlying SMT solver: an unfavorable path choice in the solving procedure can result in vastly greater running times. Lastly, four more runs reported a potentially violated assertion or crashed internally. Once again, an unfavorable choice lead to an over-approximation prohibiting Viper from completing the verification.

In summary, the results for the last specification were mixed. While restarting the verification procedure can lead to finding a successful run, the incident comes to emphasize the reliance of the approach on external tools. As a result, the applicability increases with advances in research on automated proof checking of annotated code. This constitutes another reason for the continued development of such valuable tools as Prusti and generally the Viper toolkit.

Performance of Generated Monitors

This sequence of experiments harbors no surprises: As expected, the compiled monitors exhibit superior running time when compared against the RTLola interpreter from Section 2.7.6. The comparison is based on randomly generated input data for the running example and network traffic monitoring specification. The former was adapted to be compliant with RTLola, i.e., rather than accessing the input with a future offset, the specification uses a negative offset of −2. For the first specification, the interpreter required 438 ns per event on average out of 10 runs, whereas the compiled version took 6.2 ns. This amounts to a running time reduction of 98.6%, i.e., the compiled version is 7,142.86% as fast. The second, more involved specification shows similar results: On average, the interpreter needed 1.535 µs per event, and the compiled version 63.4 ns, so the running time is reduced by 96.9%.

→Sec. 2.7.6, p. 88

Sequential vs. Parallel Evaluation

For this comparison, note that the performance gain of the parallel execution of the evaluation of independent streams strongly depends on three factors: the number of available cores, the performance of inter-thread communication, and the extent to which a specification allows for parallel evaluation. The former two factors are machine dependent.

The effect of the latter becomes apparent when comparing the performance of the sequential and the parallel stream evaluation on different specifications that vary in the size of the evaluation layers. To this end, the experiments use a specification inducing a dependency graph with three layers of the same size. Every stream has a deliberately expensive stream expression containing multiple computations of the haversine formula. Without such expensive stream expressions, the overhead of the inter-thread communication in the parallel evaluation dominated the performance benefit. Other examples for expensive and realistic functions are aggregations of multiple stream values or short-time Fourier transformations. Figure 3.14 summarizes the results. Each data point corresponds to the median of ten executions of the same specification and a randomly generated input trace. Here, each specification uses the same 10 randomly generated input sequences. However, the input values themselves do not have any

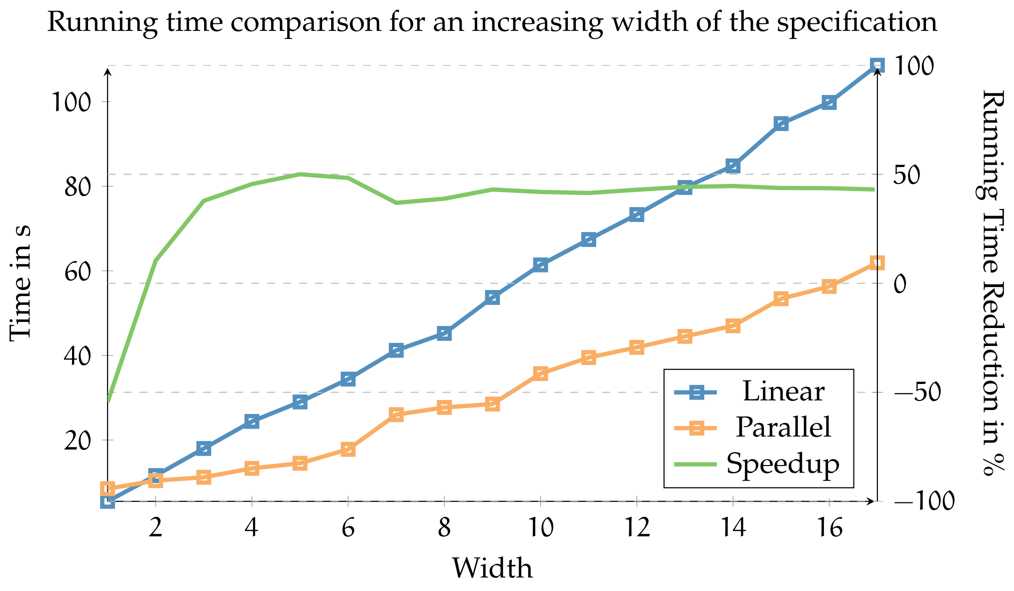


Figure 3.14.: Running time comparison of the sequential and the parallel execution. The x-axis represents the width of the specification, i.e., the size of the evaluation layers. This is an indicator for the degree to which a specification allows for parallel evaluation.

impact on the evaluation time. Clearly, the parallel stream evaluation dominates the sequential one. When for a width of 1 the parallel execution incurred a performance penalty of more than 50% due to the additional setup required, the performance quickly hones in on a gain of 50%. Since the experiments were conducted on a dual-core machine, this is the expected performance peak.

Second, consider specifications, where every stream expression again contains multiple computations of the haversine formula as a stand-in for an expensive computation. In contrast to the first line of experiments, the width of the specification remains constant with three layers of size eight each. However, the number of computations of the haversine formula per stream varies.

Figure 3.15 shows similar results to the former experiment. Again, each data point corresponds to the median of ten executions of the same specification and a randomly generated input trace. The parallel stream evaluation clearly dominates the sequential one and the performance gain settles on 50% as expected. This gain is stable throughout the entire experiments because the smallest specification with 100 iterations coincides with the one used in the first line of experiments with width 8. Hence, it was already quite complex, compensating for the ramp-up time.

The evaluation clearly shows that the parallel execution *can* make a difference. However, this is only the case if the specification is sufficiently wide and contains resource-heavy computations as can be seen for the specification with a width 1. Hence, parallel

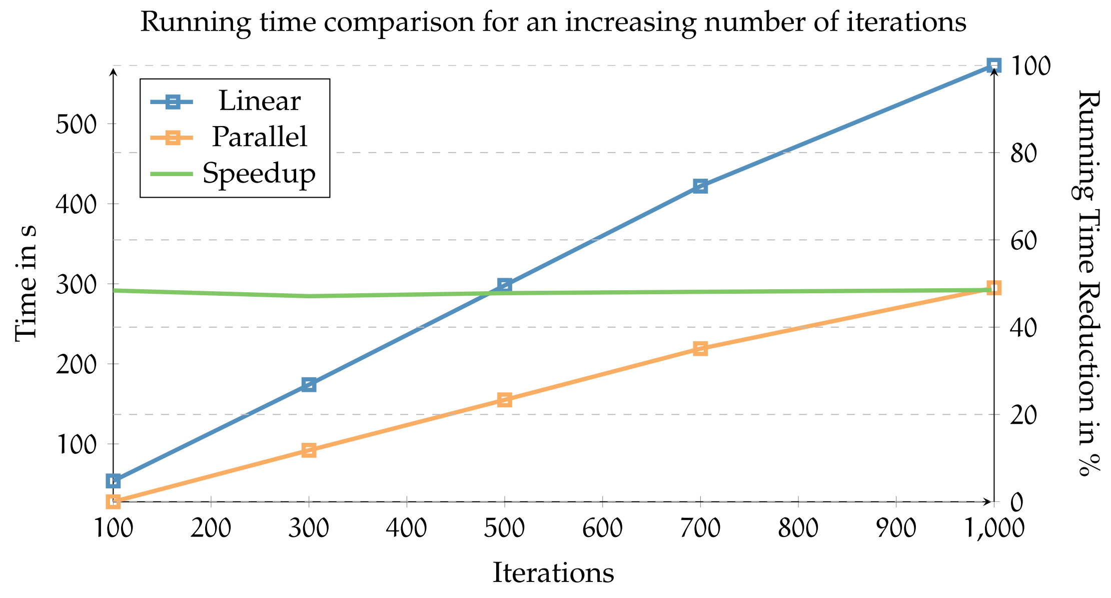


Figure 3.15.: Running time comparison of the sequential and the parallel execution. The x-axis represents the number of times a stream expression contains a haversine function call. This serves as indicator for the complexity of the evaluation. The specifications induce a dependency graph with three layers each of which contains eight streams.

evaluation does not generally perform better. Whether it will benefit or detriment performance thus highly depends on the specification at hand.

3.2.7. Recapitulation

This section presented a compilation of Lola specifications into Rust code. Using Rust as the compilation target allows for fine-grained control, yields highly performant executables, which can be used directly on many embedded platforms. The generated code contains annotations enabling verification of the code using the Viper framework. With guiding assertions in the code, as well as function contracts and loop invariants, Viper can verify monitors even for large specifications. Furthermore, the compilation allows for generating monitoring which evaluate independent streams concurrently. The experimental evaluation shows the significant performance gain achievable via parallel evaluation provided the specification is sufficiently wide and/or complex. For other specifications, the overhead required for inter-thread communication dominates the performance benefit.

The results are promising and encourage further research in this direction. For instance, compiling RTLola enables the generation of verified monitors for even more realistic cyber-physical systems.

3.2.8. Related Work

The result presented in this section is part of a line of research aiming for verifying compilers, a problem Tony Hoare declared a grand challenge in computer science [Hoa03]. Other milestones in this direction are the concept of proof-carrying code (PCC) and certifying compilers [NL98b].

In PCC [Nec97; App01; Nec02] architectures, executable code contains additional information. This information serves as proof of several significant properties of the executable itself. As a result, a host can automatically verify compliance of the code with its own security policy. A positive result indicates that the execution is safe [NL98a]. Such architectures exist both for general purpose languages like Java [CLN00] and low-level languages [AF00]. Abstraction-carrying code [Her+05; BJP06] is a variation thereof, developed for constraint logic programs. Here, the artifact embedded in the code is the result of an abstract interpretation, i.e., a fix point that serves as certificate for invariants.

Certifying compilers [NL98b] are closely related to the approach presented for RTLola. The similarity is that certifying compilers opt for checking correctness of the result of the translation rather than the source. However, this is usually an executable, whereas for RTLola it is high-level Rust code. While there is no certifying compiler for Rust, yet, there are first endeavors in this direction in the shape of the Ferrocene[12] sub-language of Rust. Moreover, there are such compilers for Java [Col+00], a subset of C [Li+10], and concepts for incorporating pointer logic [Che+07] and zero-knowledge proofs [Alm+10].

A lot of research on verifying compilers is dedicated to widely used programming languages or concepts. Work related to runtime monitoring is comparatively spares. The Copilot [Pik+10] toolchain, for example, compiles specifications into constant memory and constant time C realizations. The result is verified via the CBMC model checker [CKL04]. Hence, the setup is extremely similar to the one presented here with the main difference that the scope of the verification of Copilot is limited to the absence of various (pointer-) arithmetical errors. Note that CBMC can verify arbitrary inline assertions, however, Copilot does not generate them. Yet, Copilot is more expressive than Lola as it can express real-time properties. Moreover, there is first work in the direction of enriching compilations of RTLola specification with verification-related annotations. Baumeister extended the FPGA translation presented in the preceding section by injective traceability annotations into the hardware description [Bau20]. However, this work is still in its early stages.

Further, there is significant work in the direction of verifying compilers specifically for runtime monitoring. The two main differences are the target languages and the machinery used for the verification. First, the target languages are specification logics such as STL [WS20], metric first-order temporal [Sch+19] or metric first-order dynamic logic [Bas+20], and MTL with quantitative semantic [CM20], which come with all the

[12] `https://ferrous-systems.com/ferrocene/`; last accessed: 01.02.2022

drawbacks detailed in Section 2.9. However, their controlled nature provide a stronger foundation for any kind of verification. Second, their underlying proof engines are interactive theorem provers. The main difference is that the approach presented here is fully automatic. In contrast, as the name already suggests, interactive theorem provers interact with users during the verification process. While the tool provides valuable assistance, users have to manually lead the proof in the right direction. Hence, these users have to be domain expert and adept in logical reasoning. This renders the approach more resource-intense. Yet, combination of assistance and expertise allows for the verification of complex properties, far exceeding the capabilities of fully automatic process.

→Sec. 2.9, p. 93

Most notably here are the VeriPhy toolchain and ModelPlex. They are both based on differential dynamic logic [Pla08], a logic specifically designed to capture the complex hybrid dynamics of cyber-physical systems. ModelPlex [MP16] translates a specification verified with the KeYmaera X [Ful+15] theorem prover into three verified components. These components monitor whether the environment complies with the assumed model in the underlying specification, whether the controller behaves correctly, and the safety of a predicted future state based on the model. VeriPhy [Boh+18] also builds upon the premise of a verified specification. It then triggers a chain of translation steps, the correctness of which hinges on the initial proof. They successively generate lower-level artifacts down to executable code. This code is then correct since all previous steps are correct based on the initial proof.

Chapter 4

Conservative Model Generation

A common mathematical description of CPS are *hybrid automata*. They combine information regarding their discrete control structure and continuous physical behavior. This enables a host of analysis options throughout the development process such as verification of critical properties pre-deployment, identification of anomalous behavior during runtime and a postmortem analysis based on recorded flight data. Albeit indisputably beneficial for the development process, designing a hybrid automaton to properly reflect the semantics of the system is a delicate process. Thus, unsurprisingly, several approaches aim at automatically constructing either hybrid automata or their simpler cousins, timed automata, based on execution traces of the system. These traces are usually a development artifact as they get recorded during test runs. As such, not only do they contain information regarding the continuous behavior of the system, they are also expected to satisfy relevant coverage criteria. Hence, they cover both the "average", expected behavior including initialization and termination, plus the extreme and corner case behavior. This renders them particularly interesting for such a construction.

Hybrid Automata

Estimation methods, most prominently machine-learning, yield promising results in terms of reconstructing the correct discrete structure of the automaton and approximating the continuous dynamics. Their great accuracy notwithstanding, the *direction* of the approximation is unclear, so parts of the results might be over-, while others are under-approximations. This suffices for conveying the gist of the system, yet it limits its capabilities in terms of safety-critical analyses. For this reason, this chapter shows how another development artifact — an RTLola specification — enables an alternative construction. This construction yields a *conservative hybrid automaton*, i.e., a guaranteed over-approximation of the original system. The main contributing factor is that the specification covers the entire execution. To this end, the monitor — and by proxy the specification — needs to keep track of different operational phases such as the takeoff or landing phase to judge the situation accurately. Extracting this information about the

Conservative Hybrid Automaton

operational phases grants a valuable starting point for the discrete control structure of the hybrid automaton.

Specification Automaton

Hence, the construction algorithm first extracts the *implicit discrete control structure* from the specification, resulting in a strong over-approximation of the system. It then proceeds by analyzing the evolution of samples over time, and using this information to refine the automaton. Due to a lack of generalization, the resulting automaton is an under-approximation. To compensate for this, the last step of the algorithm merges control modes within the automaton to finally obtain an over-approximation of the original system. This mixture of a top-down and bottom-up construction results in an automaton that is both a provable over-approximation and retains a high level of precision.

Apart from being conservative, the construction distinguishes itself from existing approaches in two major ways. First, the specification roughly indicates the general discrete structure of the constructed automaton. This alleviates the need to second-guess the structure in its entirety, reducing revisions to local sub-structures. This pushes scalability far beyond L^*-based approaches [Ang87] like Medhat et al. [Med+15] in which significant time is spent to determine the discrete structure. Secondly, the construction reduces the level of over-approximation by merging modes of an under-approximation only if needed. This can result in more fine-grained refinements than when successively widening dynamics until the language of the automaton encompasses every input trace [Sot+19].

An empirical evaluation validates three major claims. First, the construction requires few traces to produce decent results. For a fourteen-mode automaton, for example, seven hand-picked or on average 35 random traces suffice for a perfect reconstruction. Second, the precision — while not flawless — comes close to the optimal result provided the input data is adequate. Third, the construction algorithm scales extraordinarily well. Even large automata with over 1000 modes can be constructed within mere seconds. All three benefits are the result of relying on development artifacts in form of test traces and a runtime monitoring specification: a readily available resource often left under-utilized.

4.1. Preliminaries and Notation

This section introduces the basics of timed traces and hybrid automata in combination with the running example for this chapter. Hybrid automata were originally presented by Henzinger in 1995 [Hen96]. While numerous variations were discussed throughout the years, this chapter considers rectangular hybrid automata. Moreover, this section introduces special notation for handling and destructing vectors, traces, and automata.

4.1.1. Convex Geometry

Definition 4.1 (Rectangle)

Let $\langle \mathcal{I}_i \rangle_{1 \leqslant i < n}$ be a family of closed intervals over the real numbers. Each interval is a 1-dimensional *rectangle*. The multiplication of a k- and an ℓ-dimensional rectangle produces a $k+\ell$-dimensional rectangle. Addition and multiplication of a rectangle $\mathcal{I}$ with a scalar $\lambda \in \mathbb{R}$ are *geometric translation and scaling*, respectively.

Def. Rectangle

Def. Translation and Scaling

$$\lambda + \mathcal{I} = \lambda + \prod_{i=0}^{n} I_i = \lambda + \prod_{i=0}^{n} [\ell_i, u_i] = \prod_{i=0}^{n} [\lambda + \ell_i, \lambda + u_i]$$

$$\lambda \mathcal{I} = \lambda \prod_{i=0}^{n} I_i = \lambda \prod_{i=0}^{n} [\ell_i, u_i] = \prod_{i=0}^{n} [\lambda \ell_i, \lambda u_i]$$

The set $\mathcal{I} \subseteq \mathbb{R}^2$ is the set of one-dimensional rectangles over $\mathbb{R}$.

Definition 4.2 (Convex Hull)

Let $\mathcal{S}$ be a convex set. The *convex hull* of two convex sets $A \subseteq \mathcal{S}$ and $b \subseteq \mathcal{S}$ is the minimal convex set covering both A and B. For $\mathcal{S} = \mathbb{R}$, chull is defined as:

Def. Convex Hull

$$\text{chull}(A, B) = \{x \mid \exists a \in A, b \in B, \lambda \in [0,1].\, x = a + \lambda(b - a)\}$$

The convex hull of a set of convex sets is the iterative computation of the convex hull.

$$\text{chull}^+(\{A\}) = A$$

$$\text{chull}^+(\{A\} \dot\cup S) = \text{chull}(A, \text{chull}^+(S))$$

4.1.2. Hybrid Automata

An n-dimensional *multi-rectangular hybrid automaton* $\mathcal{H}$ is a 6-tuple $(M, \Lambda, \mathit{flow}, E, \gamma, s_I)$ over $\mathbb{R}^n$. Each constituent is finite with the following intuitive meaning:

Def. Rectangular Hybrid Automaton

Modes M denotes the set of discrete (control) modes.

(Initial) State A state $s \in M \times \mathbb{R}^n$ of the automaton consists of a discrete mode and an n-dimensional vector with valuations for each continuous (state) variable. The initial state $s_I = (\mu_I, x_I) \in M \times \mathbb{R}^n$ marks the mode and valuation of the starting point of the automaton.

Actions Λ denotes the set of action labels.

Dynamics The flow function *flow*: $M \to \mathcal{I}^n$ defines the dynamics of modes. When entering a mode, a random value is drawn from the respective interval for each dimension.

Edges $E \subseteq M \times \Lambda \times M$ denotes the set of edges containing discrete, labeled transitions between two modes.

Guard $\gamma \colon E \to \mathcal{I}^n$ assigns guard conditions to edges. A transition can only be taken in a state if its continuous component lies within the rectangle.

Note that this description forgoes a notion of mode invariants as they are irrelevant for the conservative construction.

Example 4.3 (Running Example)**.** Figure 4.1 shows an easy-to-grasp visual representation of a simplified hybrid automaton modeling an aircraft. The system starts in a takeoff mode and resides there until reaching cruising altitude, traveling north. Course adjustments are modeled as left or right curves. After some time, it attempts a landing. Under windy conditions, the descend-phase is elongated as a precaution. Each rectangle constitutes a mode with the first line being its name, hence:

$$M = \{\textsc{Takeoff}, \textsc{Left}, \textsc{Straight}, \textsc{Right}, \textsc{Landing}, \textsc{LandWindy}\}$$

The remaining lines define the dynamics of the mode for each dimension as differential inclusions. Evidently, the system is three-dimensional and the dynamics of the Straight mode is defined as:

$$\mathit{flow}(\textsc{Straight}) = \begin{pmatrix} [90, 300] \\ [0, 0] \\ [-2, 2] \end{pmatrix}$$

Arrows represent edges, so $e = (\textsc{Takeoff}, \mathit{cruise}, \textsc{Straight}) \in E$. Their labels are action labels and guard conditions, hence $\gamma(e) = z \geqslant 300$. Note that tautological guard conditions are omitted, so $\gamma(\textsc{Straight}, \mathit{turnL}, \textsc{Left}) = (-\infty, \infty)^3$. Lastly, the arrow pointing to Takeoff out of nowhere marks the mode as the initial one. Unless stated otherwise, the initial state is then the initial mode coupled with the zero-vector $\vec{0}$. △

Semantics

Hybrid automata allow for two kinds of transitions: control mode changes according to E and delays according to the *flow* of the current mode during which the system state evolves continuously. More formally, the semantics of a multi-rectangular hybrid automaton $\mathcal{H}$ are defined based on valid omniscient traces through $\mathcal{H}$.

Definition 4.4 (Valid Omniscient Timed Trace)
An omniscient trace $\tilde{\pi} \in \mathbb{R}^n \times \mathbb{R}^+ \times \mathbb{R}^n \times (E \times \mathbb{R}^+ \times \mathbb{R}^n)^k$ with

$$\tilde{\pi} = x_0, \delta_0, x_1, e_1, \delta_1, \ldots, x_k, e_k, \delta_k, x_{k+1}$$

is *valid* for an automaton ($\tilde{\pi} \triangleright \mathcal{H}$) iff:

Def. Valid Trace

1. The trace starts in the initial state of $\mathcal{H}$, i.e., $x_I = x_0$.
2. The first discrete transition starts in the initial mode, so $e_1 = (\mu_I, \lambda, \mu)$ for some λ and μ.
3. All guards are satisfied: $\forall 1 \leqslant i \leqslant k \colon x_i \in \gamma(e_i)$.
4. All delay transitions are valid, i.e., for $0 \leqslant i \leqslant k$ and $e_{i+1} = (\mu_s, \lambda, \mu_t)$, the state changes according to the flow: $x_{i+1} \in (x_i + \delta_i \cdot \mathit{flow}(\mu_s))$.

The *language* of an automaton is the set of valid traces: $\mathcal{L}(\mathcal{H}) = \{\tilde{\pi} \mid \tilde{\pi} \triangleright \mathcal{H}\}$.

Def. Automata Language

As the name already indicates, the notion of omniscient traces is fairly strong. Observable traces constitute their accessible, weaker cousin.

Definition 4.5 (Observable Traces)
An *observable trace* π is an omniscient trace stripped of its information regarding source and target modes: $\pi \in \mathbb{R}^n \times \mathbb{R}^+ \times \mathbb{R}^n \times (\Lambda \times \mathbb{R}^+ \times \mathbb{R}^n)^k$.

Def. Observable Trace

For the remainder of this chapter, unless stated otherwise, a trace refers to an observable trace and the languages of automata refer only to finite traces.

Notation

Any automaton with decoration such as $\mathcal{H}^+$ will be implicitly destructed into its components with the same decoration, e.g. M^+ denotes the modes of $\mathcal{H}^+$. $(x)_k$ denotes the k^{th} component of the n-dimensional vector x for $0 < k \leqslant n$. The length $|\pi|$ of a trace $\pi \in \mathbb{R}^n \times \mathbb{R}^+ \times \mathbb{R}^n \times (\Lambda \times \mathbb{R}^+ \times \mathbb{R}^n)^k$ is the number of timed transitions occurring in it, i.e., $k+1$. A trace of length $k+1$ is implicitly destructed into the following components: $\pi = x_0^\pi, \delta_0^\pi, x_1^\pi, e_1^\pi, \delta_1^\pi, \ldots, x_k^\pi, e_k^\pi, \delta_k^\pi, x_{k+1}^\pi$. Moreover, mode μ is a member of an omniscient trace if it reaches the mode at least once: $\mu \in \pi \iff \exists i \colon e_i^{\tilde{\pi}} = (\mu_1, \lambda, \mu_2)$ with $\mu \in \{\mu_1, \mu_2\}$. A *step* of a trace is the combination of a delay and a discrete transition. Further, let Π be a sequence of traces in arbitrary order. Then, π_i denotes the i^{th} entry of the sequence with $i \leqslant |\Pi|$.

Bisimulation

Discrete bisimulation on two automata is defined conventionally by disregarding any continuous behavior and behavior not shared among the automata.

Definition 4.6 (Discrete Bisimulation)

Def. Discrete Bisimulation

Two modes of μ_1, μ_2 of two automata $\mathcal{H}_1, \mathcal{H}_2$ are *discretely bisimilar* $\mu_1 \approx \mu_2$ iff for all transition labels $\lambda \in \Lambda_1 \cap \Lambda_2$:

$$(\mu_1, \lambda, \mu_1') \in E_1 \implies \exists \mu_2' \colon (\mu_2, \lambda, \mu_2') \in E_2 \wedge \mu_1' \approx \mu_2' \text{ and}$$

$$(\mu_2, \lambda, \mu_2') \in E_2 \implies \exists \mu_1' \colon (\mu_1, \lambda, \mu_1') \in E_1 \wedge \mu_2' \approx \mu_1'$$

Further, the two automata are discretely bisimilar, i.e., $\mathcal{H}_1 \approx \mathcal{H}_2$ iff $\mu_I^1 \approx \mu_I^2$.

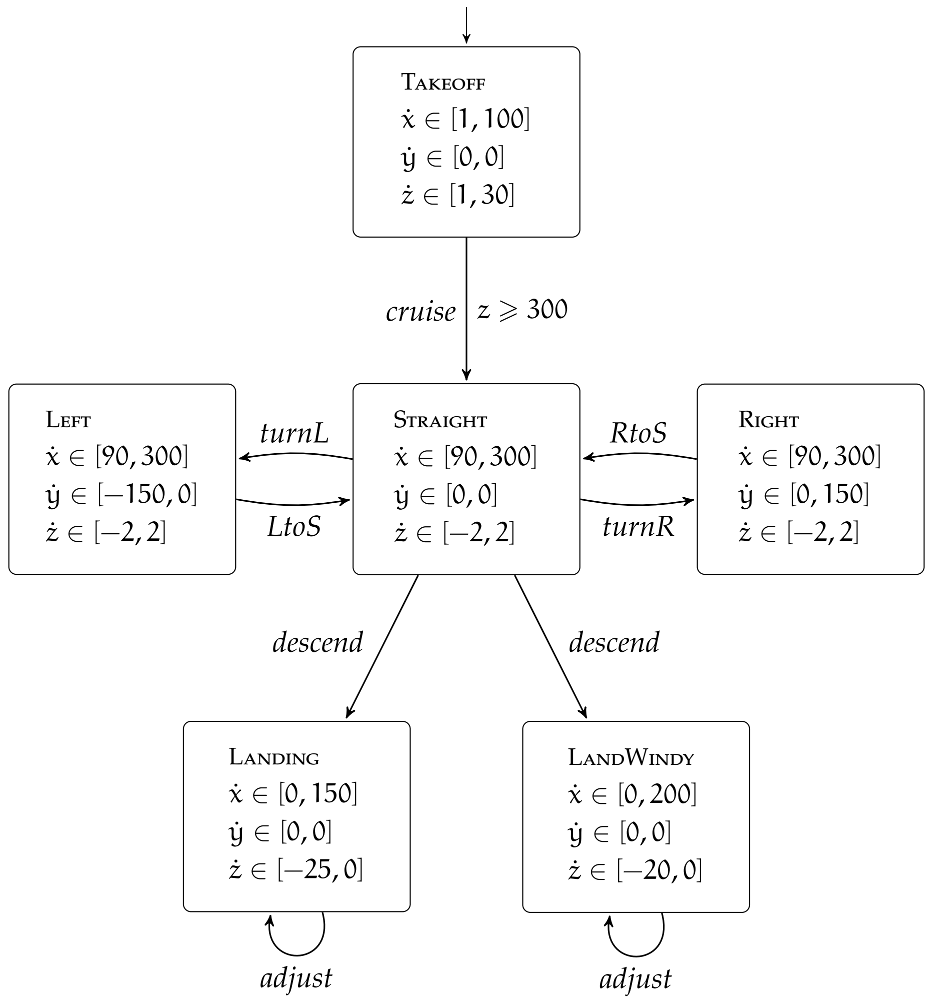


Figure 4.1.: Visual representation of a hybrid automaton. Rectangles denote modes, text inside is the name followed by differential inclusions representing the dynamics for each dimension. Arrows represent edges, their labels are action labels and guard conditions. Tautological guard conditions are omitted.

4.2. Motivation

The foundation for the conservative construction is a set of traces generated from an unknown system and a runtime monitoring specification thereof. In a nutshell, the construction first generates a discrete automaton based on the specification. It then enriches the automaton with continuous information extracted from the traces to obtain a hybrid automaton. Refining the automaton further yields a conservative approximation of the system behavior.

This section provides an intuitive overview over each of these steps for the aircraft from Example 4.3. A specification for it imposes several constraints depending on the current state of the system. For example, during takeoff, the specification requires the aircraft to accelerate; while traveling it requires a stable altitude; during landing it requires the landing gear to be lowered. An analysis of the specification hence yields a state machine — the *specification automaton* — with coarse information on different execution phases as well as conditions on phase-changes. The state machine is depicted in color in Figure 4.2, superimposed by the aircraft automaton. As can be seen, the specification does not distinguish between maintaining course or adjusting it; the requirements on the system remain the same. Yet, it contains no indication regarding the continuous behavior of the system.

Specification Automaton

Successive Enrichment

To fill these gaps, the conservative construction then successively *enriches* the specification automaton with information extracted from the set of traces. The whole process is illustrated in Figure 4.3. It first translates the specification automaton $\mathcal{A}^{\Phi}$ into a hybrid automaton $\mathcal{H}_1^+$. For each step of each trace, it adds more modes into the automaton while maintaining the structure provided by the specification. The result, i.e., $\mathcal{H}_{|\Pi|}^+$, is by design overly restrictive: it consists of a single, isolated path for each trace. A *merge process* based on the discrete behavior of modes remedies this problem and finally produces $\mathcal{H}^+$.

Merge Process

Conservative Hybrid Automaton

The key point behind $\mathcal{H}^+$ is that it is *conservative*, i.e., under certain assumptions on the specification and traces, $\mathcal{H}^+$ over-approximates $\mathcal{H}$. The assumptions are three-fold: the specification needs to be a coarse abstraction of the actual system, it must agree with the system on phase changes, and the set of traces needs to encompass sufficient information on the discrete behavior. While these assumptions seem strong, they are tailored for the use case at hand such that they are expected to be satisfied naturally. The first and second assumption concern the specification, which is hand-crafted specifically for the system. Hence, the specifier must have had knowledge regarding the abstract control structure, e.g. through which phases the system traverses during a mission. Any entity in this structure constitutes such an execution phase. These phases are abstract views on the system, i.e., they summarize several concrete control modes without containing any details on them. Each phase imposes a different, potentially overlapping set of requirements on the system. A violation of such a requirement constitutes a safety-hazard. Hence, the specification needs to accurately detect when a phase change took place. This detection is part of the specification and has to agree with the system.

Regarding the third criterion, recall that the set of traces is a development artifact that arose from the testing process. Thus, not only do they cover large parts of the system's regular execution, they represent executions in which the system was purposefully coerced into extreme and corner case behavior. Since an empirical evaluation revealed that even relatively few random walks[1] already satisfy the criterion, it is safe to assume that traces of a carefully tested, safety-critical system, do so as well. A formal description of the assumptions and the proof that $\mathcal{H}^+$ is conservative can be found in Section 4.4. →Sec. 4.4, p. 171

The empirical evaluation of the approach in Section 4.5 allows for validating three claims: →Sec. 4.5, p. 178

Few Input Traces The construction requires a low volume of input traces — especially when compared to machine-learning approaches. For the running example, the construction requires as little as three traces of length eight.

Scalability The construction scales linearly for increasing dimension and quadratically in the number of traces and size of the original/constructed system. Constructing a three-dimensional automaton with 2^{10} modes based on a specification with nine states and 512 traces requires less than a second.

Precision The level of over-approximation is within reason. For the aircraft, the construction fails to distinguish the two terminal modes. Apart from this, the resulting automaton is identical to the original one.

[1]On average 32 random traces for an automaton with 14 states and two random traces for a seven state automaton, details follow in Section 4.5, p. 178.

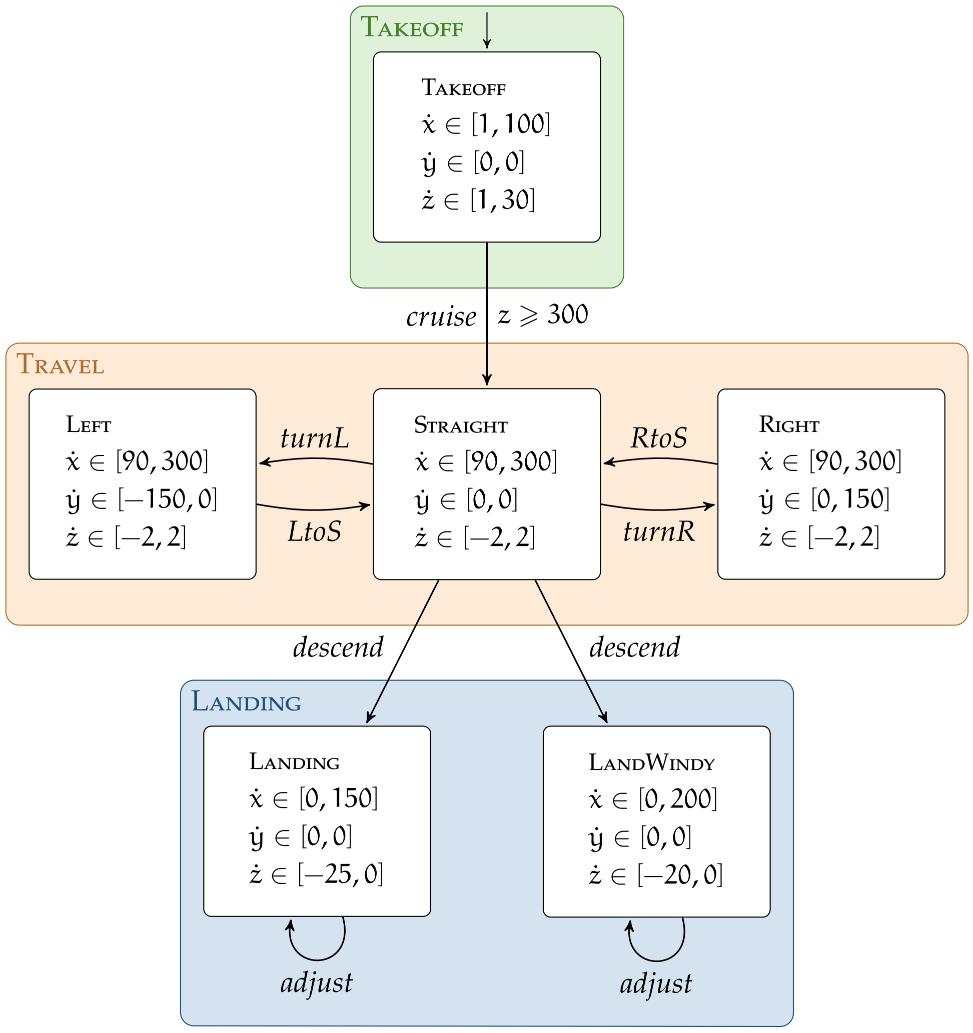


Figure 4.2.: Specification automaton of the running example superimposed by its theoretical "perfect" model.

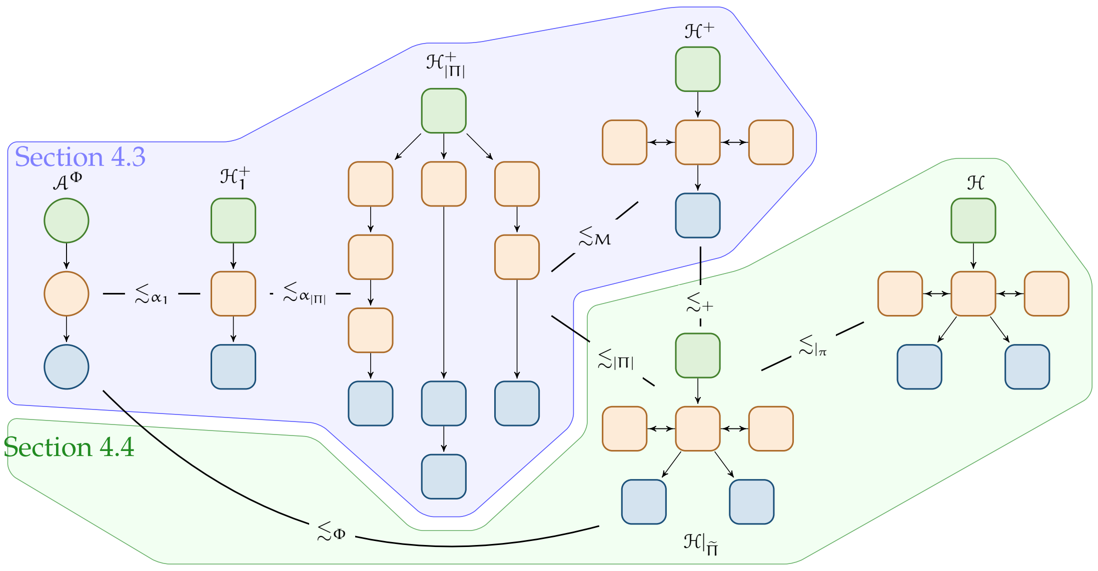


Figure 4.3.: An overview of different automata and their simulation relations, where $\mathcal{H}_1 \lesssim \mathcal{H}_2$ denotes that $\mathcal{H}_2$ simulates $\mathcal{H}_1$.

4.3. Constructing Conservative Automata

The construction proceeds in three steps: First, it extracts information from the specification to obtain a finite state machine $\mathcal{A}^{\Phi}$ and a table mapping discrete control mode changes to conditions for undergoing such a change. The automaton is a coarse abstraction of the underlying system. The second step transforms it into a hybrid automaton $\mathcal{H}^{+}$ and iteratively refines it by extracting information regarding the continuous behavior from the input traces. By design, the refinement overshoots its goals, resulting in an abstraction that is too fine. As a remedy, the third step merges parts of the automaton to construct a conservative automaton.

4.3.1. Extracting Discrete Information from the Specification

The requirements on the system change depending on its current state. For example, during the landing of an airplane, the landing gear must be lowered whereas it is required to be retracted when on traveling altitude. Hence, the specification needs to keep track of relevant parts of the system state to impose the proper restrictions. This process of keeping track induces an abstract state machine that lacks any information on the continuous dynamics since the monitor solely relies on external input data such as sensor readings. Each abstract state may summarize several concrete modes of the actual system. In the plane example, the requirements on the abstract mode "in full flight" apply to all control for staying on or adjusting the course, even though they have different continuous dynamics. By assumption, the contrary is false, i.e., a change of requirements on the system is always accompanied by a change in concrete modes. Intuitively, a change of requirements is strongly linked to an action or reaction of the system. For example, the set of requirements changes when a plane starts to descent, which is an actively initiated process with direct impact on the behavior of the system and thus a change in the control mode. A formalization of these assumptions follows in Section 4.4.1.

➔ Sec. 4.4.1, p. 171

Remark 4.1 (Automatic versus Manual Extraction)**.** The following shows an automatic extraction of the specification automaton from an RTLola specification. Yet, this requires strict compliance to certain syntactic patterns. In particular, it restricts some expressions to linear arithmetic since they need to be valid in the definition of a hybrid automaton. As a result, a convenient and clear expression in the specification might not translate to a rectangle without some manual tweaking. Hence, in some cases, it is preferable to extract the specification automaton manually based on the specification rather than relying on the syntactic extraction presented next.

A specification can keep track of the current set of requirements imposed on the system by using an output stream with the name μ^{Φ}. The value of μ^{Φ} indicates in which abstract state the system is. Assume there are two abstract states μ_1^{Φ} and μ_2^{Φ}, and a state transition occurs under some condition φ. Then, the μ^{Φ} stream has the following shape:

```
output μ^Φ := if μ^Φ = μ₁^Φ ∧ φ then μ₂^Φ else μ^Φ.last(or: μ₁)
```

For more possible abstract states and transitions, the conditional statement can be extended accordingly. In addition, suppose a state change is accompanied by a respective trigger, where the trigger is purely informative rather than indicative of an error.

```
trigger μ^Φ.last(or: μ₁) = μ₁^Φ ∧ μ^Φ = μ₂^Φ ∧ φ "μ₁^Φ → μ₁^Φ: λ."
```

The trigger checks for a change in μ^Φ from μ_1^Φ to μ_2^Φ and emits this information coupled with the name λ of the respective transition.

An analysis of the output stream and trigger declarations yields the following:

Definition 4.7 (Specification Automaton And Guard Condition Table)

Given a specification Φ, the extraction yields two artifacts. The *(abstract) specification automaton* $\mathcal{A}^\Phi = (V^\Phi, E^\Phi, v_I^\Phi)$ is a finite state machine where each state represents an operational phase. The *guard condition table* $\Gamma^\Phi = (V^\Phi \times \lambda \times V^\Phi) \to \mathcal{I}^n$ maps a phase transition to a condition in form of a multidimensional rectangle.

Def. (Abstract) Specification Automaton
Def. Guard Condition Table

In the construction of $\mathcal{A}^\Phi$, V^Φ and v_I^Φ are the domain and initial value of μ^Φ, respectively. Then, for each trigger as the one stated before, E^Φ contains the edge $(\mu_1^\Phi, \lambda, \mu_2^\Phi)$ and $\Gamma^\Phi(e) = \varphi$. Note that the following assumes $\mathcal{A}^\Phi$ to be free of unreachable states and related edges. This is the case in sensible specifications and can easily be enforced by pruning the respective parts of the automaton.

Example 4.8 (Specification Automaton Extraction)**.** Consider the following specification excerpt for the running example. For better illustration, the domain of the `mode` stream is {`Takeoff`, `Travel`, `Landing`}.

```
input altitude, lon, lat, ...: Float32
input landing_gear: Bool
input descend: NoValue

output phase @ alt ∨ descend :=
  if phase.last(or: Takeoff) = Takeoff then
    if altitude.hold(or: 0) >= 300 then Travel else Takeoff
  else if phase.last(or: Takeoff) = Travel then
    if descend.hold(or: false) then Landing else Travel

...

trigger phase.last(or: Takeoff) = Takeoff ∧ phase = Travel
    "Takeoff → Travel: cruise"
trigger phase.last(or: Takeoff) = Travel ∧ phase = Landing
    "Travel → Landing: descend"
```

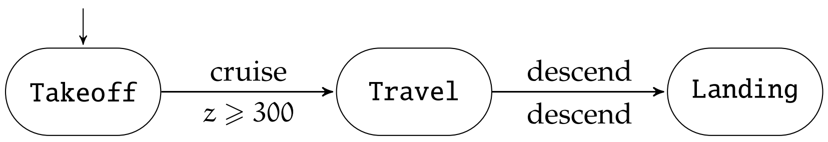


Figure 4.4.: The automaton extracted from a specification. Labels above arrows are the names of action they represent, labels below are guard conditions. Identical labels indicate control decisions.

```
trigger landing_gear ∧ phase.hold(or: Takeoff) = Travel
   "Landing gear extended at while traveling."
trigger altitude > 10,000 "Flying too high."
...
```

Here, phase is the output stream representing the current operational phase, i.e., μ^{Φ} in the explanation before. Note that the first two triggers mark transitions whereas the remaining constitute safety properties.

The specification yields the three-state automaton shown in Figure 4.4. Note the dual meaning of "descend". It is both an input of the monitor and thus a constraint, and the label of the transition. This indicates that the control logic directly informs the monitor of a phase change. As a result, the condition is tautological as it can happen any time, initiated by the controller.

Hence, the guard condition table is

$$\Gamma^{\Phi}(e) = \begin{cases} (-\infty,\infty)^2 \times [300,\infty) & \text{if } e = (\texttt{Takeoff}, \texttt{cruise}, \texttt{Travel}) \\ \mathit{true} = (-\infty,\infty)^3 & \text{if } e = (\texttt{Travel}, \texttt{descend}, \texttt{Landing}) \end{cases}$$

The connection to the underlying system becomes apparent when superimposing it, as can be seen in Figure 4.2. △

→ Fig. 4.2, p. 160

4.3.2. Extracting Continuous Information from Traces

While the specification provides information about the system's *discrete* structure, the traces reveal how the *continuous* state of the system evolves over time. They also reveal mode changes within a single abstract state. This information allows for transforming $\mathcal{A}^{\Phi}$ into a more fine-grained automaton with annotated dynamics in each mode. For this, the transformation iteratively constructs an automaton $\mathcal{H}^{+}$, processing each position of all traces in separation. This requires to keep track of two maps: a *concrete mode-map* $\psi\colon \Pi \to M$ that maps each trace to the mode of the constructed automaton in which it currently resides, and an *abstract mode-map* $\alpha\colon \mu \to V^{\Phi}$ mapping each concrete mode to an abstract state in the specification automaton $\mathcal{A}^{\Phi}$.

Concrete Mode-Map

Abstract Mode-Map

Definition 4.9 (Construction Initialization)

The construction starts with a quasi-empty hybrid automaton $\mathcal{H}_1^+$ that is structurally similar to $\mathcal{A}^\Phi$, a concrete mode-map ψ_1 and an abstract mode-map α_1 defined as:

$$M_1 = \{\mu_I\} \qquad \Lambda_1 = \emptyset \qquad E_1 = \emptyset \qquad \gamma_1(e) = \mathbb{1} \qquad \psi_1(\pi) = \mu_I$$

$$\alpha_1(\mu_I) = \nu_I^\Phi \qquad \mathit{flow}_1(\mu) = \prod_{\pi_i \in \Pi} \text{solve}(x_0^{\pi_i}, x_1^{\pi_i}, \delta_0^{\pi})$$

Here, $\mathbb{1}$ denotes the neutral element with respect to the multiplication of intervals. Moreover, the solve-function computes the singular interval representing the linear dynamics exhibited by a delay transition:

$$\text{solve}(x, x', \delta) = [(x' - x)\delta^{-1}, (x' - x)\delta^{-1}]$$

Thus, $\mathcal{H}_1^+$ already incorporates the information of each trace regarding their first delay transition.

After the initialization, the procedure successively incorporates information contained in further positions of the traces.

Definition 4.10 (Construction Step)

Given the automaton $\mathcal{H}_k^+$, concrete mode-map ψ_k, and abstract mode-map α_k from the previous construction step. Consider the kth step of each input trace, i.e., $x_k^{\pi_i}$, $\lambda_k^{\pi_i}$, $\delta_k^{\pi_i}$, and $x_{k+1}^{\pi_i}$ for all $0 < i \leqslant |\Pi|$. The kth step of the construction produces $\mathcal{H}_{k+1}^+, \psi_{k+1}$, and α_{k+1}.

Let $\alpha_k(e)$ be the abstraction of an edge $e = (\mu_1, \lambda, \mu_2)$, i.e., it determines the respective edge in the specification automaton. Formally, $\alpha_k(e) = (\alpha_k(\mu_1), \lambda, \alpha_k(\mu_2))$. Moreover, $\Phi(\pi[..k], e)$ indicates that an edge e of the specification automaton was derived from a trigger for which the RTLola monitor reports a violation for the trace π up to the kth step. Lastly, $\mu_{i,k}$ are fresh modes.

$$M_{k+1} = M_k \cup \bigcup_i \{\mu_{i,k}\} \qquad \Lambda_{k+1} = \Lambda_k \cup \bigcup_i \{\lambda_k^{\pi_i}\}$$

$$E_{k+1} = E_k \cup \bigcup_i \{(\psi_k(\pi_i), \lambda_k^{\pi_i}, \mu_{i,k})\}$$

$$\psi_{k+1}(\pi_i) = \mu_{i,k}$$

$$\gamma_{k+1}(e) = \begin{cases} \gamma_k(e) & \text{if } e \in E_k \\ \Gamma^\Phi(\alpha_{k+1}(e)) & \text{otherwise} \end{cases}$$

$$\mathit{flow}_{k+1}(\mu) = \begin{cases} \text{solve}(x_k^{\pi_i}, x_{k+1}^{\pi_i}, \delta_k^{\pi_i}) & \text{if } \mu = \mu_{i,k} \\ \mathit{flow}_k(\mu) & \text{otherwise} \end{cases}$$

$$\alpha_{k+1}(\mu) = \begin{cases} \mu^{\alpha} & \text{if } \exists \pi\colon \Phi(\pi[..k], (\alpha_k(\psi_k(\pi)), \lambda_k^{\pi}, \mu^{\alpha})) \\ \alpha_k(\mu) & \text{if } \mu \in M_k \\ \alpha_k(\psi_k(\pi_i)) & \text{if } \mu = \mu_{i,k} \end{cases}$$

Intuitively, for each position of each trace the construction

1. adds a new mode with the dynamics exhibited by the delay transition,
2. adds a new edge from μ to μ' for the discrete transition, and
3. updates the mode maps accordingly.

The latter means that if the transition was accompanied by a step in $\mathcal{A}^{\Phi}$, α maps the μ' to the respective abstract mode and looks up the guard obtained from the specification. Otherwise, it maps μ' to the same abstract state as μ with a vacuous guard indicating a lack of information.

4.3.3. Merging Modes

Evidently, following the procedure yields an automaton $\mathcal{H}^{+}_{|\Pi|}$ with $|\pi| \cdot |\Pi|$ modes arranged as a tree as can be seen in Figure 4.3. It transformed the overly coarse specification automaton into an overly fine hybrid automaton. To find the sweet spot between both extremes, the next construction step merges modes within an abstract state provided they are sufficiently similar. Suppose some relation $\sim_{\exists\lambda}$ captures this notion of similarity. Then, intuitively, the construction deems any two modes $\mu \not\sim_{\exists\lambda} \mu'$ sufficiently *dis*similar such that they must represent different modes in the original system. For this, let $\sim_{\alpha}$ denote the *refinement relation* induced by an abstract mode-map α for a constructed hybrid automaton. Here, $\mu_1 \sim_{\alpha} \mu_2$ indicates that both modes refine the same abstract state, i.e., $\alpha(\mu_1) = \alpha(\mu_2)$.

Def. Refinement Relation

Definition 4.11 (Action Similarity)

For a constructed hybrid automaton $\mathcal{H}^{+}_{i}$ for some i, two modes $\mu_1, \mu_2 \in M_i$ are *action-similar* if they share some discrete characteristics and reside in the same abstract state of the specification. Assume there are some modes $\mu'_1, \mu'_2 \in M_i$ and action $\lambda \in \Lambda_i$.

Def. Action Similarity

$$\begin{aligned} \mu_1 \sim_{\exists\lambda} \mu_2 \iff & \mu_1 \sim_{\alpha_i} \mu_2 \\ & \wedge \left(\{(\mu'_1, \lambda, \mu_1), (\mu'_2, \lambda, \mu_2)\} \subseteq E_i \vee \{(\mu_1, \lambda, \mu'_1), (\mu_2, \lambda, \mu'_2)\} \subseteq E_i\right) \end{aligned}$$

Note that by construction $\sim_{\alpha}$ is coarser than $\sim_{\exists\lambda}$.

Terminal modes need further attention: consider the automaton in Figure 4.2. There are two identical traces in the language of the automaton starting in Takeoff and traversing Straight, but ending in different Landing modes. Based on these traces the construction cannot distinguish the two terminal modes, since the difference in modes is unobservable. In fact, there is no finite set of traces for which they can be distinguished with certainty. This forces the construction to merge them as can be seen in Figure 4.5.

→ Fig. 4.2, p. 160

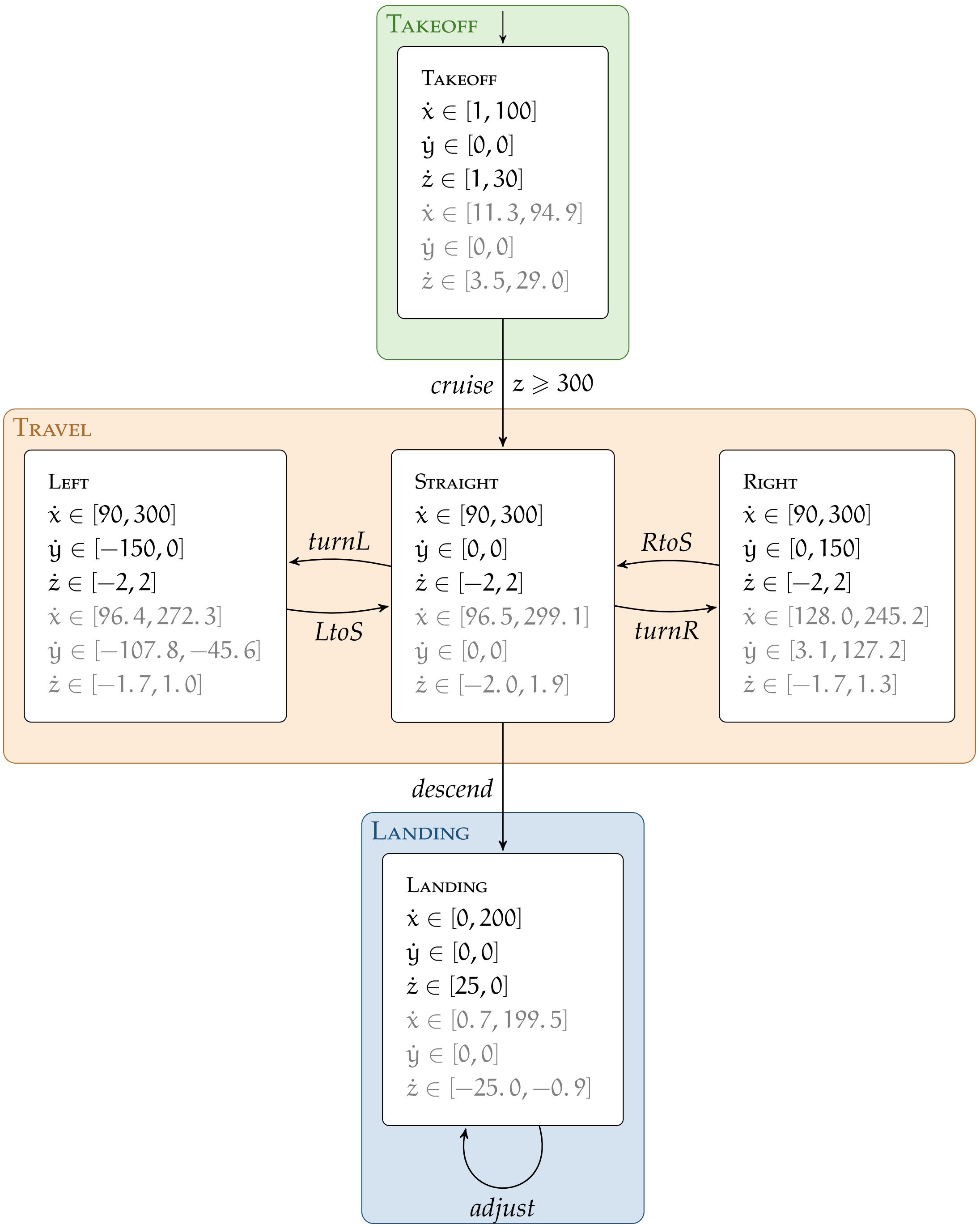


Figure 4.5.: Output of the conservative construction for the running example. Dynamics in black are obtained via three hand-picked traces whereas gray dynamics result from ten randomly generated traces.

Definition 4.12 (Terminal and Merge Similarity)

Two terminal modes are *terminal-similar* iff they reside in the same abstract state.

Def. Terminal-Similarity

$$\mu_1 \sim_\perp \mu_2 \iff \text{outdeg}(\mu_1) = \text{outdeg}(\mu_2) = 0 \wedge \alpha(\mu_1) = \alpha(\mu_2)$$

Two modes are *merge-similar* iff they are either action-similar or terminal-similar: $\sim_M = \sim_{\exists\lambda} \cup \sim_\perp$

Def. Merge Similarity

Merge Operation. The merge operation now minimizes the automaton with respect to $\sim_M$ by building the quotient automaton. Formally, a merge operates on an equivalence relation $\approx$ over the set of modes. Each equivalence class $\zeta \subseteq M$ will be replaced by a single, arbitrary representative $[\![\zeta]\!]_\approx$. By slight abuse of notation let $[\![\mu]\!]_\approx$ denote the representative of the equivalence class of μ, i.e., $[\![\mu]\!]_\approx = [\![\zeta]\!]_\approx$ for $\mu \in \zeta$. Moreover, if context permits, the subscript may be omitted. The representative conserves the language of each mode contained in ζ by retaining discrete transitions and computing the convex hull for its continuous components.

Definition 4.13 (Quotient Automaton)

Merging an automaton $\mathcal{H}$ with respect to an equivalence relation $\approx$ yields a *quotient automaton* $\mathcal{H}{\downarrow_\approx}$ where all elements of an equivalence class get merged into a single element exhibiting the convex semantics of its members.

Def. Quotient Automaton

$$\begin{aligned}
M{\downarrow_\approx} &= \{[\![\mu]\!] \mid \mu \in M\} \\
s_I{\downarrow_\approx} &= ([\![\mu_I]\!], x_I) \\
\mathit{flow}{\downarrow_\approx}([\![\zeta]\!]) &= \mathit{Conv}(\bigcup_{\mu \in \zeta} \{\mathit{flow}(\mu)\}) \\
\psi{\downarrow_\approx}(\pi) &= [\![\psi(\pi)]\!] \\
E{\downarrow_\approx} &= \{([\![\mu_1]\!], \lambda, [\![\mu_2]\!]) \mid (\mu_1, \lambda, \mu_2) \in E\} \\
\alpha{\downarrow_\approx}([\![\mu]\!]) &= \alpha(\mu) \\
\Lambda{\downarrow_\approx} &= \{\lambda \mid \exists e \in E{\downarrow_\approx} \colon e = ([\![\zeta_1]\!], \lambda, [\![\zeta_2]\!])\} \\
\gamma{\downarrow_\approx}((\zeta_1, \lambda, \zeta_2)) &= \mathit{Conv}(\{(\mu_1, \lambda, \mu_2) \mid \mu_1 \in \zeta_1 \wedge \mu_2 \in \zeta_2\})
\end{aligned}$$

4.3.4. Construction Algorithm

The overall construction algorithm now proceeds as outlined in Algorithm 1: First, the procedure extracts information from the specification, constructs the initial automaton and refines it successively by iterating over the traces. After processing all traces completely, the procedure computes and applies the merges with respect to action- and terminal-similarity.

Algorithm 1: Construct Conservative Hybrid Automaton

```
Require: Specification $\Phi$, Traces $\Pi$
1: Extract $\mathcal{A}^{\Phi}, \Gamma^{\Phi}$ from $\Phi$                          ▷ Definition 4.7
2: Construct $\mathcal{H}_1^+, \psi_1, \alpha_1$ from $\Pi$ and $\Gamma^{\Phi}$           ▷ Definition 4.9
3: for $k$ from 1 to $|\pi|$ for $\pi \in \Pi$ do
4:     Update to $\mathcal{H}_k^+, \psi_k, \alpha_k$                                ▷ Definition 4.10
5: end for
6: Compute the action-similarity $\sim_{\exists\lambda}$                          ▷ Definition 4.11
7: Compute the merge-similarity $\sim_M$                                ▷ Definition 4.12
8: Compute the conservative automaton $\mathcal{H}^+ = \mathcal{H}^+_{|\Pi|}\downarrow_{(\sim_M)}$   ▷ Definition 4.13
```

Time Complexity. The construction process consists of three phases: extraction, construction and merging. Recall that the dimensionality, i.e., the number of continuous state variables, is n. The first phase scales linearly in the size of the specification $\mathcal{O}\,|\Phi|$. The second phase constructs an automaton with a single mode per step of any trace. Its size and the running time of the construction this scales linearly with the number of traces multiplied by their length. It is also linear in the dimension since the dynamics of each dimension have to be computed separately per mode. Hence, the complexity is in $\mathcal{O}\,(n \cdot |\pi| \cdot |\Pi|)$. Lastly, the complexity of the last phase depends on the complexity of a single merge, which is linear in the dimension, and the number of merges. The latter is quadratic in the size of $\mathcal{H}^+_{|\Pi|}$, which in turn is linear in the number and length of traces: $\mathcal{O}\left(n \cdot |\mathcal{H}^+_{|\Pi|}|^2\right) = \mathcal{O}\left(n \cdot |\pi|^2 \cdot |\Pi|^2\right)$. However, this only describes the worst case. For the best case, recall that the procedure compares each mode against each other with respect to $\sim_M$ and merges them. Optimally, all elements of an equivalence class are identified successively by chance, i.e., it first identifies and merges all members of the first equivalence class, then continues with the second and so on. In this case, the process is quadratic in the number of equivalence classes: $\Omega\left(n \cdot |{\sim_M}|^2\right)$. Here, $|{\sim_M}|$ denotes the number of equivalence classes induced by $\sim_M$ with

$$\left|\mathcal{A}^{\Phi}\right| \leqslant |{\sim_M}| = \left|M^+\right| \leqslant \left|M^+_{|\Pi|}\right|$$

In conclusion, the overall asymptotic running time is dominated by the merge procedure:

$$\mathcal{O}\left(n \cdot |\pi|^2 \cdot |\Pi|^2\right)$$

4.4. Correctness of the Construction

The validation of the construction requires a proof that the constructed automaton is — under certain assumptions on the input — indeed conservative. For this, a key criterion is that the automaton over-approximates the discrete and continuous behavior of the original system when projected down to the parts that contributed to the inputs. Naturally, if the original system encompasses parts that were neither reflected in the specification, nor traversed in the input traces, the constructed automaton cannot reconstruct it.

Hence, this section first formalizes requirements on the input data. Then, a definition of projection automata enables proving that the constructed automaton subsumes the language of the projected original system.

4.4.1. Requirements on Input Data

The construction of the conservative automaton relies on the quality of the input traces and specification. Hence, they need to satisfy three criteria:

1. the specification must be an abstraction of the real system,
2. its trigger conditions must be at least as restrictive as the respective conditions on mode changes, and
3. the trace set needs to traverse every control mode of the system sufficiently often to capture the discrete behavior.

Definition 4.14 (Adequacy of Input Data)

A specification Φ and trace set Π are *adequate* for a hybrid automaton $\mathcal{H}$ iff they satisfy three criteria.

Def. Input Adequacy

1. The specification induces a *coarser automaton* $\mathcal{A}^{\Phi}$ than the original, i.e., there is a partition rendering the quotient automaton of the original system discretely bisimilar to the specification automaton.

   Def. Coarse Abstraction

   $$\exists \sim_{\Phi} : \mathcal{H}{\downarrow_{\sim_{\Phi}}} \approx \mathcal{A}^{\Phi}$$

2. For any discrete transition that is both in $\mathcal{H}$ and $\mathcal{A}^{\Phi}$, the specification contains a mode change condition that is *at least as permissive* as the guard of the respective transition in $\mathcal{H}$. Formally, let $\mu_1^{\Phi}, \mu_2^{\Phi} \in V^{\Phi}$ be two different states in the specification automaton and μ_1, μ_2 be two different states in the original automaton. Suppose these pairs of modes are discretely bisimilar, i.e., $\mu_1^{\Phi} \approx \mu_1$ and $\mu_2^{\Phi} \approx \mu_2$. Then, for any edges $(\mu_1^{\Phi}, \lambda, \mu_2^{\Phi}) \in E^{\Phi}$ in the specification automaton and edge $e = (\llbracket \mu_1 \rrbracket_{\sim_{\Phi}}, \lambda, \llbracket \mu_2 \rrbracket_{\sim_{\Phi}}) \in E{\downarrow_{\sim_{\Phi}}}$ in the quotient of the original system sharing the same label:

   Def. Trigger-Guard Compatibility

   $$\gamma(\mu, \lambda, \mu) \implies \Gamma^{\Phi}(e)$$

Def. Trace Set Coverage

3. For every mode μ in $\mathcal{H}$, let $a_\mu = \text{indeg}(\mu) + \text{outdeg}(\mu)$ be the number of input and output actions of μ in Π. The trace set needs to contain *more than* $a_\mu(a_\mu - 1)/2$ *traversals through* μ. Here, a trace π traverses through a mode if its omniscient counterpart $\widetilde{\pi}$ contains two subsequent edges first ending and then starting from μ. Formally, let $\widetilde{\pi}$ be the omniscient trace of π, $i < |\pi|$ be a non-terminal index in it. Further, let λ, λ' be two labels, and let μ_1, μ_2 be two modes, then:

$$\mu \in_i \pi \iff e_i^{\widetilde{\pi}} = (\mu_1, \lambda, \mu) \wedge e_{i+1}^{\widetilde{\pi}} = (\mu, \lambda', \mu_2)$$

With this, the formal criterion is:

$$\forall \mu \in M_{\mathcal{H}} : |\{(i, \pi) \mid \mu \in_i \pi\}| > \frac{(a_\mu)(a_\mu - 1)}{2}$$

Evidently, these criteria depend on the original hybrid automaton, which seemingly contradicts the premise of the construction since this automaton is supposed to be unavailable. Yet, the criteria are designed in a way that they are either satisfied naturally or can be satisfied without access to all formal details of the system.

To understand this, consider the first and second criterion. These criteria restrict the specification, which was hand-crafted for the underlying system. Here, a reasonable specification summarizes control modes that are subject to the same requirements; at the same time, the specification needs to capture changes in the abstract state precisely to impose the correct sub-specification on the system. Thus, even without perfect knowledge of the inner workings and dynamics of the system, the first two criteria can be ensured. Consider the third criterion, which is concerned with the trace set. A thorough testing process demands that all discrete paths through the system are tested at least once, traversing cycles only a bounded amount of times. Moreover, the system has a fixed control interface, represented by Λ. As a result, it is reasonable to assume that the number of times each control mode is traversed during the development exceeds the threshold required by the third criterion. This again does not rely on knowledge about the exact mode structure nor dynamics of the underlying system.

The exact threshold for the third criterion seems arbitrary but is anchored in graph theory, the impact of which can be seen in the next lemma.

Lemma 4.15 (Trace Connectivity). *Let Φ and Π be adequate for $\mathcal{H}$. For any mode μ in $\mathcal{H}$ with incoming edge label λ_i and outgoing edge label λ_o, there is a mode μ' in $\mathcal{H}^+$ with the very same edge labels and $\alpha(\mu) = \alpha(\mu')$.*

Proof By reduction on the graph connectivity problem. Let $\mathcal{G}(\mu, \Pi) = (V, E)$ be a graph where V is the set of labels of incoming or outgoing edges of μ in $\mathcal{H}$. For two labels $\lambda_1, \lambda_2 \in V$, there is an edge $(\lambda_1, \lambda_2) \in E$ iff there is a trace π and $i < |\pi|$ with $\mu \in_i \pi$ where λ_1 and λ_2 correspond to the existentially quantified actions λ, λ'. It follows from Menger's theorem [Men27] that λ and λ' are necessarily connected if $|E|$ exceeds $|V| \cdot (|V| - 1)/2$.

This threshold corresponds to third criterion of adequacy. Recall the action similarity $\sim_{\exists\lambda}$ defined in Definition 4.11, relates all modes with at least one common incoming or outgoing edge label. Thus, since the merge similarity $\sim_M$ refines $\sim_{\exists\lambda}$, all respective modes are merged in $\mathcal{H}^+$. By Definition 4.13, the resulting mode $[\![\mu]\!]_{\sim_M}$ retains these transitions. Lastly, since $\sim_\Phi \subseteq \sim_{\exists\lambda} \subseteq \sim_M$, merge similarity also refines $\sim_\Phi$, hence $\alpha(\mu) = \alpha([\![\mu]\!]_{\sim_M})$. This concludes the proof. □

→ Def. 4.11, p. 166

→ Def. 4.13, p. 169

4.4.2. Projection Automata

The assessment of the quality of the reconstruction depends on the projection of the original system onto the set of traces. This first requires a definition of projections on automata.

Definition 4.16 (Projection Automata)

The *projection* of an automaton $\mathcal{H}$ down to a set of omniscient traces $\widetilde{\Pi}$ is an automaton $\mathcal{H}|_{\widetilde{\Pi}}$ with the following constituents.

Def. Projection Automaton

$$
\begin{aligned}
M|_{\widetilde{\Pi}} &= \bigcup_{\widetilde{\pi}\in\widetilde{\Pi}} \bigcup_{i\leqslant|\widetilde{\pi}|} \left\{\mu_i^{\widetilde{\pi}} \mid \mu_i^{\widetilde{\pi}} \in M\right\} \\
\Lambda|_{\widetilde{\Pi}} &= \bigcup_{\widetilde{\pi}\in\widetilde{\Pi}} \bigcup_{i<|\widetilde{\pi}|} \left\{\lambda_i^{\widetilde{\pi}} \mid \lambda_i^{\widetilde{\pi}} \in \Lambda\right\} \\
E|_{\widetilde{\Pi}} &= \left\{(\mu,\lambda,\mu') \in E \mid \mu,\mu' \in M|_{\widetilde{\Pi}} \wedge \lambda \in \Lambda|_{\widetilde{\Pi}}\right\} \\
s_I|_{\widetilde{\Pi}} &= s_I \\
\gamma(e)|_{\widetilde{\Pi}} &= [\nu_e^{\min}, \nu_e^{\max}] \\
\mathit{flow}(\mu)|_{\widetilde{\Pi}} &= [\nu_\mu^{\min}, \nu_\mu^{\max}]
\end{aligned}
$$

Here, for $\varphi \in \{\min, \max\}$, the min and max values for guards and flows are:

$$
\begin{aligned}
\nu_e^{\varphi} &= \varphi\left\{x \mid \exists\widetilde{\pi}, \exists i < |\widetilde{\pi}| \colon x = x_i^{\widetilde{\pi}} \wedge e = e_i^{\widetilde{\pi}}\right\} \\
\nu_\mu^{\varphi} &= \varphi\left\{f \mid \exists\widetilde{\pi}, \exists i < |\widetilde{\pi}| \colon e_i^{\widetilde{\pi}} = (\mu',\lambda,\mu) \wedge x_i^{\widetilde{\pi}} + \delta_i^{\widetilde{\pi}} f = x_{i+1}^{\widetilde{\pi}}\right\}
\end{aligned}
$$

Intuitively, the projection strips the automaton of any information not reflected in the set of traces. This removes all modes, edges, and transition labels not contained in any trace. It retains the initial mode since, by definition, the initial state occurs in all traces. Guards and flows are reduced to the maximum and minimum value exhibited by some trace.

Note that the projection automaton $\mathcal{H}|_{\widetilde{\Pi}}$ is not meant to be constructed at any point; it serves as theoretical point of reference for the quality of the construction. It is easy to see that in general the projection reduces the expressiveness of an automaton, i.e., $\mathcal{L}(\mathcal{H}) \supseteq \mathcal{L}(\mathcal{H}|_{\widetilde{\Pi}})$. This, however, is not necessarily the case as the following theorem shows.

Theorem 4.1 (*Perfect Projection*)

For any hybrid automaton $\mathcal{H}$ there is a finite set of traces for which the projection onto these traces yields the identity, i.e., $\exists \widetilde{\Pi}^* \subseteq \mathcal{L}(\mathcal{H})\colon \Pi^*$ finite $\wedge\, \mathcal{L}(\mathcal{H}|_{\widetilde{\Pi}^*}) = \mathcal{L}(\mathcal{H})$.

Proof The proof selects traces from $\mathcal{L}(\mathcal{H})$ enabling the perfect projection. For each e in E, $\widetilde{\Pi}^*$ contains a trace $\tilde{\pi}$ with $e \in \tilde{\pi}$ if such a trace exists. This immediately entails that the projected edge set, set of actions, set of modes, and initial mode are accurate barring unreachable parts. This is sufficient since they are not reflected in the language of the automaton anyway. For each mode, $\widetilde{\Pi}^*$ encompasses four traces per dimension: one minimizing and one maximizing the flow and continuous state variable of the mode and dimension. The minimization and maximization is over the set of traces rather than over the mode itself. As a result, the projection of the flow is perfect. Lastly, for each mode, outgoing edge, and dimension, there are two traces in $\widetilde{\Pi}^*$ which maximize and minimize the continuous state value before taking the transition. Hence, the projection of the guard condition is lossless in terms the language of the automaton.

In conclusion, $\mathcal{L}(\mathcal{H}|_{\widetilde{\Pi}^*}) = \mathcal{L}(\mathcal{H})$ with:

$$\left|\widetilde{\Pi}^*\right| = |E| + n\big(5|M| + \sum_{\mu \in M} \operatorname{outdeg}(\mu)\big)$$

Here, n is the dimension of $\mathcal{H}$. □

Note that the language equality cannot be extended to identical or isomorphic automata since $\mathcal{H}$ can contain unreachable modes that are not reflected in its language and thus not in any trace. The theorem emphasizes the generality of the conservative construction: For an appropriate trace set, the projection of an automaton perfectly resembles the original system. Since the constructed automaton is conservative with respect to this very projection, it is also conservative with respect to the original system. This is independent of the exact structure of the underlying system.

4.4.3. Construction Guarantees

The first observations are that application of a merge and iterations of the construction do not reduce the language of an automaton.

Lemma 4.17 (Lossless Merge). *Given a constructed hybrid automaton $\mathcal{H}^+$ and an equivalence relation $\approx$, merging $\mathcal{H}^+$ with respect to $\approx$ yields a more permissive automaton, i.e., $\mathcal{L}(\mathcal{H}^+) \subseteq \mathcal{L}(\mathcal{H}^+\downarrow_\approx)$.*

Proof By contradiction: Assume there is a trace $\pi \in \mathcal{L}(\mathcal{H}^+) \setminus \mathcal{L}(\mathcal{H}^+\downarrow_\approx)$. As the merge operation is defined by unifying modes, π either (1) takes a discrete transition or (2) traverses a continuous state not permitted in the merged automaton. Definition 4.13 "bends" edges such that they originate and end in the respective representatives. Hence, the construction retains all edges up to elimination of duplicates due to set semantics, ruling out (1). Regarding (2), the quotient builds the convex hull for all flow and guard definitions of the merged states. The convex hull is at least as permissive as its constituents, rendering a less permissive behavior impossible. □

→ Def. 4.13, p. 169

What is left to be shows is that each construction step is lossless.

Lemma 4.18 (Lossless Construction). *Given a set of traces Π and specification Φ. For any two iterations i and j, if $i \leqslant j$, then the set of edges, the flow, and the transition guards only grow over the iterations:*

$$E_i \subseteq E_j \wedge \mathit{flow}_i \subseteq \mathit{flow}_j \wedge \forall e \in E_i \colon \gamma_i(e) \subseteq \gamma_j(e)$$

Proof This lemma follows directly from the construction step of Definition 4.10. □

This suffices to prove that the language of the constructed automaton at least includes all input traces.

Theorem 4.2 (*Input Trace Inclusion*)

Given an adequate set of traces Π and specification Φ, the language of a constructed automaton $\mathcal{H}^+$ subsumes Π, i.e., $\Pi \subseteq \mathcal{L}(\mathcal{H}^+)$.

Proof Let $\pi \in \Pi$ be an arbitrary input trace. An induction shows that any subsequence of π of length i is included in the language of $\mathcal{H}_i^+$. Recall that $\psi_i(\pi)$ is the mode in which $\pi[0, i]$ ends.

Induction Base: $i = 0$. By construction $(x_0^\pi, \mu_I) \in \mathcal{L}(\mathcal{H}_0^+)$. Moreover, $\psi_0(\pi) = \mu_I$ marks the start and end point of the trace.

Induction Step: $i - 1 \to i$. Consider $\pi[0, i] = \pi[0, i-1], \lambda_i, \delta_i, x_i$. By induction hypothesis, $\pi[0, i-1] \in \mathcal{L}(\mathcal{H}_{i-1}^+)$ and by Lemma 4.18, the language membership carries over to $\mathcal{H}_i^+$. By construction, $e = (\mu_{i-1}, \lambda_i, \psi_i(\pi)) \in E_i$ with guard $\gamma_i(\pi) = \Gamma^\Phi(e^\alpha)$. Here, $\Gamma^\Phi(e^\alpha)$ is either $\top$ if the transition is not present Φ, or the condition from the respective trigger in Φ. The former case renders it trivially satisfied. In the latter case, by construction of α_i, the respective trigger is satisfied in x_{i-1}, hence the guard condition is satisfied as well. This enables the discrete transition and ensuring that the trace ends in $\psi_i(\pi)$. The delay transition is valid because of the definition of solve. Hence, $\pi \subseteq \mathcal{L}(\mathcal{H}_{|\pi|}^+)$. By Lemma 4.17, this result carries over to $\mathcal{H}^+$, i.e., $\pi \subseteq \mathcal{L}(\mathcal{H}^+)$ □

A stronger classification of the language of $\mathcal{H}^+$ requires some insight into its discrete structure in relation to the projection automaton of the original system. Specifically, the following theorem shows that $\mathcal{H}|_{\widetilde{\Pi}}$ has a finer discrete structure than $\mathcal{H}^+$.

Theorem 4.3 (*Discrete Refinement*)

Let Φ and Π be an adequate specification and trace set for a hybrid automaton $\mathcal{H}$. The reconstruction $\mathcal{H}^+$ is coarser than the projection of $\mathcal{H}$ onto Π.

$$\exists \sim_+ : \mathcal{H}^+ \downarrow_{\sim_+} \approx \mathcal{H}|_{\widetilde{\Pi}}$$

Proof The proof proceeds in two steps. First, for an arbitrary trace π through $\mathcal{H}|_{\widetilde{\Pi}}$ it generates a trace π' through $\mathcal{H}^+$. Second, it constructs the equivalence relation $\sim_+$ based on these trace pairs.

Step 1: For a given $\widetilde{\pi} \in \mathcal{L}(\mathcal{H}|_{\widetilde{\Pi}})$, the proof inductively constructs $\widetilde{\pi}'$ with $\widetilde{\pi}' \in \mathcal{L}(\mathcal{H}^+)$ such that the observable traces for $\widetilde{\pi}$ and $\widetilde{\pi}'$ are equal. Moreover, for any step i: $\alpha(\widetilde{\pi}[0..i]) = \alpha(\widetilde{\pi}'[0..i])$. The induction base is trivial since both traces originate in the fixed initial state, which corresponds to the initial state of the specification automaton.

For the induction step, suppose the observable traces for $\widetilde{\pi}[0..i]$ and $\widetilde{\pi}'[0..i]$ are equal and $\alpha(\widetilde{\pi}[0..i]) = \alpha(\widetilde{\pi}'[0..i])$. Now, let λ be the last action label and λ' be the next. Since the label combination λ, λ' appears in $\widetilde{\pi}$, it is also present in a mode μ in $\mathcal{H}^+$ by Lemma 4.15 with $\alpha(\widetilde{\pi}[0..i]) = \alpha(\widetilde{\pi}'[0..i]) = \alpha(\mu)$. Further, $\psi(\widetilde{\pi}[0..i])$ has an incoming transition with label λ, hence by Definition 4.11, $\psi(\widetilde{\pi}[0..i]) \sim_{\exists\lambda} \mu$ due to their shared action label. Hence, by Lemma 4.17 the merge preserves this transition, and $\psi(\widetilde{\pi}'[0..i])$ has an outgoing edge with label λ', which proves that the discrete edge is present.

→ Lem. 4.15, p. 172
→ Def. 4.11, p. 166
→ Lem. 4.17, p. 174

Now, for the second part of the induction hypothesis, it suffices to show that $\psi(\widetilde{\pi}[0..i]) = \psi(\widetilde{\pi}'[0..i])$. There are two possibilities: *both* traces take a transition in terms of α or *neither* one does. This follows from the second adequacy criterion of Definition 4.14, and Lemma 4.18 stating that both automata are refinements of $\mathcal{A}^\Phi$.

→ Def. 4.14, p. 171
→ Lem. 4.18, p. 175

If both take such a transition, the claim follows because both automata refine $\mathcal{A}^\Phi$. On the contrary, if neither one takes the transition, both remain in the same state in $\mathcal{A}^\Phi$; the claim follows.

Step 2: The trace pairs $\widetilde{\pi}, \widetilde{\pi}'$ induce an equivalence relation:

$$\sim_+ = \left\{ \left(\psi\left(\widetilde{\pi}[0..i]\right), \psi\left(\widetilde{\pi}'[0..i]\right)\right) \mid i \in \mathbb{N} \right\}$$

This relation is a witness for the claim that $\mathcal{H}|_{\widetilde{\Pi}}$ is a refinement of $\mathcal{H}^+$ due to the trace inclusion proven in Step 1. □

Note that the refinement can be a true refinement since the merge criterion might falsely relate modes that are distinct in $\mathcal{H}|_{\widetilde{\Pi}}$ but share some discrete behavior.

Corollary 4.19. *$\sim_+$ is finer than $\sim_M$.*

Proof The inclusion $\sim_M \subseteq \sim_+$ follows from the theorem. The impossibility of the opposite, i.e., $\sim_+ \not\subseteq \sim_M$ can easily be seen considering the terminal similarity defined in Definition 4.12. This similarity relates every terminal node in the same abstract state. Hence, construction of a counter example is simple, as can be seen in Example 4.3. □

→ Def. 4.12, p. 169
→ Ex. 4.3, p. 154

Theorem 4.4 (*Conservative Construction*)

Let $\mathcal{H}$ be a hybrid automaton with an adequate trace set Π and specification Φ. The constructed automaton $\mathcal{H}^+$ over-approximates the language of the projection automaton $\mathcal{H}|_{\widetilde{\Pi}}$: $\mathcal{L}(\mathcal{H}|_{\widetilde{\Pi}}) \subseteq \mathcal{L}(\mathcal{H}^+)$.

Proof Let $\widetilde{\pi} \in \mathcal{L}(\mathcal{H}|_{\widetilde{\Pi}})$. The proof constructs a trace $\pi \in \mathcal{L}(\mathcal{H}^+)$ such that π is an observable counterpart for $\widetilde{\pi}$. The initial real-valued state of π is $x_0^{\widetilde{\pi}}$ with $\psi(\pi[1]) = \mu_I^+$. By construction, $\mu_I^+ \sim_+ \mu_I|_{\widetilde{\Pi}} = \mu_0^{\widetilde{\pi}}$.

For the induction, consider a delay transition $x_i^{\widetilde{\pi}}, \mu_i^{\widetilde{\pi}}, \delta_i^{\widetilde{\pi}}, x_{i+1}^{\widetilde{\pi}}$ where $\mu_i^{\widetilde{\pi}} \sim_+ \mu^+$ by Theorem 4.3. Let $x_i^{\widetilde{\pi}} + f\delta_i^{\widetilde{\pi}} = x_{i+1}^{\widetilde{\pi}}$ for $f \in \mathbb{R}^n$. By definition of the projection automaton (Definition 4.16), Π contains traces $\pi^\uparrow$ and $\pi^\downarrow$ exhibiting the flow $f^\uparrow$ and $f^\downarrow$ at point $a^\uparrow \leqslant |\pi^\uparrow|$ and $a^\downarrow \leqslant |\pi^\downarrow|$, respectively, while traversing $\mu_i^{\widetilde{\pi}}$ with $f^\downarrow \leqslant f \leqslant f^\uparrow$ due to the linearity of rectangular automata. By the definition of the construction, it at some points $a^\uparrow$ and $a^\downarrow$ constructs two modes $\mu^{a^\uparrow} \in M^+_{a^\uparrow}$ and $\mu^{a^\downarrow} \in M^+_{a^\downarrow}$ with $flow_{a^\uparrow}(\mu^{a^\uparrow}) = f^\uparrow$ and $flow_{a^\downarrow}(\mu^{a^\downarrow}) = f^\downarrow$. Lemmas 4.17 and 4.18 guarantee that further construction steps and merges retain this information. Moreover, Theorem 4.3 implies that $\mu^{a^\uparrow}$, $\mu^{a^\downarrow}$, and μ^+ are equal with respect to $\sim_+$. By Corollary 4.19, they are also equal with respect to $\sim_M$. Thus, $\mathit{flow}(\mu^+) \subseteq \mathit{Conv}(\{\mathit{flow}(\mu) \mid \mu \in \zeta^+\})$ with $f^\downarrow, f^\uparrow, f \in \mathit{flow}(\mu^+)$. As a result, π may contain the subsequence $x_i^{\widetilde{\pi}}, \delta_i^{\widetilde{\pi}}, x_{i+1}^{\widetilde{\pi}}$ representing the delay transition.

→ Thm. 4.3, p. 176
→ Def. 4.16, p. 173
→ Lems. 4.17, 4.18, p. 174 and 175
→ Thm. 4.3, p. 176
→ Cor. 4.19, p. 176

For discrete transitions, consider $e = (\mu_i^{\widetilde{\pi}}, \lambda_i^{\widetilde{\pi}}, \mu_{i+1}^{\widetilde{\pi}})$. We show that $e^+ = (\mu_s^+, \lambda_i^{\widetilde{\pi}}, \mu_t^+)$ is a valid transition assuming that $\mu_s^+ = \psi(\pi[0..i])$, i.e., the trace constructed so far ended in μ_s^+. There are three cases:

Case a) Both $\mu_i^{\widetilde{\pi}} \sim_\Phi \mu_{i+1}^{\widetilde{\pi}}$ and $\mu_s^+ \sim_\alpha \mu_t^+$. Intuitively, this means that both automata remain in the same state of the specification automaton. In this case, by construction: $\gamma_i(e^+) = \Gamma^\Phi(\alpha_i(e^+)) = \mathbb{1}$ and by Lemmas 4.17 and 4.18: $\gamma_i(e^+) \subseteq \gamma^+(e^+)$. Thus, the guard is satisfied trivially. The existence of the edge in the constructed automaton follows from Theorem 4.3.

→ Lems. 4.17, 4.18, p. 174 and 175
→ Thm. 4.3, p. 176

Case b) Neither $\mu_i^{\widetilde{\pi}} \sim_\Phi \mu_{i+1}^{\widetilde{\pi}}$ nor $\mu_s^+ \sim_\alpha \mu_t^+$. Intuitively, this means neither automaton remains in the same state of the specification automaton. In this case, $\gamma_i(e^+) = \Gamma^\Phi(\alpha_i(e^+))$. By the second criterion of Definitions 4.14 and 4.16, we know that $\gamma|_{\widetilde{\Pi}}(e) \implies \gamma(e)$ and $\gamma(e) \implies \Gamma^\Phi(\alpha_i(e^+))$. Again, by Lemma 4.18 and Lemma 4.17 we know $\gamma_i(e^+) \subseteq \gamma^+(e^+)$ and the existence of the edge in the constructed automaton follows from Theorem 4.3.

→ Defs. 4.14, 4.16, p. 171 and 173
→ Lem. 4.18, p. 175
→ Lem. 4.17, p. 174
→ Thm. 4.3, p. 176

Case c) Either $\mu_i^{\widetilde{\pi}} \not\sim_\Phi \mu_{i+1}^{\widetilde{\pi}}$ or $\mu_s^+ \not\sim_\alpha \mu_t^+$ but not both. This case is impossible for adequate specifications (Definition 4.14, Item 1) and by the definition of $\sim_\Phi$ and $\sim_\alpha$.

→ Def. 4.14, p. 171

Thus, the discrete transition exists and is applicable in the reconstructed automaton. This concludes the proof. □

4.5. Experiments

The empirical evaluation shows the scalability and precision of the approach. It is based on a prototype implementation in Rust. All experiments were conducted on an Intel i5-7200u with 8 GB of Ram.

4.5.1. Aircraft System

As a first proof of concept, consider the running example from Example 4.3. For adequate input traces, the output will always be structurally equal with varying dynamics. This can be seen in Figure 4.5, which shows the results of two construction processes. The dynamics in black are constructed from three hand-picked traces of length eight. Two of these traces traverse all three Travel modes, whereas the last one skips the course adjustment modes and loops in one of the Landing modes instead. As can be seen, by picking the state values for the traces in such a way that they represent the extreme behavior, the reconstruction of the dynamics is perfect. Conversely, the constructed dynamics based on an adequate trace set of ten traces obtained by conducting random walks on the original system is shown in gray. The traces can be found in Appendix A.3. Evidently, the reconstruction closely resembles the original system both structurally and in terms of dynamics despite being based on a small set of random traces.

→ Fig. 4.5, p. 168

→ Appx. A.3, p. 221

4.5.2. Scalability

Recall the complexity of each step of the construction algorithm, i.e., extraction, construction, and merging, from Section 4.3.4. The extraction only requires a single pass over the specification and is thus negligible. The construction and merges depend on the dimensionality of the system, and the number and length of traces. The merge also depends on the number of equivalence classes with respect to $\sim_M$ in the best case, which is the size of the output automaton.

→ Sec. 4.3.4, p. 170

For this reason, the scalability evaluation considers exactly these three factors: dimensionality, number and length of traces, and output-size. To this end, it automatically generates an automaton with matching specification and adequate trace set. The automaton is shaped like a binary tree of variable depth d (scales the length of traces) where each of the $2^{d+1} - 1$ nodes is a control mode with dynamics of variable dimension (scales the dimensionality). The specification summarizes a variable number of modes with equal depth (scales the output size) and generates a variable number of adequate traces (scales the number of traces) enabling the respective merges.

Output Size For the impact of the output size, consider Figure 4.6 depicting the running time (blue lines) and memory consumption (orange lines) for varying sizes of the original automaton. The dark blue and dark orange line represent runs where $\sim_M$ only equates identities, prohibiting any merges. For the lighter lines, two modes are equal if they have

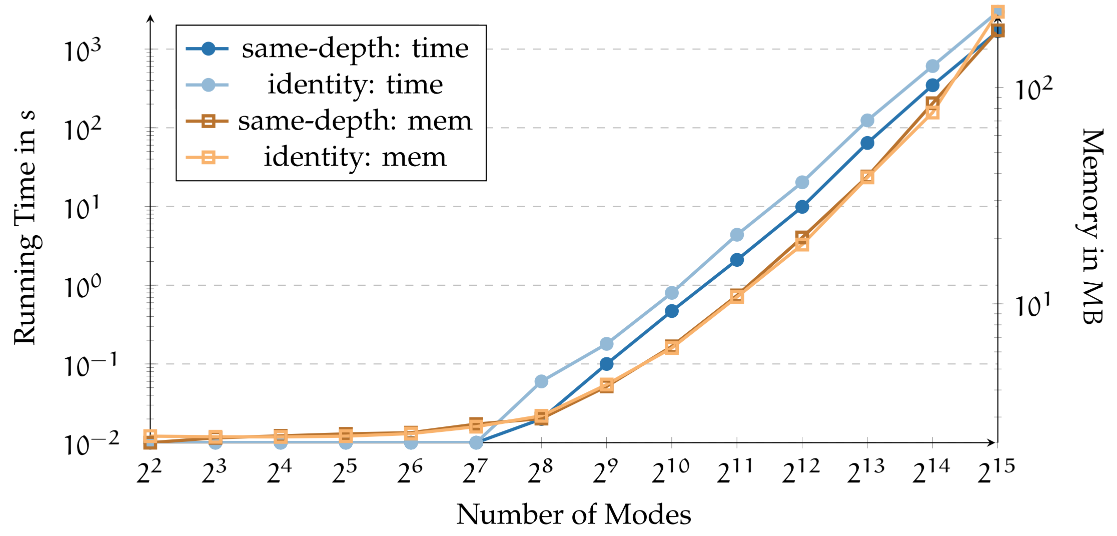


Figure 4.6.: The results of the scalability analysis for different sizes of the original automaton and specifications enabling many (darker lines) or no (lighter lines) merges.

the same depth in the underlying automaton, allowing for vast merges. The number and size of the traces required for an adequate trace set scales exponentially with the depth of the automaton, accounting for the exponential increase in both metrics. Independent of the existence of merges, the running time lies below a second for automata with less than 2^{10} modes and terminated after less than an hour (identity) or half an hour (same-depth) for automata with 2^{15} modes. Hence, the merge behavior has a significant impact on the running time, yet changes nothing on the asymptotic complexity, as expected.

The memory consumption scales similarly, starting to rise significantly around 2^7 owing both to the increased number of traces stored in memory, and the resulting size of $\mathcal{H}^+_{|\Pi|}$. Note that the memory consumption almost exclusively stems from the construction process; merging only de-allocates memory. Since the reported memory consumption is the maximum of memory allocated at a point in time, the merge behavior makes no difference whatsoever.

Also note that the running time of the merges absolutely dominates the overall running time. At a size of 2^9 modes the running time of the construction process merely amounts to 3.061% and further decreases to 0.015% for automata with 2^{15} modes. This trend holds for both lines of experiments.

Dimensionality The dimensionality impacts the running time to a lesser extent as can be seen in Figure 4.7. Here, the number of modes is a constant $2^{11} - 1$. Raising the dimension from 1,000 to 7,000 for an automaton size of 10^{10} increases the running time

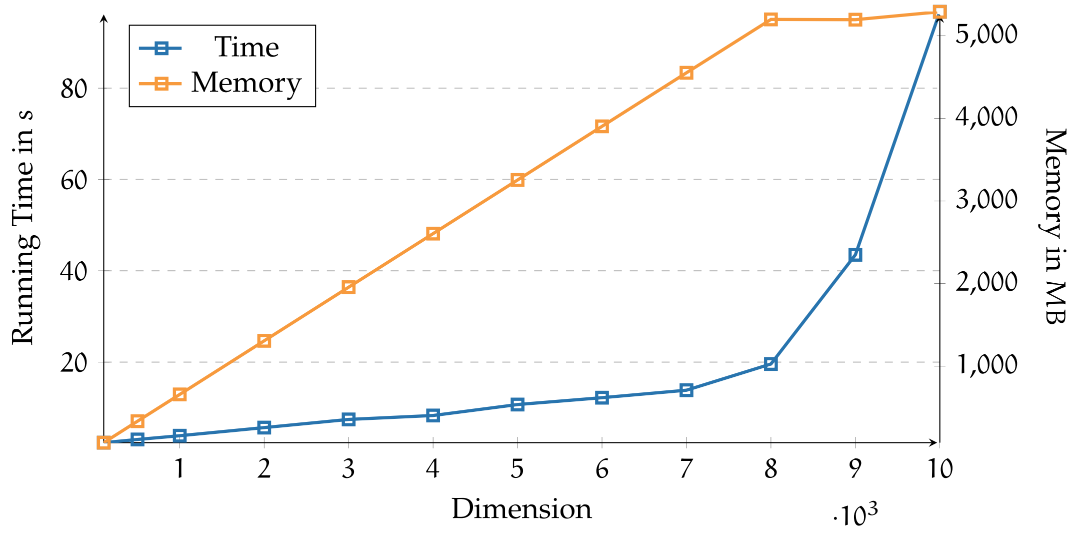


Figure 4.7.: Running time and space consumption of the reconstruction for varying dimensions. The original automaton has $2^{11} - 1$ modes and each of the around 1,000 trace has length eleven.

from around 4 s to 14 s. The limiting factor here is the memory consumption: each additional dimension increases the memory consumption of every guard condition, mode, and trace step. As a result, 10,000 dimensions requires around 5 GB of memory, which is also reflected in a relatively steep increase in running time to 96 s. Here, the running time starts to explode as soon as the machine can no longer efficiently handle the memory at slightly over 5 GB. At this point, the machine starts swapping, resulting in the plateau in the memory consumption, i.e., the resident set size, as well as the steep incline in running time. It stands to reason that a larger RAM would allow the running time to continue to scale linearly above 8,000 dimensions.

Trace Size Lastly the number of traces has an almost identical impact on both the running time and memory consumption, as illustrated in Figure 4.8. Here, the automata are five-dimensional with a constant number of modes and varying number of traces, each having a length of eleven. The scale of the impact lies in-between the one of the dimensionality and the size of the output. Raising the number of traces from 3,000 to 16,000 increases the running time roughly 40-fold.

4.5.3. Comparison Against Other Approaches

The construction of a hybrid automaton requires the determination of both the discrete structure and continuous behavior. Regarding the former, the conservative construction relies on the information provided by the specification and refines abstract states according to trace information. This restricts revisions to a local level within an abstract

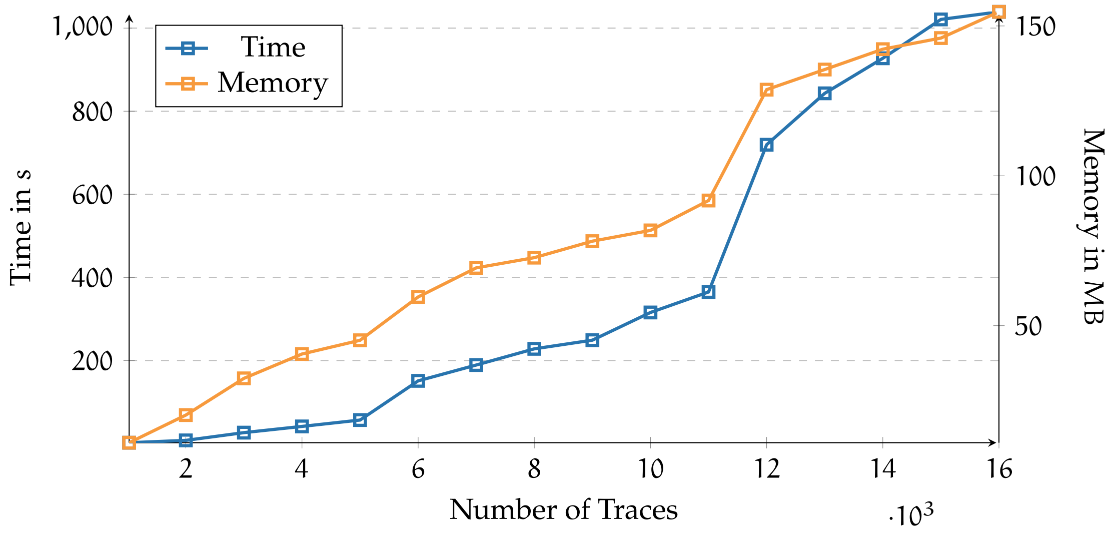


Figure 4.8.: Performance of the reconstruction for a varying number of traces. The original automaton has $2^{11} - 1$ modes and five dimensions. Each trace has length eleven.

state. In absence of such a specification, other approaches resort to learning algorithms. Medhat et al. [Med+15] use a modification of Angluin's L^* algorithm [Ang87] to learn the discrete structure separately from the dynamics while Tappler et al. [Tap+19] learn a timed automaton with genetic programming. We will evaluate the conservative construction against both of these approaches. Note that the results are not fully comparable since neither approach has a specification automaton to start with, hence it merely provides a rough overview.

Angluin's L^* and Clustering. Medhat et al. [Med+15] use an adaptation of L^* and clustering to identify the discrete and continuous behavior of a system, respectively. In their case study, they use a Simulink model of a closed-loop engine timing control system. Figure 4.9 shows an approximate representation of the system as hybrid model. Their construction uses eight traces to generate an automaton that resembles the underlying system for new traces up to an error of 2.6%. The conservative construction can perfectly reconstruct the automaton with one hand-picked trace of length ten within less than 1 ms. When using random walks, an average of 35 traces of length fifteen suffices for the perfect reconstruction. In this example, the specification always summarizes modes belonging to a certain operation of the system, i.e., a drop, jump, ramp-up and the stable configuration.

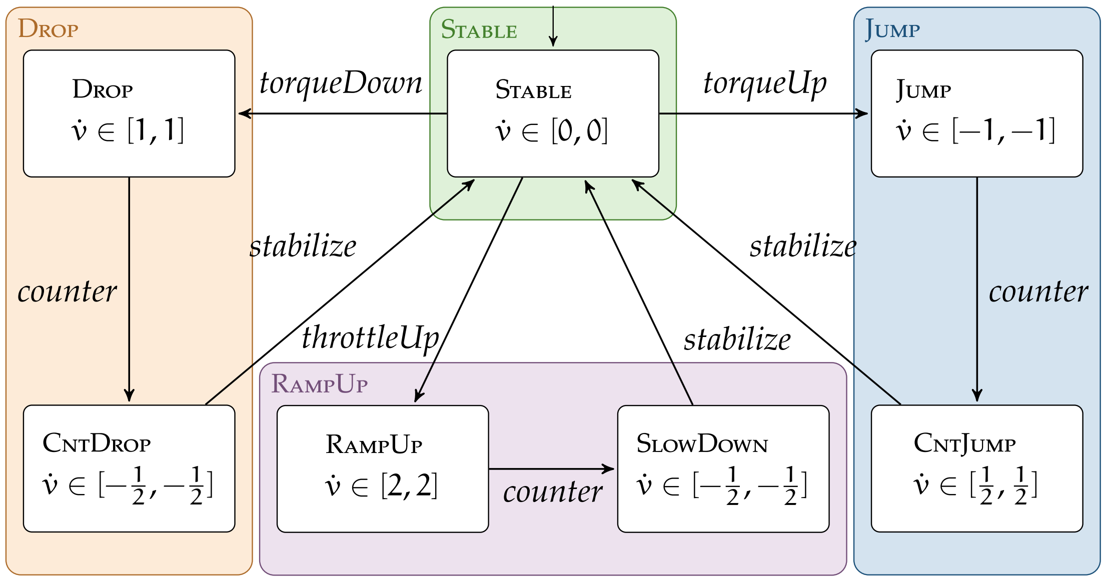


Figure 4.9.: Hybrid automaton approximating the engine timing control system by Medhat et al. [Med+15]. A single trace of length ten enables perfect reconstruction. Colored states indicate the specification automaton.

Genetic Programming. Tappler et al. [Tap+19] use genetic programming to successively adapt a candidate automaton to encompass all input traces. As an example, they consider a timed automaton modeling a car alarm system displayed in Figure 4.10. A sufficiently precise reconstruction requires 2,000 randomly generated traces and took a mean of around $100\,\mathrm{min}$. When using seven hand-selected traces, the conservative construction can perfectly reconstruct the system within less than $1\,\mathrm{ms}$, disregarding resets. With random walks, an average of 35 traces of length 15 is necessary for the perfect reconstruction.

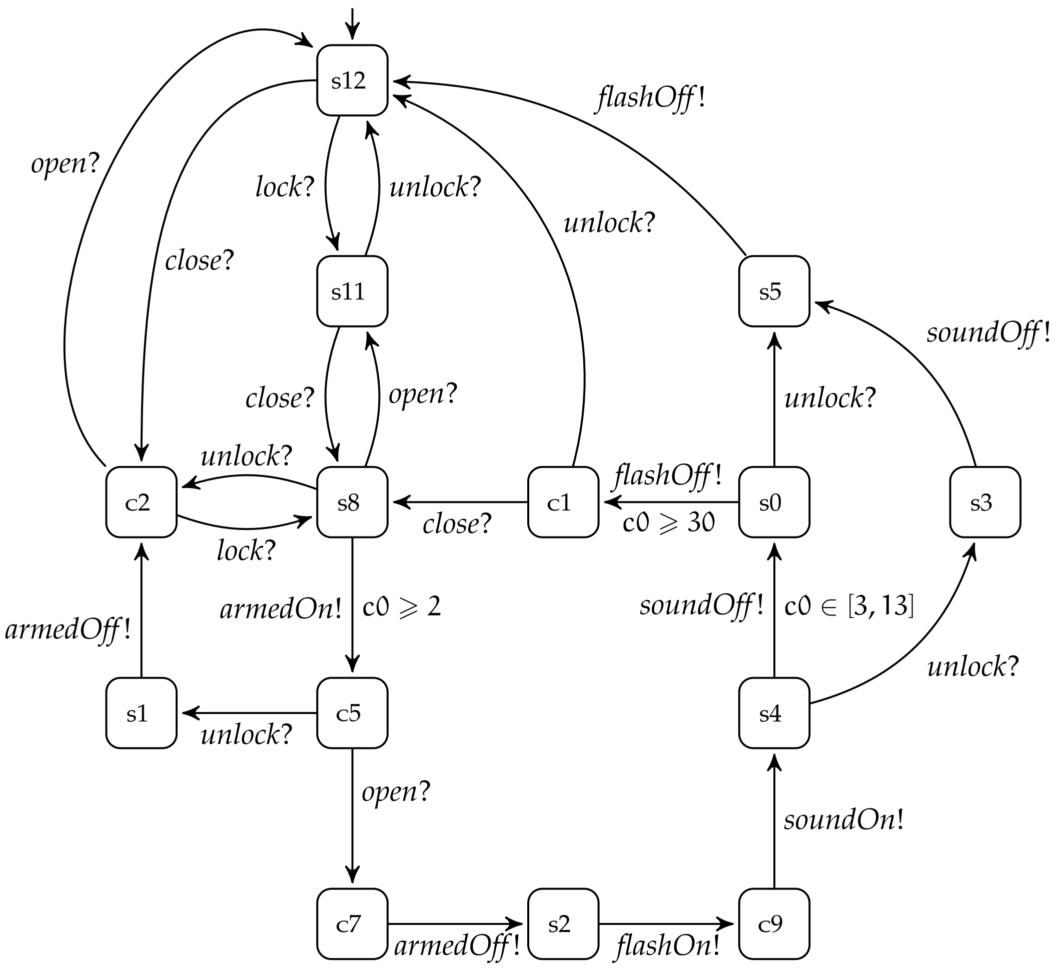


Figure 4.10.: Timed automaton for the car alarm system by Tappler et al. [Tap+19]. Perfect reconstruction requires at least seven traces of length twelve, on average 35 traces of length 15.

4.6. Recapitulation

This chapter presented a construction algorithm for conservative hybrid automata from development artifacts in the shape of a runtime monitoring specification and pre-recorded execution traces. The construction is validated mathematically by proving that the result is an over-approximation under certain assumptions on the inputs. An additional empirical evaluation revealed both the extraordinary scalability of the construction and that even randomly generated inputs regularly satisfy the input requirements. Considering that — in a realistic setting — these inputs are high-quality artifacts acquired during development of the system, they should no longer be left under-utilized. Treating them as the valuable assets they are allows for constructing precise, conservative hybrid automata in a scalable fashion. These automata then become assets themselves to better comprehend the system, perform a static analysis, or use them during runtime for prediction.

4.7. Related Work

The theory of hybrid automata was first studied by Henzinger [Hen96] as the real-time extension of timed automata [AD94]. Learning the complex structure of timed and hybrid automata is a line of research that resulted in deterministic and stochastic reconstruction algorithms.

Niggemann et al. [Nig+12] present the tool HyBUTLA that builds prefix trees of the traces and applies merges when appropriate. Since Angluin's L* algorithm [Ang87] is a prominent solution for learning discrete automata, several extensions for timed automata were proposed [GJL04; An+19]. Based on that, Medhat et al. [Med+15] split the learning process of a hybrid automaton into two steps. They first learn the discrete model of the automaton with L* and then capture the dynamics using clustering. Both of these techniques can potentially be replaced or integrated into different frameworks. Hence, their approach is complementary to the conservative construction: substituting the clustering for the simpler solve function can yield better precision at the price of conservativeness. Focusing on the medical application domain, HyMN [LBG18] learns patient specific parameters for hybrid automata deterministically. Soto et al. [Sot+19] synthesize a hybrid automaton with an online algorithm without relying on a specification as discrete template. While precision is very high and completeness is shown, learning a trace prompts a global analysis of the previously learned hybrid automaton, which incurs performance penalties. The conservative construction avoids this complexity by using the specification automaton and the adequacy criterion. This way, revisions are local and still retain correctness guarantees. Other approaches for learning hybrid automata using mathematical models for node identification were proposed by Summerville et al. [SOM17] and Breschi et al. [BPB16].

If large datasets of traces are available, stochastic learning of hybrid automata is feasible. Tappler et al. [Tap+19] use genetic programming to reconstruct timed automata both in an offline and online setting [APT20]. Santana et al. [San+15] build hybrid automata with the Expectation-Maximization algorithm to iteratively define the model parameters. An unsupervised learning approach was presented by Lee et al. [Lee+17], whereas Birgelen and Niggemann [BN17] use self-organizing feature maps. Despite the success of machine learning, the results do not provide provable guarantees.

Chapter 5

Conclusion

This thesis has presented a runtime monitoring approach with a strong focus on safety and comprehension. The language is syntactically similar to common programming languages, emphasizing clarity, declarativeness, and modularity over conciseness. The formal basis and strong type system enables a set of intricate analyses to support the specifier. Further, they increase confidence in the specification and yield artifacts providing insights into the specification as well as monitors for it. Two compilers realize a monitor for a given specification either as a hardware description or Rust code with verification annotations enabling automatic verification. Apart from the sheer monitoring task, a specification of a system coupled with test traces thereof enable the construction of a conservative model of the underlying system.

Recent practical projects and cooperations prove the relevance of RTLola for the avionic industry. The projects integrated RTLola into autonomous aircraft of the German Aerospace Center and prototypes of a leading German multicopter manufacturer. Part of these projects are also discussions with the European Union Aviation Safety Agency. The goal is to determine concrete criteria for the certification of the monitor, as well as the impact it has on the certification of the system as a whole. Here, detailed static information regarding the specification and its monitor are valuable assets since they boost confidence and comprehension.

The significance of such components is likely to increase in the foreseeable future. Essentially, a system has a small core of safety-critical, comprehensive components such as a runtime monitor and a backup controller. This backup controller can be a low-performance component only capable of carrying out the most basic tasks. Yet, it does so safely. This safe core enables the deployment of additional high-performance components such as machine-learned controllers without the need to understand and test every detail of them. During regular operation, the high-performance controller is in charge. Only when the monitor detects a problem, it switches over to the safety controller. As a result, the safety of the overall system is ensured without fully trusting the efficient controller. This emphasizes the value of future work in the direction of comprehensible runtime verification for cyber-physical systems.

5.1. Future Directions

There are three major directions for future work building upon the results of this thesis.

Practicality

The first and most evident is the continuation of work integrating monitoring systems like RTLola into real, *practical applications*. Such work provides crucial feedback for recent developments in runtime verification research. It reveals what key characteristics a monitor needs to bring to reliably support specifiers and to enable the certification of the system with the help of the monitor. Here, avionics is a suitable target due to its strong focus on safety. However, the increasing autonomy in the automotive industry also pushes software safety concerns in the foreground. Many constraints on components for aircraft translate equally or similarly from avionics to automotive. This also holds for medical cyber-physical systems, in particular for wearable ones like implanted pacemakers. While safety plays a major role in stationary devices like radiation therapy machines, too, they are less affected by resource constraints.

Usability

The next direction concerns empirical research broadly regarding the *usability* of runtime monitoring. There are two major interaction points between users and the system. The following uses RTLola as an example, yet these points can be generalized. First, the input format. So far, the basis for RTLola is a textual — and thus linear — specification, which the toolkit interprets literally. However, this is not necessarily optimal. For example, it is easier to grasp the general structure of a specification based on the two-dimensional representation of the dependency graph than a raw specification. Hence, it stands to reason that a similarly graph-inspired visual input format might improve the user experience of specifiers. The market dominance of MATLAB/Simulink is indicative for this claim. Such an interface can also be interactive, enriching the specification with additional information for the users, such as the results of specification analysis steps. A similar feat is widely adopted in integrated development environments like Microsoft's Visual Studio Code or JetBrains' IntelliJ IDEA. Here, a user study could determine which representation better supports developers in their quest to find logical errors in a specification.

The second vector for user interaction is the handling of outputs. Textual output in form of trigger messages and output values is perfectly suited for an interaction with the system itself. However, this form of communication is lacking when interacting with a human. An autonomous drone, for example, operates normally without immediate supervision. Upon detection of a problem, though, it contacts a central hub such that it can perform a handover to a remote human pilot. Here, it is paramount that the pilot immediately detects the source of the problem without rummaging through logs in text format. Hence, the monitor needs to filter and present its own outputs appropriately.

Last, there are plenty of open questions regarding the quality of monitoring itself. One direction particularly interesting for RTLola is model-based *prediction* and introspection. Here, an RTLola monitor has access to the conservative hybrid automaton. Mapping start variables of the model to input streams allows the monitor to estimate the continuous development of these streams. Naturally, the time horizon of predictions needs to be low since their quality degrades over time. The reason is that the monitor has to predict the path of the system through the automaton. Any non-determinism in the automaton thus leads to a drop in accuracy, adding up over time. However, when considering the asynchronous setting of RTLola, the prediction can serve another purpose. Suppose the monitor receives inputs from several sensors until one of them ceases to produce values. It can now predict the current value of the input based on the model. However, rather than blindly guessing or over-approximating the path through the automaton, it can base its prediction on the values received from other sensors. This has the potential to drastically increase the quality of the estimate.

Prediction

The model also enables *introspection* regarding security and error estimation. For the former suppose a monitor receives regular updates with no apparent anomaly. Estimating the expected evolution of input streams with respect to the model can reveal that one or several sensors deliver spurious values. This could indicate malfunction or a security breach. For the error estimation, consider the following: A monitor receives regular updates from all available sensors until one of them ceases to produce values frequently. As a result, sample and hold accesses to the affected stream produce less valuable information. The model enables the monitor to estimate whether these computations still yield realistic values, i.e., estimate the error. If the error grows too large, it should mark them as unreliable when disseminating outputs to the system.

Introspection

In summary, while this thesis provides a foundation for comprehensible runtime monitoring for cyber-physical systems, the work is far from complete. The aforementioned topics are but a fraction of open questions and research directions. Further work can drastically increase applicability and effectiveness of practically relevant monitoring and thus lead to a safer future.

Bibliography

[Ado+17] Florian-Michael Adolf, Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Christoph Torens. “Stream Runtime Monitoring on UAS”. In: *Rv 2017*. Vol. 10548. Lncs. Springer, 2017, pp. 33–49. DOI: `10.1007/978-3-319-67531-2\_3`.

[Ado+18] Florian-Michael Adolf, Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Christoph Torens. “Stream Runtime Monitoring on UAS”. In: *CoRR* abs/1804.04487 (2018). arXiv: `1804.04487`. URL: `http://arxiv.org/abs/1804.04487`.

[APT20] Bernhard K. Aichernig, Andrea Pferscher, and Martin Tappler. “From Passive to Active: Learning Timed Automata Efficiently”. In: *NASA Formal Methods - 12th International Symposium, NFM 2020, Moffett Field, CA, USA, May 11-15, 2020, Proceedings*. Ed. by Ritchie Lee, Susmit Jha, and Anastasia Mavridou. Vol. 12229. Lecture Notes in Computer Science. Springer, 2020, pp. 1–19. DOI: `10.1007/978-3-030-55754-6\_1`.

[Alm+10] José Bacelar Almeida, Endre Bangerter, Manuel Barbosa, Stephan Krenn, Ahmad-Reza Sadeghi, and Thomas Schneider. “A certifying compiler for zero-knowledge proofs of knowledge based on σ-protocols”. In: *European Symposium on Research in Computer Security*. Springer. 2010, pp. 151–167.

[Alq+18] Sarra Alqahtani, Ian Riley, Samuel Taylor, Rose Gamble, and Roger Mailler. “MTL Robustness for Path Planning with A”. In: *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*. 2018, pp. 247–255.

[AD94] Rajeev Alur and David L. Dill. “A Theory of Timed Automata”. In: *Theor. Comput. Sci.* 126.2 (1994), pp. 183–235. DOI: `10.1016/0304-3975(94)90010-8`.

[An+19] Jie An, Mingshuai Chen, Bohua Zhan, Naijun Zhan, and Miaomiao Zhang. “Learning One-Clock Timed Automata”. In: *CoRR* abs/1910.10680 (2019). arXiv: `1910.10680`. URL: `http://arxiv.org/abs/1910.10680`.

[Ang87] Dana Angluin. “Learning Regular Sets from Queries and Counterexamples”. In: *Inf. Comput.* 75.2 (1987), pp. 87–106. DOI: `10.1016/0890-5401(87)90052-6`.

[App01] Andrew W Appel. “Foundational proof-carrying code”. In: *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*. Ieee. 2001, pp. 247–256.

[AF00] Andrew W Appel and Amy P Felty. “A semantic model of types and machine instructions for proof-carrying code”. In: *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2000, pp. 243–253.

[ACM02] Eugene Asarin, Paul Caspi, and Oded Maler. “Timed regular expressions”. In: *J. Acm* 49.2 (2002), pp. 172–206. DOI: `10.1145/506147.506151`.

[Ast+19] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. “Leveraging Rust Types for Modular Specification and Verification”. In: *Proc. ACM Program. Lang.* 3.Oopsla (2019), 147:1–147:30. DOI: `10.1145/3360573`.

[Bar+04] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. “Rule-Based Runtime Verification”. In: *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, Italy, January 11-13, 2004, Proceedings*. Ed. by Bernhard Steffen and Giorgio Levi. Vol. 2937. Lecture Notes in Computer Science. Springer, 2004, pp. 44–57. DOI: `10.1007/978-3-540-24622-0\_5`.

[BRH10] 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`.

[BF18] Ezio Bartocci and Yliès Falcone, eds. *Lectures on Runtime Verification - Introductory and Advanced Topics*. Vol. 10457. Lecture Notes in Computer Science. Springer, 2018. ISBN: 978-3-319-75631-8. DOI: `10.1007/978-3-319-75632-5`.

[Bas+20] David A. Basin, Thibault Dardinier, Lukas Heimes, Srdan Krstic, Martin Raszyk, Joshua Schneider, and Dmitriy Traytel. “A Formally Verified, Optimized Monitor for Metric First-Order Dynamic Logic”. In: *Ijcar 2020*. Ed. by Nicolas Peltier and Viorica Sofronie-Stokkermans. Vol. 12166. Lncs. Springer, 2020, pp. 432–453. DOI: `10.1007/978-3-030-51074-9\_25`.

[Bas+15] 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`.

[BKT17] David A. Basin, Srdjan Krstic, and Dmitriy Traytel. “AERIAL: Almost Event-Rate Independent Algorithms for Monitoring Metric Regular Properties”. In: *RV-CuBES 2017*. 2017, pp. 29–36.

[BLS07] Andreas Bauer, Martin Leucker, and Christian Schallhart. “The Good, the Bad, and the Ugly, But How Ugly Is Ugly?” In: *Rv 2007*. Vol. 4839. Lncs. Springer, 2007, pp. 126–138. DOI: `10.1007/978-3-540-77395-5\_11`.

[BLS11] Andreas Bauer, Martin Leucker, and Christian Schallhart. “Runtime Verification for LTL and TLTL”. In: *ACM Trans. Softw. Eng. Methodol.* 20.4 (2011), 14:1–14:64. DOI: `10.1145/2000799.2000800`.

[Bau20] Jan Baumeister. “Tracing Correctness: A Practical Approach to Traceable Runtime Monitoring”. Master Thesis. Saarland University, 2020.

[Bau+21] Jan Baumeister, Bernd Finkbeiner, Matthis Kruse, Stefan Oswald, Noemi Passing, and Maximilian Schwenger. *Automatic Optimizations for Runtime Verification Specifications*. 2021. URL: `https://www.react.uni-saarland.de/publications/BFKOPS21.pdf`.

[Bau+20a] Jan Baumeister, Bernd Finkbeiner, Sebastian Schirmer, Maximilian Schwenger, and Christoph Torens. “RTLola Cleared for Take-Off: Monitoring Autonomous Aircraft”. In: *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*. Ed. by Shuvendu K. Lahiri and Chao Wang. Vol. 12225. Lecture Notes in Computer Science. Springer, 2020, pp. 28–39. DOI: `10.1007/978-3-030-53291-8\_3`.

[Bau+19a] Jan Baumeister, Bernd Finkbeiner, Maximilian Schwenger, and Hazem Torfah. “FPGA Stream-Monitoring of Real-time Properties”. In: *ACM Trans. Embedded Comput. Syst.* 18.5s (2019), 88:1–88:24. DOI: `10.1145/3358220`.

[Bau+19b] Jan Baumeister, Bernd Finkbeiner, Maximilian Schwenger, and Hazem Torfah. *On the Similarities of Aircraft and Humans: Monitoring CPS with StreamLAB*. 2019. URL: `https://www.react.uni-saarland.de/publications/cybercardia19.pdf`.

[Bau+20b] Jan Baumeister, Florian Kohn, Stefan Oswald, Malte Schledjewski, Maximilian Schwenger, and Leander Tentrup. *RTLola Frontend*. `https://docs.rs/rtlola-frontend/`. Accessed: 06.01.2022. 2020.

[BBC18] BBC. *Audi chief Rupert Stadler arrested in diesel emissions probe*. Online; accessed: 2020-10-15. 2018. URL: `https://www.bbc.com/news/business-44517753`.

[Ber16] Gerard Berry. “Formally Unifying Modeling and Design for Embedded Systems - A Personal View”. In: *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016,* *Proceedings, Part II*. 2016, pp. 134–149. doi: 10.1007/978-3-319-47169-3_11.

[BG92] Gérard Berry and Georges Gonthier. "The Esterel Synchronous Programming Language: Design, Semantics, Implementation". In: *Sci. Comput. Program.* 19.2 (1992), pp. 87–152. doi: 10.1016/0167-6423(92)90005-v.

[BJP06] Frédéric Besson, Thomas P. Jensen, and David Pichardie. "Proof-Carrying Code from Certified Abstract Interpretation and Fixpoint Compression". In: *Theor. Comput. Sci.* 364.3 (2006), pp. 273–291. doi: 10.1016/j.tcs.2006.08.012.

[Bie+21] Sebastian Biewer, Bernd Finkbeiner, Holger Hermanns, Maximilian A. Köhl, Yannik Schnitzer, and Maximilian Schwenger. "RTLola on Board: Testing Real Driving Emissions on your Phone". In: *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II*. Ed. by Jan Friso Groote and Kim Guldstrand Larsen. Vol. 12652. Lecture Notes in Computer Science. Springer, 2021, pp. 365–372. doi: 10.1007/978-3-030-72013-1_20.

[BN17] Alexander von Birgelen and Oliver Niggemann. "Using self-organizing maps to learn hybrid timed automata in absence of discrete events". In: *22nd IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2017, Limassol, Cyprus, September 12-15, 2017*. Ieee, 2017, pp. 1–8. doi: 10.1109/etfa.2017.8247695.

[bm15] bluss and mitchmindtree. *Petgraph: Graph Data Structure Library*. https://github.com/petgraph/petgraph. Accessed: 06.01.2022. 2015.

[Boh+18] Brandon Bohrer, Yong Kiam Tan, Stefan Mitsch, Magnus O. Myreen, and André Platzer. "VeriPhy: verified controller executables from verified cyber-physical system models". In: *Pldi 2018*. Ed. by Jeffrey S. Foster and Dan Grossman. Acm, 2018, pp. 617–630. doi: 10.1145/3192366.3192406.

[BZ08] Marc Boule and Zeljko Zilic. "Automata-based assertion-checker synthesis of PSL properties". In: *ACM Trans. Design Autom. Electr. Syst.* 13.1 (2008), 4:1–4:21. doi: 10.1145/1297666.1297670.

[Bou+17] Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. "A formally verified compiler for Lustre". In: *Pldi 2017*. Ed. by Albert Cohen and Martin T. Vechev. Acm, 2017, pp. 586–601. doi: 10.1145/3062341.3062358.

[BPB16] Valentina Breschi, Dario Piga, and Alberto Bemporad. "Learning hybrid models with logical and continuous dynamics via multiclass linear separation". In: *55th IEEE Conference on Decision and Control, CDC 2016, Las Vegas, NV, USA, December 12-14, 2016*. Ieee, 2016, pp. 353–358. DOI: 10.1109/cdc.2016.7798294.

[Bro19] Graham Brooker. "Chapter Eleven - The Artificial Pancreas". In: *Handbook of Biomechatronics*. Ed. by Jacob Segil. Academic Press, 2019, pp. 405–456. ISBN: 978-0-12-812539-7. DOI: https://doi.org/10.1016/B978-0-12-812539-7.00015-5.

[CGS20] Martín Ceresa, Felipe Gorostiaga, and César Sánchez. "Declarative Stream Runtime Verification (hLola)". In: *Programming Languages and Systems - 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 - December 2, 2020, Proceedings*. Ed. by Bruno C. d. S. Oliveira. Vol. 12470. Lecture Notes in Computer Science. Springer, 2020, pp. 25–43. DOI: 10.1007/978-3-030-64437-6_2.

[CM20] Agnishom Chattopadhyay and Konstantinos Mamouras. "A Verified Online Monitor for Metric Temporal Logic with Quantitative Semantics". In: *Runtime Verification - 20th International Conference, RV 2020, Los Angeles, CA, USA, October 6-9, 2020, Proceedings*. Ed. by Jyotirmoy Deshmukh and Dejan Nickovic. Vol. 12399. Lecture Notes in Computer Science. Springer, 2020, pp. 383–403. DOI: 10.1007/978-3-030-60508-7_21.

[Che+07] Yiyun Chen, Lin Ge, Baojian Hua, Zhaopeng Li, Cheng Liu, and Zhifang Wang. "A pointer logic and certifying compiler". In: *Frontiers of Computer Science in China* 1.3 (2007), pp. 297–312.

[Cho95] Jan Chomicki. "Efficient Checking of Temporal Integrity Constraints Using Bounded History Encoding". In: *ACM Trans. Database Syst.* 20.2 (1995), pp. 149–186. DOI: 10.1145/210197.210200.

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. "A Tool for Checking ANSI-C Programs". In: *Tacas 2004*. Vol. 2988. Lncs. Springer, 2004, pp. 168–176. DOI: 10.1007/978-3-540-24730-2_15.

[CLN00] Christopher Colby, Peter Lee, and George C. Necula. "A Proof-Carrying Code Architecture for Java". In: *Cav 2000*. Vol. 1855. Lncs. Springer, 2000, pp. 557–560. DOI: 10.1007/10722167_44.

[Col+00] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. "A Certifying Compiler for Java". In: *Pldi 2000*. Acm, 2000, pp. 95–107. DOI: 10.1145/349299.349315.

[Con+18a] Lukas Convent, Sebastian Hungerecker, Martin Leucker, Torben Scheffel, Malte Schmitz, and Daniel Thoma. “TeSSLa: Temporal Stream-Based Specification Language”. In: *Sbmf 2018*. Vol. 11254. Lncs. Springer, 2018, pp. 144–162. DOI: `10.1007/978-3-030-03044-5\_10`.

[Con+18b] Lukas Convent, Sebastian Hungerecker, Torben Scheffel, Malte Schmitz, Daniel Thoma, and Alexander Weiss. “Hardware-Based Runtime Verification with Embedded Tracing Units and Stream Processing”. In: *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings*. 2018, pp. 43–63. DOI: `10.1007/978-3-030-03769-7\_5`.

[DAn+05] Ben D’Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, and Zohar Manna. “Lola: Runtime Monitoring of Synchronous Systems”. In: *Time 2005*. Burlington, Vermont: IEEE Computer Society Press, June 2005, pp. 166–174.

[Dah+05] Anat Dahan, Daniel Geist, Leonid Gluhovsky, Dmitry Pidan, Gil Shapir, Yaron Wolfsthal, Lyes Benalycherif, Romain Kamdem, and Younes Lahbib. “Combining System Level Modeling with Assertion Based Verification”. In: *6th International Symposium on Quality of Electronic Design (ISQED 2005), 21-23 March 2005, San Jose, CA, USA*. 2005, pp. 310–315. DOI: `10.1109/isqed.2005.32`.

[Dam84] Luis Damas. “Type assignment in programming languages”. PhD Thesis. 1984.

[DFS21] Johann C Dauer, Bernd Finkbeiner, and Sebastian Schirmer. “Monitoring with Verified Guarantees”. In: *International Conference on Runtime Verification*. Springer. 2021, pp. 62–80.

[Des+17] Jyotirmoy V. Deshmukh, Alexandre Donzé, Shromona Ghosh, Xiaoqing Jin, Garvit Juniwal, and Sanjit A. Seshia. “Robust online monitoring of signal temporal logic”. In: *Formal Methods in System Design* 51.1 (2017), pp. 5–30. DOI: `10.1007/s10703-017-0286-7`.

[Din06] Din. *Bahnanwendungen*. Din 60880. Berlin, Germany: Deutsches Institut für Normung, 2006.

[DFM13a] Alexandre Donzé, Thomas Ferrere, and Oded Maler. “Efficient robust monitoring for STL”. In: *International Conference on Computer Aided Verification*. Springer. 2013, pp. 264–279.

[DFM13b] Alexandre Donzé, Thomas Ferrère, and Oded Maler. “Efficient Robust Monitoring for STL”. In: *Cav 2013*. Vol. 8044. Lncs. Springer, 2013, pp. 264–279. DOI: `10.1007/978-3-642-39799-8\_19`.

[DM10] Alexandre Donzé and Oded Maler. “Robust satisfaction of temporal logic over real-valued signals”. In: *International Conference on Formal Modeling and Analysis of Timed Systems*. Springer. 2010, pp. 92–106.

[Dru00] Doron Drusinsky. “The Temporal Rover and the ATG Rover”. In: *SPIN Model Checking and Software Verification*. 2000, pp. 323–330. DOI: `10.1007/10722468\_19`.

[Dru03] Doron Drusinsky. “Monitoring Temporal Rules Combined with Time Series”. In: *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*. Ed. by Warren A. Hunt Jr. and Fabio Somenzi. Vol. 2725. Lecture Notes in Computer Science. Springer, 2003, pp. 114–117. DOI: `10.1007/978-3-540-45069-6\_11`.

[Eis+03] Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout. “Reasoning with Temporal Logic on Truncated Paths”. In: *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*. Ed. by Warren A. Hunt Jr. and Fabio Somenzi. Vol. 2725. Lecture Notes in Computer Science. Springer, 2003, pp. 27–39. DOI: `10.1007/978-3-540-45069-6\_3`.

[FP06] Georgios E Fainekos and George J Pappas. “Robustness of temporal logic specifications”. In: *Formal Approaches to Software Testing and Runtime Verification*. Springer, 2006, pp. 178–192.

[FP09] Georgios E Fainekos and George J Pappas. “Robustness of temporal logic specifications for continuous-time signals”. In: *Theoretical Computer Science* 410.42 (2009), pp. 4262–4291.

[Fay19] Peter Faymonville. “Monitoring with Parameters”. PhD Thesis. Saarland University, 2019.

[Fay+16] Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Hazem Torfah. “A Stream-Based Specification Language for Network Monitoring”. In: *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings*. Ed. by Yliès Falcone and César Sánchez. Vol. 10012. Lecture Notes in Computer Science. Springer, 2016, pp. 152–168. DOI: `10.1007/978-3-319-46982-9\_10`.

[Fay+19a] Peter Faymonville, Bernd Finkbeiner, Malte Schledjewski, Maximilian Schwenger, Marvin Stenger, Leander Tentrup, and Hazem Torfah. “StreamLAB: Stream-based Monitoring of Cyber-Physical Systems”. In: *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*. Ed. by Isil Dillig and Serdar Tasiran. Vol. 11561. Lecture Notes in Computer Science. Springer, 2019, pp. 421–431. DOI: `10.1007/978-3-030-25540-4\_24`.

[Fay+19b] Peter Faymonville, Bernd Finkbeiner, Malte Schledjewski, Maximilian Schwenger, Leander Tentrup, and Hazem Torfah. *Real-time Stream Monitoring with StreamLAB*. 2019. URL: `https://www.react.uni-saarland.de/publications/FFS+19a.pdf`.

[Fay+17] 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`.

[Fin+21] Bernd Finkbeiner, Andreas Keller, Jessica Schmidt, and Maximilian Schwenger. "Robust Monitoring for Medical Cyber-Physical Systems". In: *Proceedings of the Workshop on Medical Cyber Physical Systems and Internet of Medical Things*. Mcps '21. Nashville, Tennessee: Association for Computing Machinery, 2021, pp. 17–22. ISBN: 9781450383271. DOI: `10.1145/3446913.3460318`.

[FK09] Bernd Finkbeiner and Lars Kuhtz. "Monitor Circuits for LTL with Bounded and Unbounded Future". In: *Runtime Verification, 9th International Workshop, RV 2009, Grenoble, France, June 26-28, 2009. Selected Papers*. 2009, pp. 60–75. DOI: `10.1007/978-3-642-04694-0\_5`.

[Fin+20] Bernd Finkbeiner, Stefan Oswald, Noemi Passing, and Maximilian Schwenger. "Verified Rust Monitors for Lola Specifications". In: *Runtime Verification - 20th International Conference, RV 2020, Los Angeles, CA, USA, October 6-9, 2020, Proceedings*. Ed. by Jyotirmoy Deshmukh and Dejan Nickovic. Vol. 12399. Lecture Notes in Computer Science. Springer, 2020, pp. 431–450. DOI: `10.1007/978-3-030-60508-7\_24`.

[FSS20] Bernd Finkbeiner, Jessica Schmidt, and Maximilian Schwenger. *Simplex Architecture Meets RTLola*. 2020. URL: `https://www.react.uni-saarland.de/publications/FSS20.pdf`.

[FS04] 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`.

[For04] Bryan Ford. "Parsing expression grammars: a recognition-based syntactic foundation". In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*. Ed. by Neil D. Jones and Xavier Leroy. Acm, 2004, pp. 111–122. DOI: `10.1145/964001.964011`.

[Ful+15] Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völp, and André Platzer. "KeYmaera X: An Axiomatic Tactical Theorem Prover for Hybrid Systems". In: *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*. Ed. by Amy P. Felty and Aart Middeldorp. Vol. 9195. Lecture Notes in Computer

Science. Springer, 2015, pp. 527–538. DOI: 10.1007/978-3-319-21401-6_36.

[GF64] Bernard A. Galler and Michael J. Fischer. "An improved equivalence algorithm". In: *Commun. ACM* 7.5 (1964), pp. 301–303. DOI: 10.1145/364099.364331.

[Gam+95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Design Patterns. *Elements of reusable object-oriented software*. Vol. 99. Addison-Wesley Reading, Massachusetts, 1995.

[GS18] Felipe Gorostiaga and César Sánchez. "Striver: Stream Runtime Verification for Real-Time Event-Streams". In: *Rv 2018*. Vol. 11237. Lncs. Springer, 2018, pp. 282–298. DOI: 10.1007/978-3-030-03769-7_16.

[GS21] Felipe Gorostiaga and César Sánchez. "HLola: a Very Functional Tool for Extensible Stream Runtime Verification". In: *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II*. Ed. by Jan Friso Groote and Kim Guldstrand Larsen. Vol. 12652. Lecture Notes in Computer Science. Springer, 2021, pp. 349–356. DOI: 10.1007/978-3-030-72013-1_18.

[GJL04] Olga Grinchtein, Bengt Jonsson, and Martin Leucker. "Learning of Event-Recording Automata". In: *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, Joint International Conferences on Formal Modelling and Analysis of Timed Systems, FORMATS 2004 and Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2004, Grenoble, France, September 22-24, 2004, Proceedings*. Ed. by Yassine Lakhnech and Sergio Yovine. Vol. 3253. Lecture Notes in Computer Science. Springer, 2004, pp. 379–396. DOI: 10.1007/978-3-540-30206-3_26.

[Hal+91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. "The synchronous dataflow programming language LUSTRE". In: *Proceedings of the IEEE*. 1991, pp. 1305–1320.

[Hal05] Nicolas Halbwachs. "A synchronous language at work: the story of Lustre". In: *3rd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2005), 11-14 July 2005, Verona, Italy, Proceedings*. 2005, pp. 3–11. DOI: 10.1109/memcod.2005.1487884.

[HJ08] Kevin W Hamlen and Micah Jones. "Aspect-oriented in-lined reference monitors". In: *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*. 2008, pp. 11–20.

[HG05] Klaus Havelund and Allen Goldberg. “Verify Your Runs”. In: *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*. Ed. by Bertrand Meyer and Jim Woodcock. Vol. 4171. Lecture Notes in Computer Science. Springer, 2005, pp. 374–383. DOI: 10.1007/978-3-540-69149-5_40.

[HR02] Klaus Havelund and Grigore Rosu. “Synthesizing Monitors for Safety Properties”. In: *Tacas 2002*. 2002, pp. 342–356. DOI: 10.1007/3-540-46002-0_24.

[HV08] Klaus Havelund and Eric Van Wyk. “Aspect-oriented monitoring of C programs”. In: (2008).

[Hen96] Thomas A. Henzinger. “The Theory of Hybrid Automata”. In: *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*. IEEE Computer Society, 1996, pp. 278–292. DOI: 10.1109/lics.1996.561342.

[Her+05] Manuel V. Hermenegildo, Elvira Albert, Pedro López-García, and Germán Puebla. “Abstraction-Carrying Code and Resource-Awareness”. In: *Ppdp 2005*. Acm, 2005, pp. 1–11. DOI: 10.1145/1069774.1069775.

[Hin69] Roger Hindley. “The principal type-scheme of an object in combinatory logic”. In: *Transactions of the american mathematical society* 146 (1969), pp. 29–60.

[Hoa03] C. A. R. Hoare. “The verifying compiler: A grand challenge for computing research”. In: *J. Acm* 50.1 (2003), pp. 63–69. DOI: 10.1145/602382.602403.

[Iec06] Iec. *Nuclear Power Plants*. Iec 60880. Geneva, Switzerland: International Electrotechnical Commission, 2006.

[Iec10] Iec. *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems*. Iec 61508. Geneva, Switzerland: International Electrotechnical Commission, 2010.

[Iec12] Iec. *Medical device software — Software life cycle processes*. Iec 62304. Geneva, Switzerland: International Electrotechnical Commission, 2012.

[05] “IEEE Standard for Property Specification Language (PSL)”. In: *IEEE Std 1850-2005* (2005), pp. 1–143. DOI: 10.1109/ieeestd.2005.97780.

[Iso18] Iso. *Road Vehicles – Functional Safety*. Iso 26262 – 1:2018. Geneva, Switzerland: International Organization for Standardization, 2018.

[II10] Iso and Iec. *Programming languages – C*. ISO/IEC Committee Draft 9899:201x. Accessed: 02.02.2022. Geneva, Switzerland: International Organization for Standardization / International Electrotechnical Commission, 2010. URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1539.pdf.

[Jak+15] Stefan Jaksic, Ezio Bartocci, Radu Grosu, Reinhard Kloibhofer, Thang Nguyen, and Dejan Nickovic. "From signal temporal logic to FPGA monitors". In: *Memocode 2015*. 2015, pp. 218–227. DOI: 10.1109/memcod.2015.7340489.

[Jak+18] Stefan Jakšić, Ezio Bartocci, Radu Grosu, Thang Nguyen, and Dejan Ničković. "Quantitative monitoring of STL with edit distance". In: *Formal methods in system design* 53.1 (2018), pp. 83–112.

[Kah62] Arthur B. Kahn. "Topological sorting of large networks". In: *Commun. ACM* 5.11 (1962), pp. 558–562. DOI: 10.1145/368996.369025.

[Kic+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. "Aspect-Oriented Programming". In: *ECOOP'97 - Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 9-13, 1997, Proceedings*. Ed. by Mehmet Aksit and Satoshi Matsuoka. Vol. 1241. Lecture Notes in Computer Science. Springer, 1997, pp. 220–242. DOI: 10.1007/BFb0053381.

[Kim+99] Moonjoo Kim, Mahesh Viswanathan, Hanêne Ben-Abdallah, Sampath Kannan, Insup Lee, and Oleg Sokolsky. "Formally specified monitoring of temporal properties". In: *11th Euromicro Conference on Real-Time Systems (ECRTS 1999), 9-11 June 1999, York, England, UK, Proceedings*. IEEE Computer Society, 1999, pp. 114–122. DOI: 10.1109/emrts.1999.777457.

[LBG18] Imane Lamrani, Ayan Banerjee, and Sandeep K. S. Gupta. "HyMn: Mining linear hybrid automata from input output traces of cyber-physical systems". In: *IEEE Industrial Cyber-Physical Systems, ICPS 2018, Saint Petersburg, Russia, May 15-18, 2018*. Ieee, 2018, pp. 264–269. DOI: 10.1109/icphys.2018.8387670.

[Lee+17] Gilwoo Lee, Zita Marinho, Aaron M. Johnson, Geoffrey J. Gordon, Siddhartha S. Srinivasa, and Matthew T. Mason. "Unsupervised Learning for Nonlinear PieceWise Smooth Hybrid Systems". In: *CoRR* abs/1710.00440 (2017). arXiv: 1710.00440. URL: http://arxiv.org/abs/1710.00440.

[Lee+99] Insup Lee, Sampath Kannan, Moonjoo Kim, Oleg Sokolsky, and Mahesh Viswanathan. "Runtime Assurance Based On Formal Specifications". In: *Pdpta 1999*. 1999, pp. 279–287.

[Leu+18] Martin Leucker, César Sánchez, Torben Scheffel, Malte Schmitz, and Alexander Schramm. "TeSSLa: runtime verification of non-synchronized real-time streams". In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. 2018, pp. 1925–1933.

[LS09] Martin Leucker and Christian Schallhart. "A brief account of runtime verification". In: *J. Log. Algebraic Methods Program.* 78.5 (2009), pp. 293–303. DOI: 10.1016/j.jlap.2008.08.004.

[Li+05] 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 Rec.* 34.1 (2005), pp. 39–44. DOI: `10.1145/1058150.1058158`.

[LC96] Yamin Li and Wanming Chu. "A New Non-Restoring Square Root Algorithm and its VLSI Implementation". In: *1996 International Conference on Computer Design (ICCD '96), VLSI in Computers and Processors, October 7-9, 1996, Austin, TX, USA, Proceedings*. 1996, pp. 538–544. DOI: `10.1109/iccd.1996.563604`.

[Li+10] Zhaopeng Li, Zhong Zhuang, Yiyun Chen, Simin Yang, Zhenting Zhang, and Dawei Fan. "A certifying compiler for Clike subset of C language". In: *2010 4th IEEE International Symposium on Theoretical Aspects of Software Engineering*. Ieee. 2010, pp. 47–56.

[LF07] Hong Lu and Alessandro Forin. *The Design and Implementation of P2V, An Architecture for Zero-Overhead Online Verification of Software Programs*. Tech. rep. Msr-tr-2007-99. Aug. 2007, p. 12. URL: `https://www.microsoft.com/en-us/research/publication/the-design-and-implementation-of-p2v-an-architecture-for-zero-overhead-online-verification-of-software-programs/`.

[MN04] Oded Maler and Dejan Nickovic. "Monitoring Temporal Properties of Continuous Signals". In: *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, Joint International Conferences on Formal Modelling and Analysis of Timed Systems, FORMATS 2004 and Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2004, Grenoble, France, September 22-24, 2004, Proceedings*. Ed. by Yassine Lakhnech and Sergio Yovine. Vol. 3253. Lecture Notes in Computer Science. Springer, 2004, pp. 152–166. DOI: `10.1007/978-3-540-30206-3\_12`.

[Mal17] Marcel Maltry. "FPGA-based Monitoring for Stream Specification Languages". MA thesis. Saarland University, 2017.

[Mas+20] Corto Mascle, Daniel Neider, Maximilian Schwenger, Paulo Tabuada, Alexander Weinert, and Martin Zimmermann. "From LTL to rLTL Monitoring: Improved Monitorability through Robust Semantics". In: *Hscc 2020*. Acm, 2020, 7:1–7:12. DOI: `10.1145/3365365.3382197`.

[Mat70] Yuri V. Matijasevic. "Enumerable Sets are Diophantine". In: *Soviet Math. Dokl.* 11 (1970), pp. 354–358. URL: `https://ci.nii.ac.jp/naid/10009422455/en/`.

[Med+15] Ramy Medhat, S. Ramesh, Borzoo Bonakdarpour, and Sebastian Fischmeister. "A framework for mining hybrid automata from input/output traces". In: *2015 International Conference on Embedded Software, EMSOFT 2015, Amster-*

dam, Netherlands, October 4-9, 2015. 2015, pp. 177–186. DOI: 10.1109/emsoft.2015.7318273.

[Mee86] Lambert Meertens. “Algorithmics : towards programming as a mathematical activity”. In: *Towards programming as a mathematical activity. Mathematics and computer science*. Jan. 1986, pp. 289–334.

[Men27] Karl Menger. “Zur allgemeinen Kurventheorie”. ger. In: *Fundamenta Mathematicae* 10.1 (1927), pp. 96–115. URL: http://eudml.org/doc/211191.

[MSS21] Niklas Metzger, Sanny Schmitt, and Maximilian Schwenger. “Conservative Hybrid Automata from Development Artifacts”. In: *CoRR* abs/2111.05613 (2021). arXiv: 2111.05613. URL: https://arxiv.org/abs/2111.05613.

[Mil78] Robin Milner. “A Theory of Type Polymorphism in Programming”. In: *J. Comput. Syst. Sci.* 17.3 (1978), pp. 348–375. DOI: 10.1016/0022-0000(78)90014-4.

[MP16] Stefan Mitsch and André Platzer. “ModelPlex: verified runtime validation of verified cyber-physical system models”. In: *Formal Methods Syst. Des.* 49.1-2 (2016), pp. 33–74. DOI: 10.1007/s10703-016-0241-z.

[MRS17] Patrick Moosbrugger, Kristin Y. Rozier, and Johann Schumann. “R2U2: Monitoring and Diagnosis of Security Threats for Unmanned Aerial Systems”. In: *Formal Methods Syst. Des.* 51.1 (2017), pp. 31–61. DOI: 10.1007/s10703-017-0275-x.

[MDM16] Nick Moss, Kei Davis, and Patrick McCormick. “The ARES high-level intermediate representation”. In: *2016 Third Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. Ieee. 2016, pp. 32–39.

[MB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *Tacas 2008*. Vol. 4963. Lncs. Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[MSS16] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. “Viper: A Verification Infrastructure for Permission-Based Reasoning”. In: *Vmcai 2016*. Vol. 9583. Lncs. Springer, 2016, pp. 41–62. DOI: 10.1007/978-3-662-49122-5_2.

[Nas20] Nasa. *Software Assurance and Software Safety Standard*. Standard. NASA, 2020.

[Nec02] George C Necula. “Proof-carrying code. design and implementation”. In: *Proof and system-reliability*. Springer, 2002, pp. 261–288.

[Nec97] George C. Necula. “Proof-Carrying Code”. In: *Popl 1997*. ACM Press, 1997, pp. 106–119. DOI: 10.1145/263699.263712.

[NL98a] George C. Necula and Peter Lee. “Safe, Untrusted Agents Using Proof-Carrying Code”. In: *Mobile Agents and Security*. Ed. by Giovanni Vigna. Vol. 1419. Lecture Notes in Computer Science. Springer, 1998, pp. 61–91. DOI: `10.1007/3-540-68671-1\_5`.

[NL98b] George C. Necula and Peter Lee. “The Design and Implementation of a Certifying Compiler”. In: *Pldi 1998*. Acm, 1998, pp. 333–344. DOI: `10.1145/277650.277752`.

[NA18] S. Newton and Nathan Aschbacher. “The Challenge of Using C in Safety-Critical Applications”. In: 2018.

[Nig+12] Oliver Niggemann, Benno Stein, Asmir Vodencarevic, Alexander Maier, and Hans Kleine Büning. “Learning Behavior Models for Hybrid Timed Systems”. In: *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada*. 2012. URL: `http://www.aaai.org/ocs/index.php/AAAI/AAAI12/paper/view/4993`.

[Osw20] Stefan Oswald. “Verifiable Runtime Monitor Generation for Lola Specifications”. Bachelor Thesis. Saarland University, 2020.

[Pel+08] Rodolfo Pellizzoni, Patrick O’Neil Meredith, Marco Caccamo, and Grigore Rosu. “Hardware Runtime Monitoring for Dependable COTS-Based Real-Time Embedded Systems”. In: *Proceedings of the 29th IEEE Real-Time Systems Symposium, RTSS 2008, Barcelona, Spain, 30 November - 3 December 2008*. 2008, pp. 481–491. DOI: `10.1109/rtss.2008.43`.

[Pik+10] Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. “Copilot: A Hard Real-Time Runtime Monitor”. In: *Rv 2010*. Vol. 6418. Lncs. Springer, 2010, pp. 345–359. DOI: `10.1007/978-3-642-16612-9\_26`.

[Pla08] André Platzer. “Differential Dynamic Logic for Hybrid Systems”. In: *J. Autom. Reasoning* 41.2 (2008), pp. 143–189. DOI: `10.1007/s10817-008-9103-8`.

[Pnu77] Amir Pnueli. “The Temporal Logic of Programs”. In: *Annual Symposium on Foundations of Computer Science, 1977*. IEEE Computer Society, 1977, pp. 46–57. DOI: `10.1109/sfcs.1977.32`.

[Ril18] Charles Riley. *Volkswagen’s diesel scandal costs hit $30 billion*. CNN Business. Online; accessed: 2020-10-15. 2018. URL: `https://money.cnn.com/2017/09/29/investing/volkswagen-diesel-cost-30-billion/index.html`.

[RH05] Grigore Rosu and Klaus Havelund. “Rewriting-Based Techniques for Runtime Verification”. In: *Autom. Softw. Eng.* 12.2 (2005), pp. 151–197. DOI: `10.1007/s10515-005-6205-y`.

[Rtc11] Rtca. *Software Considerations in Airborne Systems and Equipment Certification*. Do Do-178c. International Organization for Standardization, 2011.

[San+15] Pedro Henrique Santana, Spencer Lane, Eric Timmons, Brian Charles Williams, and Carlos Forster. "Learning Hybrid Models with Guarded Transitions". In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*. 2015, pp. 1847–1853. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9480.

[STA18] Sebastian Schirmer, Christoph Torens, and Florian Adolf. "Formal Monitoring of Risk-based Geofences". In: *2018 AIAA Information Systems-AIAA Infotech Aerospace*. 2018. DOI: 10.2514/6.2018-1986. eprint: https://arc.aiaa.org/doi/pdf/10.2514/6.2018-1986.

[Sch+19] Joshua Schneider, David A. Basin, Srdan Krstic, and Dmitriy Traytel. "A Formally Verified Monitor for Metric First-Order Temporal Logic". In: *Rv 2019*. Ed. by Bernd Finkbeiner and Leonardo Mariani. Vol. 11757. Lncs. Springer, 2019, pp. 310–328. DOI: 10.1007/978-3-030-32079-9_18.

[Sch71] Victor Schneider. "On the number of registers needed to evaluate arithmetic expressions". In: *BIT Numerical Mathematics* 11.1 (1971), pp. 84–93.

[Sch19a] Maximilian Schwenger. "Let's not Trust Experience Blindly: Formal Monitoring of Humans and other CPS". Master Thesis. Saarland University, 2019.

[Sch19b] Maximilian Schwenger. *RustTyC*. https://github.com/Schwenger/RustTyC/. Accessed: 06.01.2022. 2019.

[Sch20] Maximilian Schwenger. "Monitoring Cyber-Physical Systems: From Design to Integration". In: *Runtime Verification - 20th International Conference, RV 2020, Los Angeles, CA, USA, October 6-9, 2020, Proceedings*. Ed. by Jyotirmoy Deshmukh and Dejan Nickovic. Vol. 12399. Lecture Notes in Computer Science. Springer, 2020, pp. 87–106. DOI: 10.1007/978-3-030-60508-7_5.

[SR03] Koushik Sen and Grigore Rosu. "Generating Optimal Monitors for Extended Regular Expressions". In: *Electron. Notes Theor. Comput. Sci.* 89.2 (2003), pp. 226–245. DOI: 10.1016/s1571-0661(04)81051-x.

[Sot+19] Miriam García Soto, Thomas A. Henzinger, Christian Schilling, and Luka Zeleznik. "Membership-Based Synthesis of Linear Hybrid Automata". In: *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*. Ed. by Isil Dillig and Serdar Tasiran. Vol. 11561. Lecture Notes in Computer Science. Springer, 2019, pp. 297–314. DOI: 10.1007/978-3-030-25540-4_16.

[SOM17] Adam Summerville, Joseph C. Osborn, and Michael Mateas. "CHARDA: Causal Hybrid Automata Recovery via Dynamic Analysis". In: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*. Ed. by Carles Sierra. ijcai.org, 2017, pp. 2800–2806. DOI: 10.24963/ijcai.2017/390.

[Tap+19] Martin Tappler, Bernhard K. Aichernig, Kim Guldstrand Larsen, and Florian Lorber. "Time to Learn - Learning Timed Automata from Tests". In: *Formal Modeling and Analysis of Timed Systems - 17th International Conference, FORMATS 2019, Amsterdam, The Netherlands, August 27-29, 2019, Proceedings*. Ed. by Étienne André and Mariëlle Stoelinga. Vol. 11750. Lecture Notes in Computer Science. Springer, 2019, pp. 216–235. DOI: `10.1007/978-3-030-29662-9\_13`.

[Tea10] Crates.io Team. *Crates.op*. `https://crates.io`. Accessed: 06.01.2022. 2010.

[TR05] Prasanna Thati and Grigore Roşu. "Monitoring algorithms for metric temporal logic specifications". In: *Electronic Notes in Theoretical Computer Science* 113 (2005), pp. 145–162.

[The98] The European Parliament and the Council of the European Union. "Directive 98/69/EC of the European Parliament and of the Council". In: *Official Journal of the European Communities* (1998). URL: `http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:31998L0069:EN:HTML`.

[The17] The European Parliament and the Council of the European Union. *Commission Regulation (EU) 2017/1151*. June 2017. URL: `http://data.europa.eu/eli/reg/2017/1151/oj` (visited on 10/15/2020).

[Tis18] Dragoş Tiselice. *pest. The Elegant Parser*. `https://pest.rs`. Accessed: 06.01.2022. 2018.

[Tor+17] Christoph Torens, Florian Adolf, Peter Faymonville, and Sebastian Schirmer. "Towards Intelligent System Health Management using Runtime Monitoring". In: *AIAA Information Systems-AIAA Infotech Aerospace*. American Institute of Aeronautics and Astronautics (AIAA), Jan. 2017. DOI: `10.2514/6.2017-0419`.

[Tut+15] Monica Tutuianu, Pierre Bonnel, Biagio Ciuffo, Takahiro Haniu, Noriyuki Ichikawa, Alessandro Marotta, Jelica Pavlovic, and Heinz Steven. "Development of the World-wide harmonized Light duty Test Cycle (WLTC) and a possible pathway for its introduction in the European legislation". In: *Transportation Research Part D: Transport and Environment* 40.Supplement C (2015), pp. 61–75. ISSN: 1361-9209. DOI: `10.1016/j.trd.2015.07.011`.

[Ulu17] Dogan Ulus. "Montre: A Tool for Monitoring Timed Regular Expressions". In: *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*. Ed. by Rupak Majumdar and Viktor Kuncak. Vol. 10426. Lecture Notes in Computer Science. Springer, 2017, pp. 329–335. DOI: `10.1007/978-3-319-63387-9\_16`.

[WW62] Author(s) B. P. Welford and B. P. Welford. “Note on a method for calculating corrected sums of squares and products”. In: *Technometrics* (1962), pp. 419–420.

[Wic+95] Brian A. Wichmann, A. A. Canning, D. L. Clutterbuck, L. A. Winsborrow, N. J. Ward, and D. William R. Marsh. “Industrial perspective on static analysis”. In: *Softw. Eng. J.* 10.2 (1995), pp. 69–75. DOI: `10.1049/sej.1995.0010`.

[WS20] Thomas Wright and Ian Stark. “Property-Directed Verified Monitoring of Signal Temporal Logic”. In: *Runtime Verification - 20th International Conference, RV 2020, Los Angeles, CA, USA, October 6-9, 2020, Proceedings*. Ed. by Jyotirmoy Deshmukh and Dejan Nickovic. Vol. 12399. Lecture Notes in Computer Science. Springer, 2020, pp. 339–358. DOI: `10.1007/978-3-030-60508-7\_19`.

[ZJP21] Bingzhuo Zhong, Claudius Jordan, and Julien Provost. “Extending Signal Temporal Logic with Quantitative Semantics by Intervals for Robust Monitoring of Cyber-physical Systems”. In: *ACM Transactions on Cyber-Physical Systems* 5.2 (2021), pp. 1–25.

Appendix A

Appendix

The appendix consists of three parts. First, Appendix A.1 contains preliminary information, introducing notation, standard mathematical concepts, and background work on sliding window aggregations. Second, Appendix A.2 constitutes supplementary material to Section 3.2. It contains and explains the full output of the compilation of a Lola specification. Last, Appendix A.3 concerns Chapter 4, supplementing the traces mentioned in the evaluation of the automaton construction.

→Sec. 3.2, p. 128
→Chap. 4, p. 151

A.1. Preliminaries

This section introduces some common mathematical concepts, clarifies notation, and introduces RTLola-specific background information regarding the computation of sliding window aggregations.

A.1.1. Sequences and Partitions

Definition A.1 (Sequence Notation)

Let $f\colon \mathcal{I} \to \mathcal{J}$ be a function for some sets $\mathcal{I}$ and $\mathcal{J}$. Further, let $<$ be a total order for $\mathcal{I}$. Then, $\langle f(i)\rangle_{i\in I}$ is a *family*, i.e., it generates a sequence where each element is the result of applying f to elements of I in their order with respect to $<$.

Def. Family

$$\langle f(i)\rangle_{i\in\emptyset} = \varepsilon$$

$$\langle f(i)\rangle_{i\in I} = f(\min I) \circ \langle f(i)\rangle_{i\in I\setminus\{\min I\}} \qquad \text{(for } I \neq \emptyset\text{)}$$

Here, $\circ$ is the sequence concatenation operator and ε denotes the empty sequence.

For a non-empty sequence τ, the notation $\tau.\mathrm{last}$ accesses the last element. If τ has at least n elements, $\tau[n]$ accesses the nth element.

Def. Sequence Access

An arrow over a variable, such as $\vec{x}$, explicitly marks it as sequence or vector to avoid ambiguities.

The notation for families is quite liberal; both the function and the set can be stated in any way that provides an unambiguous result. Moreover, with a slight abuse of notation, if there is no order on the elements of the set, a family becomes an unordered set rather than a sequence.

Example A.2 (Sequences)**.**

$$\langle 3i \rangle_{i \in \{1,2,4,8\}} = (3, 6, 12, 24)$$

$$\langle 3i \rangle_{i \in \{1,2,4,8\}} . \mathrm{last} = 24$$

$$\langle x < 5 \rangle_{x \leqslant 20} = (1, 2, 3, 4)$$

$$\langle x \rangle_{x \in \{3, \mathit{true}, \mathit{Never}\}} = \{\mathit{Never}, \mathit{true}, 3\}$$

△

Definition A.3 (Partition)

Def. Partition

Let S be a non-empty set. The set of sets $\mathcal{P} = \{\zeta_1, \ldots, \zeta_k\}$ is a *partition* of S iff (1) $\forall \zeta \in \mathcal{P} : \emptyset \neq \zeta \subseteq S$, (2) $\bigcup_{\zeta \in \mathcal{P}} \zeta = \mathcal{P}$, and (3) $\forall \zeta_i, \zeta_j \in \mathcal{P} : i \neq j \implies \zeta_i \cap \zeta_j = \emptyset$. Elements of $\mathcal{P}$ are *equivalence classes*. $[\![\zeta]\!] \in \zeta$ denotes a unique but arbitrary representative of ζ.

Definition A.4 (Ordered (Interval-) Partition)

Def. Ordered Interval Partition

Let $\mathcal{I}$ be a closed interval. A sequence $I_1, \ldots, I_k$ for some $k \in \mathbb{N}$ is an *ordered interval partition* of $\mathcal{I}$ iff $\{I_i \mid i \leqslant k\}$ is a partition of $\mathcal{I}$ and its constituents are consecutive, i.e.:

$$\forall i < k \colon |I_i| = 0 \vee (I_i = [u, \ell] \wedge I_{i+1} = [u', \ell'] \implies u + 1 = l')$$

Intuitively, an ordered interval partition splits a series of numbers into parts such that the concatenation of the parts yields the original series.

A.1.2. Miscellaneous

Alpha-Renaming

Alpha-renaming is a replacement operation on expressions.

Definition A.5 (Alpha-Renaming)

Def. Alpha-Renaming

Let e be an expression and α a replacement, i.e., a sequence $\langle e_i \mapsto s_i \rangle_{i \leqslant n}$ for some $n \in \mathbb{N}$. The *alpha-renaming* $e\,[\alpha]$ is an expression where each occurrence of any (sub-) expression e_i in e is replaced by s_i.

The replacement itself is mostly agnostic of the language of the expressions, however, it introduces parentheses if necessary and for the purpose of better illustration, it applies some operator simplifications.

Example A.6 (Alpha-Renaming)**.**

$$a + bc\,[a \mapsto 3, b \mapsto x + y] = 3 + (x + y)c$$

`x.offset(by: -q)` $[x \mapsto \textit{in}, q \mapsto -3] =$ `in.offset(by: 3)`

△

Boolean Formulas

Definition A.7 (Positive Boolean Formula)

A *positive boolean formula* $\Phi \in \mathbb{B}_P^+$ over a set of propositions P is a boolean expression of the following shape with $p \in P$:

Def. Positive Boolean Formula

$$\Phi ::= p \mid \Phi \vee \Phi \mid \Phi \wedge \Phi$$

A.1.3. Sliding Window Aggregations

A sliding window has two parameters: a real-valued size, and an aggregation function. It takes a sequence of timestamped values as inputs and applies the aggregation to it. However, when evaluating the sliding window, it only takes values account which lie less than the size of the window in the past. Of particular interest regarding the aggregation functions are list homomorphisms since they can be evaluated efficiently.

Definition A.8 (List Homomorphisms [Mee86])

Let $\langle I_i \rangle_{i \leqslant k}$ be an ordered interval partition of some set $\mathcal{J} \subseteq \mathbb{N}$.

A function $f \colon T \to T_r$ is a *list homomorphism* if there is a mapping function $\mathrm{map} \colon T \to T'$, an associative binary reduction operation $\oplus \colon T' \times T' \to T'$ with neutral element ε, and a finalization function $\mathrm{fin} \colon T' \to T_r$ with the following property:

Def. List Homomorphism

$$f(\vec{x}) = \mathrm{fin}\left(\oplus \left\langle \oplus \left\langle \mathrm{map}(\vec{x}\,[i]) \right\rangle_{i \in I} \right\rangle_{I \in \mathcal{J}}\right)$$

Note that due to its associativity, $\oplus$ can be applied to a non-empty sequence of values by successively reducing arbitrary pairs of the sequence. For an empty sequence, this yields ε.

Intuitively, a list homomorphism aggregates a sequence of values to produce a single value. The crux is that rather than summarizing the whole sequence at once, a list homomorphism allows for pre-aggregating arbitrary sub-sequences provided the original order is respected. For this, it uses several sub-functions. The mapping function transforms a single value into an intermediate value of type T'. The reduction takes two of the intermediate values and reduces them to one. This operation is robust against "empty" values, i.e., values which represent an empty sub-aggregation, denoted by the neutral element ε. Lastly, after successive aggregation, the finalization function transforms an intermediate value into a result of type T_r.

Definition A.9 ((Efficient) Sliding Window Aggregation)

Let $\vec{v}$ be a timestamped sequence of values of type T with monotonically increasing timestamps, let $t \in \mathbb{R}^+$ be a timestamp greater than the timestamp of $\vec{v}.\text{last}$ and let $\gamma\colon T^* \to T'$ be an aggregation function. Here, both T and T' are finite sets. Further, let $\vec{v} = \vec{x} \circ \vec{y}$ be a split of $\vec{v}$ such that $\vec{y}$ captures all values with a timestamp within $(t - \delta, t]$.

Def. Sliding Window

A *sliding window aggregation* over a time span of $\delta \in \mathbb{R}^+$ is a function $\omega_\gamma\colon T^* \to \mathbb{R}^+ \to T'$ defined as

$$\omega_\gamma(\vec{c})(t) = \gamma(\vec{y})$$

Def. Efficient Sliding Window Aggregation

An implementation of a real-time sliding window aggregation if *efficient* if it receives the sequence element by element and has a space complexity in $\mathcal{O}(1)$.

Note that barely any sliding window aggregation can be computed efficiently. Consider, for example, $\gamma = \sum$. Since the implementation receives inputs successively, it has to pre-aggregate values, which is possible via addition. However, the decision on whether the value a is relevant for the aggregation, i.e., whether $a \in \vec{y}$ is due at the point of reception of a. Since the timestamp of evaluation is unknown, the decision is impossible to make.

This strong limitation is lifted by supplying information on potential points of evaluation. Suppose these timestamps will always be drawn from $\{k\pi \mid k \in \mathbb{N}\}$ for some fixed π. In this case, the implementation can split the real time axis in equidistant frames of with δ and pre-summate all values within one frame. It then only has to store π/δ pre-aggregation results. As soon at it receives a point of evaluation, it can sum up all of these intermediate results to obtain the final result [Li+05].

While this method works well for list homomorphic aggregation functions, it fails for example for the median computation. Its correctness is summarized in Corollary 21 of my earlier work [Sch19a].

Corollary (Static Memory Bounds for Sliding Windows [Sch19a])**.** *For a list homomorphic function γ and a period π, there is an efficient sliding window aggregation provided it is only computed at multiples of π.*

Note that the original statement is specific to RTLola but trivially extends to general sequences of values.

Note further that there are aggregation functions prohibiting efficient implementations such as if γ is the median function. Since the implementation has no information about future inputs, it has to memorize all values and compute the result at the end. As a result, the space complexity exceeds $\mathcal{O}(1)$. However, many practically relevant aggregations are list homomorphisms such as the summation and products, extrema, averages, integration, and (co-)variance [WW62; Sch19a].

A.2. Software Compilation: Full Output

This section supplements the full output of the software compilation presented in Section 3.2.3, including the verification annotations The underlying specification is the one from Example 3.3, i.e., the running example.

A.2.1. Memory and Ghost Memory

The memory and ghost memory are encoded as follows:

```
pub struct Memory {
  pub altitude_0: i64,
  pub altitude_1: i64,
}

impl Memory {
  #[pure]
  #[requires="idx >= 0 && idx < 2"]
  #[ensures="idx == 0 ==> result == self.altitude_0"]
  #[ensures="idx == 1 ==> result == self.altitude_1"]
  pub fn get_alt(&self, idx: usize) -> i64 {
    if idx == 0 {
      self.altitude_0
    } else if idx == 1 {
      self.altitude_1
    } else {
      unreachable!()
    }
  }
  #[ensures="self.get_alt(1) == v"]
  #[ensures="self.altitude_0 == old(self.altitude_1)"]
  pub fn store_alt(&mut self, v: i64) {
    self.altitude_0 = self.altitude_1;
    self.altitude_1 = v;
  }
}
```

The `Memory` struct consists of two integers encoding the last two values of the input stream `altitude`. The functions `get_alt` and `store_alt` serve to access one of the last two values of `altitude` and to extend `altitude` with a new value, respectively. Since `get_alt` retrieves a value of the stream `altitude` from the working memory, the generated annotations are exactly as described in Section 3.2.4: `#[requires="idx >= 0 && idx < 2"]` requires that the index `idx` does not exceed the memory reserved for `altitude`, i.e., that is does not exceed the memory bound $\mu(|altitude|) = 2$. Furthermore, the annotation `#[ensures="idx == i ==> result == self.altitude_0"]` for $0 \leqslant i \leqslant 1$ ensures that the return value of `get_alt` corresponds to the respective value of the inputs stream `altitude` stored in `Memory`. Similarly, `#[ensures="self.get_alt(1)== v"]`, as well as the annotation `#[ensures="self.altitude_0 == old(self.altitude_1)"]` ensure that the effects of the function `store_alt` are also committed in `Memory`.

```
struct GhostMemory_i64 {
  mem: Vec<i64>,
}

impl GhostMemory_i64 {
  #[trusted]
  #[ensures="result.len() == 0"]
  pub fn new() -> Self {
    GhostMemory_i64 { mem: Vec::new() }
  }
  #[trusted]
  #[ensures="self.len() == old(self.len()) + 1"]
  #[ensures="self.get(self.len() -1) == v"]
  #[ensures="forall i: usize :: (0 <= i && i < old(self.len())) ==>
    (old(self.get(i)) == self.get(i))"]
  pub fn store(&mut self, v: i64) {
    self.mem.push(v);
  }
  #[trusted]
  #[pure]
  #[requires="idx >= 0 && idx < self.len()"]
  pub fn get(&self, idx: usize) -> i64 {
    self.mem[idx]
  }
  #[trusted]
  #[pure]
  #[ensures="result >= 0"]
  pub fn len(&self) -> usize {
    self.mem.len()
  }
}
```

The `GhostMemory` struct wraps a vector `mem`, i.e., a dynamically growing sequence of data. The functions `new`, `store`, `get`, and `len` allow for creating a new ghost memory, storing a value in the vector, retrieving it from the ghost memory, and determining the current length of the ghost memory, respectively. All functions utilize vector operations that stem from the `std::vec` standard library. These operations cannot be verified by Prusti, so the annotation `#[trusted]` in front of all functions prompt Prusti to assumed their correctness while treating them as black boxes. The annotation `#[pure]` indicates that all functions solely mutate their own stack portions.

Similar to the annotation of the previously seen function `get_alt`, the annotation `#[requires="idx >= 0 && idx < self.len()"]` in front of the function `get` requires that the index does not exceed the current length of the ghost memory. The functions `new` and `len` are equipped with annotations stating simple facts about them. These facts are essential to verify functions that utilize `new` and `len`: If a new ghost memory is created, it is empty and the current length of the ghost memory is always non-negative. Similarly, the annotations of the function store establish facts about its effects: The length of ghost memory is increased by one if it is extended with a new value, after extending the ghost memory with value v, its last value is indeed v, and, lastly, values stored in the ghost memory are not changed when extending it.

A.2.2. Emission and Retrieval

The generated code defines three functions get_input, exists_input, and emit whose precise implementation is neither relevant for the stream evaluation, nor for the verification of the monitor. They merely serve as the connection to the underlying system.

```
#[trusted]
fn get_input() -> i64 { ... }

#[trusted]
fn exists_input() -> bool { ... }

#[trusted]
fn emit(above: bool, below: bool) { ... }
```

The function get_input reads and then returns the next value of the input stream altitude. The function exists_input determines whether there is a new value of the input stream. Hence, it indicates when the input stream ceases to produce new values. Lastly, emit takes two boolean variables above and below as input. The function notifies the system if a trigger condition is satisfied.

A.2.3. Stream Evaluation

The function main contains the stream evaluation. Thus, it is split into three phases, the *prefix*, the *loop*, and the *postfix*. Separate functions handle the evaluation of each phase.

```
pub fn main(mem: &mut Memory) {
  let mut altitude: i64;
  let mut tooLow: bool;
  let mut tooHigh: bool;
  let mut trigger_below: bool;
  let mut trigger_above: bool;
  let mut gm_alt = GhostMemory_i64::new();

  prefix(mem, &mut gm_alt)

  let mut iter = 3;
  let mut con = exists_input();
  #[invariant="iter >= 3"]
  #[invariant="gm_alt.len() == iter"]
  #[invariant="mem.get_alt(0) == gm_alt.get(iter - 2)"]
  #[invariant="mem.get_alt(1) == gm_alt.get(iter - 1)"]
  #[invariant="trigger_below == ((gm_alt.get(iter - 3) < 200) && ((gm_alt.get(iter -
    2) < 200) && (gm_alt.get(iter - 1) < 200)))"]
    #[invariant="trigger_above == ((gm_alt.get(iter - 3) > 600) && ((gm_alt.get(iter -
      2) > 600) && (gm_alt.get(iter - 1) > 600)))"]
    while (con) {
      altitude = get_input();
      tooLow = ((mem.get_alt(0) < 200) && ((mem.get_alt(1) < 200) && (altitude <
      200)));
      tooHigh =
        ((mem.get_alt(0) > 600) && ((mem.get_alt(1) > 600) && (altitude > 600)));
      trigger_below = tooLow;
      trigger_above = tooHigh;
      emit(
        trigger_above,
        trigger_below,
      );
      mem.store_alt(altitude);
      gm_alt.store(altitude);
      iter += 1;
      con = exists_input();
    }

    postfix(mem, &mut gm_alt)
  }
```

The function main has six variables, one for each stream and each trigger as well as one for the ghost memory. It first calls the prefix function defined later. Recall that the prefix length $\eta_{\Phi}^{\overleftarrow{}}$ of the Lola specification from Example 3.3 is two. Hence, since we start with iteration 0, from iteration 2 onwards all accesses to altitude with offset -1 succeed. The function prefix performs the first *three* iterations, which leads to easier and more concise loop invariants. Hence, when the prefix function returns, iter is set to 3. Variable con captures the condition of the while-loop of the monitor loop phase, i.e. whether there is another monitor input. Its value is the result of the exists_input function, and set before entering the loop as well as at the end of the loop body.

The invariants of the while-loop require that iter is always greater than or equal to 3, the ghost memory gm_alt has an entry for every iteration, the memory mem captures the last two values stored in the ghost memory, and that the variables for the triggers are only set to *true* if the respective condition holds for the values retrieved from the ghost memory. For this recall that — due to the shift of tooLow and tooHigh — the values of the output streams and the trigger at position i is computed in iteration $i + 1$. Hence, in iteration i, their values are determined by checking whether the altitude is too low or too high at the current, the last, and the second to last position.

The body of the while-loop describes the stream evaluation: It reads a new value of the input stream altitude and stores it in the respective variable. Then it computes the values of the variables tooLow and tooHigh according to the definition of the streams based on the values retrieved from the memory mem. Similarly, it computes the values of the variables representing the triggers. It calls the emit function regardless of the trigger values since the function takes care of interpreting the results. Afterwards, the new value of the input stream `altitude` is stored both in the regular memory `mem` and the ghost memory `gm_alt`. This is not necessary for the output streams since no other stream or trigger refers to them with a negative offset. Next, the variable `iter` is incremented, denoting that the current iteration is finished, and the condition `con` is again determined by calling the function `exists_input`.

Next, we present the `prefix` function.

```
fn prefix(
  mem: &mut Memory,
  gm_alt: GhostMemory_i64,
 ) {
  //Local Variables
  let mut altitude: i64;
  let mut tooLow: bool;
  let mut tooHigh: bool;
  let mut trigger_below: bool;
  let mut trigger_above: bool;

  //Iteration 0
  altitude = get_input();
  gm_alt.store(altitude);
  mem.store_alt(altitude);
  //Iteration 1
  let altitude = get_input();
  tooLow = ((0 < 200) && ((mem.get_alt(1) < 200) && (altitude < 200)));
  tooHigh = ((0 > 600) && ((mem.get_alt(1) > 600) && (altitude > 600)));
  trigger_below = tooLow;
  trigger_above = tooHigh;
  emit(
    trigger_above,
    trigger_below,
  );
  gm_alt.store(altitude);
  mem.store_alt(altitude);
  assert!(
    trigger_below
      == ((0 < 200) && ((gm_alt.get(0) < 200) && (gm_alt.get(1) < 200)))
  );
  assert!(
    trigger_above
      == ((0 > 600) && ((gm_alt.get(0) > 600) && (gm_alt.get(1) > 600)))
  );
  //Iteration 2
  let altitude = get_input();
  tooLow = ((mem.get_alt(0) < 200) && ((mem.get_alt(1) < 200) && (altitude < 200)));
  tooHigh = ((mem.get_alt(0) > 600) && ((mem.get_alt(1) > 600) && (altitude > 600)));
  trigger_below = tooLow;
  trigger_above = tooHigh;
  emit(
    trigger_above,
      trigger_below,
    );
    gm_alt.store(altitude);
    mem.store_alt(altitude);
    assert!(
      trigger_below
        == ((gm_alt.get(0) < 200)
          && ((gm_alt.get(1) < 200) && (gm_alt.get(2) < 200)))
    );
    assert!(
      trigger_above
        == ((gm_alt.get(0) > 600)
          && ((gm_alt.get(1) > 600) && (gm_alt.get(2) > 600)))
    );
    let mut con = exists_input();
  }
```

The function `prefix` takes the memory `mem` and the ghost memory `gm_alt` as input. It has a local variable for each stream and trigger. By construction, the prefix is similar to the monitor loop. Yet, instead of using a loop, it spells the first three iterations out explicitly and replaces the failing stream accesses to `altitude` with the default value 0. Due to the shift in the output streams, they are not computed in the very first iteration. The `assert!` statements after the second and the third iteration make the implicit assumptions of the loop invariants of the monitor loop explicit. They are critical to ensure that Viper can verify the last two loop invariants upon loop entry.

Last is the function `postfix`.

```
  fn postfix(
    mem: &mut Memory,
    gm_alt: GhostMemory_i64,
   ) {
    //Local Variables
    let mut altitude: i64;
    let mut tooLow: bool;
    let mut tooHigh: bool;
    let mut trigger_below: bool;
    let mut trigger_above: bool;

    //Iteration N + 1
    tooLow = ((mem.get_alt(0) < 200) && ((mem.get_alt(1) < 200) && (0 < 200)));
    tooHigh = ((mem.get_alt(0) > 600) && ((mem.get_alt(1) > 600) && (0 > 600)));
    trigger_below = tooLow;
    trigger_above = tooHigh;
    emit(
      trigger_above,
      trigger_below,
    );
    assert!(
      trigger_below
        == ((gm_alt.get(iter - 2) < 200)
          && ((gm_alt.get(iter - 1) < 200) && (0 < 200)))
    );
    assert!(
      trigger_above
        == ((gm_alt.get(iter - 2) > 600)
          && ((gm_alt.get(iter - 1) > 600) && (0 > 600)))
    );
  }
```

Yet again, postfix is structurally similar to the prefix function and monitor loop body. The main difference is that it performs the last iteration $N+1$. Here, altitude has ceased to produce new values. Hence, the access to altitude with offset $+1$ fails, so the monitor substitutes the default value 0. Note that due to the shift of tooLow and tooHigh, the access with offset 0 to altitude still succeeds and is correct.

A.2.4. Lola Specifications for Software

The following shows the Lola adaptation of a network monitoring specification.

```
input src, dst, length: Int32
input fin: Bool, push: Bool, syn: Bool
constant server: Int32 := ...

output count : Int32 :=
    if count.last(or: 0) > 201 then 0 else count.last(or: 0) + 1
output receiver : Int32 :=
    if dst=server then receiver.offset(by: -2).defaults(to: 0) + 2
    else if count > 200 then 0
    else receiver[-1,0]
trigger receiver > 50 "Many incoming connections."

output received : Int32 := if dst=server ∧ push then 0 else length
output workload : Int32 := if count > 200 then workload.last(or: 0) +
    1 else 0
trigger workload > 25 "Workload too high."
output opened : Int32 := opened.last(or: 0) + int(dst=server ∧ syn)
output closed : Int32 := closed.last(or: 0) + int(dst=server ∧ fin)
trigger opened - closed < 0
    "Closed more connections than have been opened."
```

Listing A.1: Lola specification for monitoring network traffic.

A.3. Conservative Automata: Input Traces

The following are the adequate traces used to construct the aircraft system depicted in Figure 4.5 with dynamics in solid black.

$$
\begin{aligned}
& (0,0,0) \\
\xrightarrow{\textit{cruise},300}\ & (300,0,300) \\
\xrightarrow{\textit{turnL},5}\ & (750,0,290) \\
\xrightarrow{\textit{LtoS},5}\ & (1200,-750,280) \\
\xrightarrow{\textit{turnL},5}\ & (1650,-750,270) \\
\xrightarrow{\textit{LtoS},5}\ & (2100,-1500,260) \\
\xrightarrow{\textit{turnR},5}\ & (2550,-1500,250) \\
\xrightarrow{\textit{RtoS},5}\ & (3000,-1500,240) \\
\xrightarrow{\textit{descend},5}\ & (3450,-1500,230) \\
\xrightarrow{5}\ & (3450,-1500,150)
\end{aligned}
$$

$$
\begin{aligned}
& (0,0,0) \\
\xrightarrow{\textit{cruise},10}\ & (1000,0,300) \\
\xrightarrow{\textit{turnR},5}\ & (2500,0,310) \\
\xrightarrow{\textit{RtoS},5}\ & (4000,750,320) \\
\xrightarrow{\textit{turnR},5}\ & (5500,750,330) \\
\xrightarrow{\textit{RtoS},5}\ & (7000,1500,340) \\
\xrightarrow{\textit{turnL},5}\ & (8500,1500,350) \\
\xrightarrow{\textit{LtoS},5}\ & (10000,1500,360) \\
\xrightarrow{\textit{descend},5}\ & (11500,1500,370) \\
\xrightarrow{5}\ & (12500,1500,370)
\end{aligned}
$$

$$
\begin{aligned}
& (0,0,0) \\
\xrightarrow{\textit{cruise},20}\ & (1000,0,300) \\
\xrightarrow{\textit{descend},5}\ & (2000,0,300) \\
\xrightarrow{\textit{adjust},5}\ & (2375,0,275) \\
\xrightarrow{\textit{adjust},5}\ & (2750,0,250) \\
\xrightarrow{\textit{adjust},5}\ & (3125,0,225) \\
\xrightarrow{\textit{adjust},5}\ & (3500,0,200) \\
\xrightarrow{\textit{adjust},5}\ & (3875,0,175) \\
\xrightarrow{\textit{adjust},5}\ & (4250,0,150) \\
\xrightarrow{5}\ & (4625,0,125)
\end{aligned}
$$