RML: Runtime Monitoring Language

Luca Franceschini

University of Genoa

Department of Informatics, Bioengineering, Robotics,
and Systems Engineering

Ph.D. in Computer Science and Systems Engineering
Computer Science curriculum

RML: Runtime Monitoring Language

by

Luca Franceschini

March, 2020

Candidate
Luca Franceschini
`luca.franceschini@dibris.unige.it`

Title
RML: Runtime Monitoring Language

Advisor
Davide Ancona
DIBRIS, Università degli Studi di Genova `davide.ancona@unige.it`
External Reviewers
Ferruccio Damiani
Dipartimento di Informatica - Università degli Studi di Torino, Italy

Wolfgang Ahrendt
Chalmers University of Technology Department of Computer Science and Engineering, Gothenburg, Sweden

Adrian Francalanza
Computer Science Faculty of Information & Communications Technology University of Malta

ABSTRACT

Runtime verification is a relatively new software verification technique that aims to prove the correctness of a specific run of a program, rather than statically verify the code. The program is instrumented in order to collect all the relevant information, and the resulting trace of events is inspected by a monitor that verifies its compliance with respect to a specification of the expected properties of the system under scrutiny. Many languages exist that can be used to formally express the expected behavior of a system, with different design choices and degrees of expressivity.

This thesis presents RML, a specification language designed for runtime verification, with the goal of being completely modular and independent from the instrumentation and the kind of system being monitored. RML is highly expressive, and allows one to express complex, parametric, non-context-free properties concisely. RML is compiled down to TC, a lower level calculus, which is fully formalized with a deterministic, rewriting-based semantics.

In order to evaluate the approach, an open source implementation has been developed, and several examples with Node.js programs have been tested. Benchmarks show the ability of the monitors automatically generated from RML specifications to effectively and efficiently verify complex properties.

PUBLICATIONS

Some ideas and figures have appeared previously in the following publications. Other publications that are not relevant to this thesis have been omitted from the list.

Ancona, Dagnino, and Franceschini (2018) and Ancona, Ferrando, Franceschini, et al. (2017) studied how parametric trace expressions can be used to model properties of Java programs such that runtime verification techniques can be used. These papers lead to the development of many features later introduced in RML.

Ancona, Franceschini, Delzanno, et al. (2017) developed and exploited the first prototype of a runtime verification instrumentation for Node.js, which proved the practicality of the modular architecture approach.

Ancona, Franceschini, Ferrando, et al. (2019) defined an alternative version of trace expression semantics and formally proved its determinism; the same semantics is used in this thesis for the deterministic trace calculus (see Chapter 2).

Leotta, Ancona, et al. (2018) applied runtime verification techniques to IoT systems, using a formalism that was a precursor to the ones defined in this work. This approach has later been compared to testing in the same setting (Leotta, Clerissi, et al., 2019).

In an extended abstract (Franceschini, 2019) some of the main contributions of this work have been briefly introduced.

Ancona, Davide, Francesco Dagnino, and Luca Franceschini (2018). "A formalism for specification of Java API interfaces." In: *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops, ISSTA 2018, Amsterdam, Netherlands, July 16-21, 2018*. Ed. by Julian Dolby, William G. J. Halfond, and Ashish Mishra. ACM, pp. 24–26. ISBN: 978-1-4503-5939-9. DOI: 10.1145/3236454.3236476. URL: https://doi.org/10.1145/3236454.3236476.

Ancona, Davide, Angelo Ferrando, Luca Franceschini, and Viviana Mascardi (2017). "Parametric Trace Expressions for Runtime Verification of Java-Like Programs." In: *Proceedings of the 19th Workshop on Formal Techniques for Java-like Programs, Barcelona , Spain, June 20, 2017*. ACM, 10:1–10:6. ISBN: 978-1-4503-5098-3. DOI: 10.1145/3103111.3104037. URL: https://doi.org/10.1145/3103111.3104037.

Ancona, Davide, Luca Franceschini, Giorgio Delzanno, Maurizio Leotta, Marina Ribaudo, and Filippo Ricca (2017). "Towards Runtime Monitoring of Node.js and Its Application to the Internet of Things." In: *Proceedings First Workshop on Architectures, Languages and Paradigms for IoT, ALP4IoT@iFM 2017, Turin, Italy, September 18, 2017*. Ed. by Danilo

Pianini and Guido Salvaneschi. Vol. 264. EPTCS, pp. 27–42. DOI: 10.
4204/EPTCS.264.4. URL: https://doi.org/10.4204/EPTCS.264.4.

Ancona, Davide, Luca Franceschini, Angelo Ferrando, and Viviana Mas-
cardi (2019). "A Deterministic Event Calculus for Effective Runtime
Verification." In: *Proceedings of the 20th Italian Conference on Theoret-
ical Computer Science, ICTCS 2019, Como, Italy, September 9-11, 2019*.
Ed. by Alessandra Cherubini, Nicoletta Sabadini, and Simone Tini.
Vol. 2504. CEUR Workshop Proceedings. CEUR-WS.org, pp. 248–260.
URL: http://ceur-ws.org/Vol-2504/paper28.pdf.

Cherubini, Alessandra, Nicoletta Sabadini, and Simone Tini, eds. (2019).
*Proceedings of the 20th Italian Conference on Theoretical Computer Science,
ICTCS 2019, Como, Italy, September 9-11, 2019*. Vol. 2504. CEUR Work-
shop Proceedings. CEUR-WS.org. URL: http://ceur-ws.org/Vol-
2504.

*Conference Companion of the 3rd International Conference on Art, Science,
and Engineering of Programming, Genova, Italy, April 1-4, 2019* (2019).
ACM. ISBN: 978-1-4503-6257-3. URL: https://dl.acm.org/citation.
cfm?id=3328433.

Damiani, Ernesto, George Spanoudakis, and Leszek A. Maciaszek, eds.
(2019). *Proceedings of the 14th International Conference on Evaluation of
Novel Approaches to Software Engineering, ENASE 2019, Heraklion, Crete,
Greece, May 4-5, 2019*. SciTePress. ISBN: 978-989-758-375-9.

Dolby, Julian, William G. J. Halfond, and Ashish Mishra, eds. (2018).
*Companion Proceedings for the ISSTA/ECOOP 2018 Workshops, ISSTA
2018, Amsterdam, Netherlands, July 16-21, 2018*. ACM. ISBN: 978-1-4503-
5939-9. DOI: 10.1145/3236454. URL: https://doi.org/10.1145/
3236454.

Franceschini, Luca (2019). "RML: runtime monitoring language: a
system-agnostic DSL for runtime verification." In: *Conference Com-
panion of the 3rd International Conference on Art, Science, and Engineer-
ing of Programming, Genova, Italy, April 1-4, 2019*. ACM, 28:1–28:3.
ISBN: 978-1-4503-6257-3. DOI: 10.1145/3328433.3328462. URL: https:
//doi.org/10.1145/3328433.3328462.

Leotta, Maurizio, Davide Ancona, Luca Franceschini, Dario Olianas,
Marina Ribaudo, and Filippo Ricca (2018). "Towards a Runtime Veri-
fication Approach for Internet of Things Systems." In: *Current Trends
in Web Engineering - ICWE 2018 International Workshops, MATWEP, En-
Wot, KD-WEB, WEOD, TourismKG, Cáceres, Spain, June 5, 2018, Revised
Selected Papers*. Ed. by Cesare Pautasso, Fernando Sánchez-Figueroa,
Kari Systä, and Juan Manuel Murillo Rodriguez. Vol. 11153. Lec-
ture Notes in Computer Science. Springer, pp. 83–96. ISBN: 978-3-
030-03055-1. DOI: 10.1007/978-3-030-03056-8_8. URL: https:
//doi.org/10.1007/978-3-030-03056-8%5C_8.

Leotta, Maurizio, Diego Clerissi, Luca Franceschini, Dario Olianas, Da-
vide Ancona, Filippo Ricca, and Marina Ribaudo (2019). "Comparing
Testing and Runtime Verification of IoT Systems: A Preliminary Eval-

uation based on a Case Study." In: *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2019, Heraklion, Crete, Greece, May 4-5, 2019*. Ed. by Ernesto Damiani, George Spanoudakis, and Leszek A. Maciaszek. SciTePress, pp. 434–441. ISBN: 978-989-758-375-9. DOI: 10.5220/0007745604340441. URL: https://doi.org/10.5220/0007745604340441.

Pautasso, Cesare, Fernando Sánchez-Figueroa, Kari Systä, and Juan Manuel Murillo Rodriguez, eds. (2018). *Current Trends in Web Engineering - ICWE 2018 International Workshops, MATWEP, EnWot, KD-WEB, WEOD, TourismKG, Cáceres, Spain, June 5, 2018, Revised Selected Papers*. Vol. 11153. Lecture Notes in Computer Science. Springer. ISBN: 978-3-030-03055-1. DOI: 10.1007/978-3-030-03056-8. URL: https://doi.org/10.1007/978-3-030-03056-8.

Pianini, Danilo and Guido Salvaneschi, eds. (2018). *Proceedings First Workshop on Architectures, Languages and Paradigms for IoT, ALP4IoT@iFM 2017, Turin, Italy, September 18, 2017*. Vol. 264. EPTCS. URL: http://arxiv.org/abs/1802.00976.

Proceedings of the 19th Workshop on Formal Techniques for Java-like Programs, Barcelona , Spain, June 20, 2017 (2017). ACM. ISBN: 978-1-4503-5098-3. DOI: 10.1145/3103111. URL: https://doi.org/10.1145/3103111.

CONTENTS

LIST OF FIGURES

LISTINGS

INTRODUCTION

Ensuring software correctness is one of the crucial high-level goals of programming language theory and theoretical computer science. The task is of the uttermost importance, as software is ubiquitous in our life. This includes low-level electronic devices, smartphones, personal computers and even safety-critical cyber-physical system. The cost of software bugs, depending on the system, can range from negligible to catastrophic, making this research area not only interesting from a purely theoretical perspective, but also very relevant in practice.

Decades of software research on the topic produced a great number of approaches, some more widespread than others in industry, with different specific goals and strengths. The list includes, but is not limited to, static analysis, model checking, type systems, theorem proving, and testing. Different levels of confidence in the software under scrutiny being correct can be achieved, though generally greatest assurance comes with less automatic and more complex tools. Finding a good trade-off between usability and confidence is the final goal for general-purpose techniques.

Runtime verification (Leucker and Schallhart, 2009) is a relatively new approach to software verification, originated from model checking. In static verification, desired properties can quickly become untractable (or even undecidable) because of state explosion, as tools have to consider all possible execution paths by reasoning on the source code. Runtime verification aims to solve a simpler problem, that is, verify correctness (with respect to a given specification) of *one* execution of the program.

Clearly the trade-off is different and constraints are more relaxed: one needs to repeat the verification procedure for every run of the system, and only then errors can be raised. Because of this, runtime verification is well suited to be employed as a complementary technique to other, more standard ones, to be used when it actually is the best tool for the job.

While theoretically simpler, runtime verification introduces a whole new set of problems to be solved. First, an interface to the execution of the program needs to be established, so that software runs are observed and data about them can be collected and analyzed. Then, the verification procedure must be efficient enough to be practically usable on each run. Real-time verification can also be an option, running the analysis alongside of the program: here efficiency is critical, as it is the communication between the two parts. This approach is sometimes called *online* runtime verification, as opposed to *offline*, where the analysis is done after the program terminated.
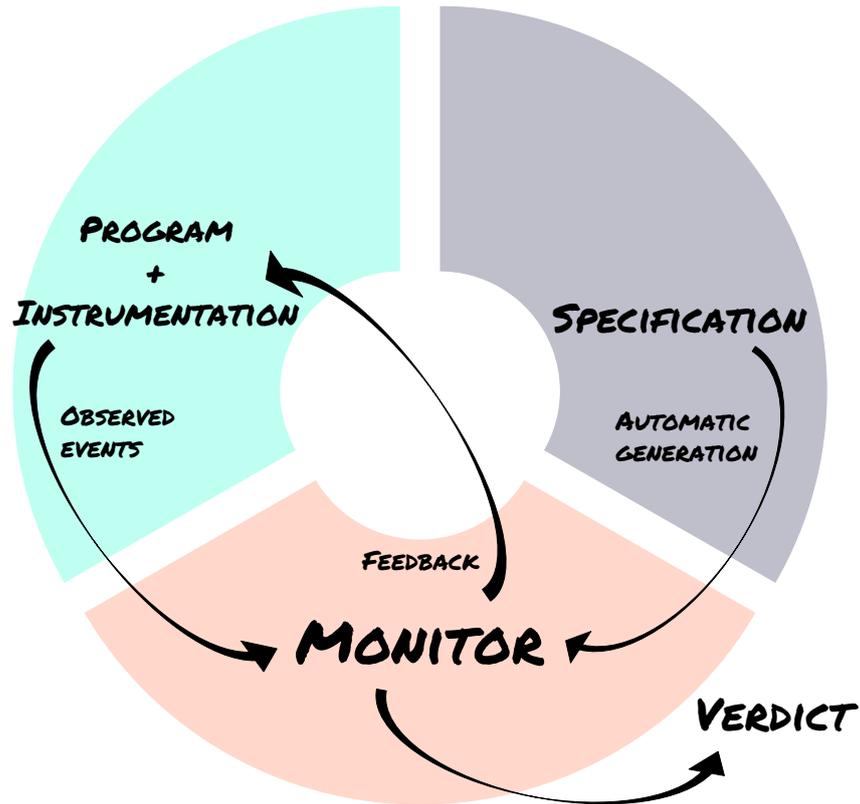
Figure 1.1: Runtime verification modular architecture.

The software layer that lies between the program and the verification procedure is commonly known as *instrumentation*, while the component actually verifying the behavior of the system is the *monitor*. There are many ways to instrument a program, divided in two main categories: *static* instrumentation changes the source code by adding instructions that collect metadata about the execution (possibly without affecting the program semantics); *dynamic* instrumentation instead run as a part of the interpreter or virtual machine executing the program.

Some challenges, on the other hand, are inherited from static verification. The choice of the language used to describe the expected property of the system is of paramount importance, as its expressivity determines what can and what cannot be verified with the system. In this regard, two main approaches exist: either use the programming language of the program under scrutiny to write the verification procedure, thus mixing the two levels, or use a dedicated domain specific language (DSL).

This thesis introduces *Runtime Monitoring Language* (*RML*), a DSL explicitly devised for runtime verification purposes. RML has been inspired by previous work on *trace expressions* (Ancona, Ferrando, and Mascardi, 2016, 2017), another, simpler, runtime verification formalism.

The driving concept behind the design of RML is *modularity*. The architecture of a modular runtime verification system is depicted in Figure 1.1. An instrumented program runs and the relevant events observed are collected and sent to the monitor. The monitor can also reply to the instrumentation giving a feedback on the execution so far. When enough events have been observed to decide whether the desired property holds for the system under test, a verdict is emitted. Finally, it is very desirable that the monitor is automatically generated from the specification, otherwise the amount of work required quickly becomes unbearable.

The philosophy of RML is based on the assumption that these three components are independent and can be completely decoupled, for greater flexibility. This allows RML to be as system-independent as possible, and thus applicable to a wide range of systems, including heterogeneous ones. Clearly an instrumentation system needs to be developed for every programming language/platform that we want to monitor, though this (admittedly not simple) task only needs to be done once.

To give the reader a concrete idea, the following is a simple example of an RML specification for the correct behavior of a file API:

```
open(fd) matches { funcName: 'open', result: fd };
rw(fd) matches { funcName: 'read' | 'write', args: [fd, ...] };
close(fd) matches { funcName: 'close', args: [fd, ...] };

Main = empty \/ { let fd; open(fd) Use };
Use = (rw(fd) Use) \/ close(fd);
```

RML specifications are divided in two parts. First, the kinds of events that we are interested to monitor are defined: these are called *event types*. Then, the real specification builds on top of such abstraction to define the correct behavior at a higher level.

The definition and semantics of RML relies on a core calculus called *Trace Calculus* (*TC*). Part of the contribution of this work is the full formalization of TC, including an implementable *deterministic* rewriting semantics.

Another contribution of the work presented in this thesis is the development of a usable prototype for RML, which includes a compiler translating RML specifications to Prolog, and a Prolog monitor that verifies events against such specifications. Furthermore, as a proof of concept regarding the idea of a modular architecture, an instrumentation has been implemented for Node.js, a JavaScript runtime environment, which has been used to run tests and benchmarks. All the developed source code is open source and hosted on GitHub repositories, and can be found together with introductory material on the RML website[1].

Outline.   The thesis is organized as follows.

---

1 https://rmlatdibris.github.io/

Chapter 2 presents TC, formalized its syntax and semantics, formal properties and results.

Chapter 3 introduces RML, and defines its semantics by a translation to the lower level calculus TC.

Chapter 4 shows how RML can be used in practice and what common verification patterns arose in our experience with it, as well as all the different features that can be combined together to express complex properties.

Chapter 5 provides the implementation details of RML, from its compiler to the monitor encoding the formal semantics.

Chapter 6 assesses the performance of RML with a series of benchmark examples.

Chapter 7 contains an overview of the runtime verification state of the art, showing some of the most common approaches to the task.

Finally, Chapter 8 contains final remarks and discusses future directions of work in this research line.

TRACE CALCULUS

This chapter introduces the *trace calculus TC*, the low level calculus underlying RML, to which the latter is compiled. The formalism originated from another, simpler one, namely *trace expressions* (Ancona, Ferrando, and Mascardi, 2016), still explicitly devised for runtime verification but lacking many of the key features described in this chapter (the most important of which are generics, conditionals, computation capabilities, formalization of event types, and determinism).

The main characteristics of TC are:

EVENTS The semantics is based on a general notion of event that does not depend on any particular kind of system.

EVENT TYPES The basic building block of a specification is an event type, that is, a set of events conveniently represented by a pattern.

EXPRESSIVITY Supported operators are prefix, concatenation, union, intersection and shuffle; furthermore, recursion is supported as well.

PARAMETRICITY Specifications depending on values that will be discovered at runtime are supported.

GENERICS Part of specifications can be made reusable by abstraction over some of their variables, to be instantiated in a different context; these variables can also be used for basic computations with conditional operators.

REGULAR TERMS Recursion is natively supported by allowing specification terms to be regular (Courcelle, 1983): the specification is given through a finite set of syntactic equations.

OPERATIONAL SEMANTICS The implementation of the formalism is based on a rewriting system, and the resulting rewriting semantics can be directly implemented.

INFINITE TRACES The semantics is not limited to finite sequences of events: some specifications allow non-termination.

The rest of this chapter goes through all the concepts above, providing a full formalization and motivating examples.

2.1 EVENTS

Events are observations of the execution of the program under monitoring. They can be, for instance, function and method calls, variable accesses, file system operations, network communications, and virtually any observable behavior of a system.

The formalism of choice for RML events is inspired by JSON (JavaScript Object Notation)[1], as it allows encoding primitive data types, lists and structured objects based on property-key pairs. Field keys are simply identifiers, however, differently from JSON, they are not strings and need not be quoted, as that would make the event syntax more complex and harder to read with no real benefits. Literals include numbers, strings and boolean values; there is no counterpart to the JSON null element.

Events and other syntactic entities will be defined using the convenient and standard Backus-Naur form (BNF). For the sake of readability, we will sometimes abuse BNF notation to describe a syntax where the order of elements is not relevant; if so, it will be explicitly stated.

Definition 1. (Event) Given a set of field keys $\mathcal{K}$, numeric literals $\mathcal{N}$ and string literals $\mathcal{S}$, the set of *event expressions* $\mathcal{E}$ is inductively defined by the following grammar:

$$
\begin{array}{rcll}
\mathcal{E} & ::= & O & \\
  & | & l & \\
  & | & \mathcal{N} & \text{(number)} \\
  & | & \mathcal{S} & \text{(string)} \\
  & | & \mathbb{B} & \text{(boolean)} \\
O & ::= & \{\mathcal{K}\!:\!\mathcal{E}, ..., \mathcal{K}\!:\!\mathcal{E}\} & \text{(event object)} \\
l & ::= & [\mathcal{E}, ..., \mathcal{E}] & \text{(event list)}
\end{array}
$$

The set of *event objects* (or simply *events*) is denoted as O. The order of key-value pairs inside events is not relevant.

The choice of a formalism that allows arbitrarily long, nested, structured data is essential to give the instrumentation component enough flexibility to encode all necessary information in events, as it will be shown in the examples.

A possibly infinite sequence of events is an *event trace*, or simply a trace. Such sequences effectively encode program executions w.r.t. the kind of observations made by the instrumentation layer. In other words, if the instrumentation is able to capture all relevant information, the trace is all we need to analyze a run of a program.

Definition 2. (Trace) Given a set of event objects O, a *trace t* is a possibly infinite sequence of event objects. The set of all finite (infinite) traces

---

over is denoted as O^* (O^ω). The set of all traces over O is O^∞ ($O^\infty = O^* \cup O^\omega$). The empty trace is denoted as ϵ.

Infinite traces may seem useless at first, and indeed not all formalisms take them into account. In practice, only finite traces will be produced and verified by a monitor. However, infinite traces are needed to precisely describe the semantics of non-terminating programs, such as web servers, where termination is always considered an error as these systems are supposed to run indefinitely. Specifications of such systems can only accept infinite traces, and without them, their semantics would be empty. In such cases, real monitors working incrementally (online verification) will get a finite sequence and check whether it is a valid *finite prefix* of those infinite traces.

Example 1. (Function call events) Monitoring function calls is a common instrumentation technique in runtime verification, as it can be used, for instance, to ensure correct usage of APIs. In this setting, a sensible event set could be {funcName: s, args: l} where serialized events include basic metadata about the invocation, namely the name of the invoked function and its arguments. Meta-variables s and l range over string literals and event lists, according to Definition 1.

In this context, the monitoring af program operating on a file would produce, for instance, a trace $t = o_1 o_2 o_3 o_4$, with the following events:

$o_1 = $ {funcName: "*open*", args: ["*john.txt*", "*r*"]}

$o_2 = $ {funcName: "*write*", args: ["*hello*"]}

$o_3 = $ {funcName: "*write*", args: ["*world*"]}

$o_4 = $ {funcName: "*close*", args: []}

A more advanced instrumentation could provide many more metadata, for instance returned objects, thrown exceptions, target object of method calls in object-oriented programming, etc.

As it happens in the example above, the set of events O is often infinite.

The sets $\mathcal{K}$, $\mathcal{N}$, $\mathcal{S}$ and O will be left implicit when they are not important.

2.1.1 *Circularity*

Note that the event language described above cannot encode circular objects. We chose not to support them for the following reasons:

- Events as described in this section are a subset of the commonly understood and supported JSON standard, while circular objects are not, and only unofficial extensions exist.

- Serialization and deserialization of event objects is fast thanks to the existing highly optimized JSON libraries; on the other hand,

dealing with circularity requires ad hoc algorithms and some bookkeeping, slowing down the whole verification procedure.

- Though circular objects are common in some languages (e.g., JavaScript) in our experience the circularity itself is usually not interesting from the specification point of view.

- Circularity can be handled by unfolding the self-referencing object and truncating its depth, a step that is actually done for any object in our instrumentation implementation to avoid serialization of huge object graphs (though this aspect of the instrumentation can be fine-tuned according to the specification needs).

2.2 EVENT TYPES

In order to be as general as possible, instrumentation components are expected to provide all required information regarding the observed run of the program. This way the developer can write a broad range of properties to be enforced. Single specifications, however, rarely need all of these metadata and benefit from a more abstract way to reason about events. *Event types* are designed to solve this issue and fill the gap between what the instrumentation observes and what the specification (and thus the monitor) needs. They are based on syntactic *event patterns*, a convenient syntax to define (and reason on) sets of related events with a common structure.

Our notion of event types resembles so-called *symbolic events* (Falcone, Havelund, and Reger, 2013). However, that notion of symbolic events is only based on an event name and a list of variables, while RML patterns and event types declaration are much more general and flexible, as it will be shown in the rest of this Section.

Considering again Example 1, for instance, it would be nice to reason about all file operations, both read and write, regardless of the arguments, as they are all I/O operations on the same file. This can be achieved with the event type *rw* declared as follows:

$$rw \text{ matches } \{\text{funcName: } \text{``}read\text{''} \mid \text{``}write\text{''}\}$$

(the operator | is meant to be read as "or").

The other crucial aspect of event types is that they can contain *variables*. During the verification procedure, such variables will be bound to the correct values from the observed event matching the pattern, and the specification will be able to use this variable, effectively supporting *parametric specifications*, that is, specifications depending on values that will only be known at runtime.

In Example 1, one could be interested in capturing the name of opened file and bound to a variable, in order to check whether *that* particular file is later closed. Event type $open(n)$ below does exactly that:

$$open(n) \text{ matches } \{\text{funcName: } \text{``}open\text{''}, \text{args: } [n, \ldots]\}$$

The rest of this section precisely defines how event types are declared, what does it mean for an event to match an event type, and finally how they can be used.

2.2.1 *Event patterns*

Definition 3. (Event pattern) Given a set of events O over field keys $\mathcal{K}$, and a set of variables $\mathcal{X}$, the sets of *event patterns* $\mathcal{P}$ is inductively defined according to the following context-free grammar:

$$
\begin{array}{rcll}
\mathcal{P} & ::= & op & \\
& | & lp & \\
& | & \_ & \text{(wildcard pattern)} \\
& | & \mathcal{P} \mid \mathcal{P} & \text{(choice pattern)} \\
& | & \mathcal{X} & \text{(variable)} \\
& | & \mathcal{E} & \text{(event expression literal)} \\
op & ::= & \{\mathcal{K}: \mathcal{P}, \dots, \mathcal{K}: \mathcal{P}\} & \text{(object pattern)} \\
lp & ::= & [\mathcal{P}, \dots, \mathcal{P}] & \text{(list pattern)} \\
& | & [\mathcal{P}, \dots, \mathcal{P}, \text{more}] & \text{(flexible list pattern)}
\end{array}
$$

The order of key-value pairs in object patterns is not relevant.

The structure of patterns closely resemble the shape of events, thus objects, list and primitive values are included.

Object patterns are "open": they are meant to be understood as encoding the required set of fields an event object must have, though more are allowed. List patterns can either encode the exact number of elements (list pattern) or just the first n elements, possibly followed by others (flexible list pattern).

The wildcard pattern _ can match anything inside an event. Finally, the operator that actually describe the choice among multiple structures is |, and it can appear in any point for maximum flexibility.

Example 2. (Function call event pattern) Event patterns can be used to abstract over a (possibly infinite) set of events. For instance, the following event pattern matches any call to function foo with exactly one argument (to be bound to a variable) returning a non-empty list (which is ignored):

$$\{\text{funcName: } ``foo", \text{args: } [x], \text{result: } [\_, \text{more}]\}$$

2.2.2 *Event substitutions*

In order to define if and how patterns match events, a notion of substitution is needed. We will take the approach of considering substitutions as partial function on variables only containing relevant mapping, as

opposed to total maps behaving like the identity on irrelevant variables. Both the definitions are often used in the literature.

Definition 4. (Event substitution) Given a set of event patterns $\mathcal{P}$ and variables $\mathcal{X}$, an *event substitution* is a partial map $\sigma\colon \mathcal{X} \rightharpoonup \mathcal{P}$. The set of all such substitutions is denoted as Σ.

In order to refine patterns at runtime when variables are instantiated thanks to the observed events, we need to apply substitution to patterns. The definition of such application is entirely standard and is given inductively over the structure of patterns.

Definition 5. (Event substitution application) Given a set of event patterns $\mathcal{P}$ over event expressions $\mathcal{E}$ and variables $\mathcal{X}$, the *application* $(\Sigma \times \mathcal{P} \to \mathcal{P})$ of a substitution σ to a pattern $p \in \mathcal{P}$ is denoted as σp and is inductively defined by the following equations:

$$\sigma\{k_1\colon p_1,\dots,k_n\colon p_n\} = \{k_1\colon \sigma p_1,\dots,k_n\colon \sigma p_n\}$$
$$\sigma[p_1,\dots,p_n] = [\sigma p_1,\dots,\sigma p_n]$$
$$\sigma[p_1,\dots,p_n,\text{more}] = [\sigma p_1,\dots,\sigma p_n,\text{more}]$$
$$\sigma\_ = \_$$
$$\sigma(p_1 \mid p_2) = (\sigma p_1) \mid (\sigma p_2)$$
$$\sigma x = \sigma(x) \qquad\qquad (x \in dom(\sigma))$$
$$\sigma x = x \qquad\qquad (x \notin dom(\sigma))$$
$$\sigma e = e \qquad\qquad (e \in \mathcal{E})$$

The domain of an event substitution σ, i.e., the set of variables over which it is defined, is denoted as $dom(\sigma)$. The functional restriction of a substitution σ on a set S is denoted as $\sigma \restriction S$. We will use the notation $\sigma_1 \cup \dots \cup \sigma_n$ to denote a substitution σ such that $dom(\sigma) = dom(\sigma_1) \cup \dots \cup dom(\sigma_n)$ and $\sigma \restriction dom(\sigma_1) = \sigma_1 \wedge \dots \wedge \sigma \restriction dom(\sigma_n) = \sigma_n$. In other words, substitution can only be merged when they agree on the mapping of all their common variables. $\sigma \setminus \{x_1,\dots,x_n\}$ denotes the restriction of σ that does not map the given variables, i.e., $\sigma \restriction (dom(\sigma) \setminus \{x_1,\dots,x_n\})$. Finally, $\emptyset$ denotes the empty substitution.

Example 3. (Function call event substitution) Previous Example 2 showed an example of a pattern p for events encoding function calls with one argument, represented by variable x. Assuming the following event o is observed:

$$\{\text{funcName}\colon \text{``}foo\text{''}, \text{args}\colon [7], \text{result}\colon [1,45]\}$$

Then $\sigma p = o$, with $\sigma = \{x \mapsto 7\}$.

2.2.3   *Event pattern matching*

Having defined events, patterns and substitutions, we can precisely define the matching of events (coming from observations made on the

$$\text{PM-OBJ}\ \frac{p_1 : e_1 \rightsquigarrow \sigma_1 \qquad \cdots \qquad p_n : e_n \rightsquigarrow \sigma_n}{\{k_1 : p_1, \ldots, k_n : p_n\} : \{k_1 : e_1, \ldots, k_{n+k} : e_{n+k}\} \rightsquigarrow \sigma}\ \sigma = \sigma_1 \cup \cdots \cup \sigma_n$$

$$\text{PM-LIST}\ \frac{p_1 : e_1 \rightsquigarrow \sigma_1 \qquad \cdots \qquad p_n : e_n \rightsquigarrow \sigma_n}{[p_1, \ldots, p_n] : [e_1, \ldots, e_n] \rightsquigarrow \sigma}\ \sigma = \sigma_1 \cup \cdots \cup \sigma_n$$

$$\text{PM-LIST-MORE}\ \frac{[p_1, \ldots, p_n] : [e_1, \ldots, e_n] \rightsquigarrow \sigma}{[p_1, \ldots, p_n, \text{more}] : [e_1, \ldots, e_{n+k}] \rightsquigarrow \sigma}$$

$$\text{PM-WILDCARD}\ \frac{}{\_ : e \rightsquigarrow \emptyset} \qquad\qquad \text{PM-VAR}\ \frac{}{x : e \rightsquigarrow \{x \mapsto e\}}$$

$$\text{PM-LEFT}\ \frac{p_1 : e \rightsquigarrow \sigma}{p_1 \mid p_2 : e \rightsquigarrow \sigma} \qquad \text{PM-RIGHT}\ \frac{p_1 \nmid e \qquad p_2 : e \rightsquigarrow \sigma}{p_1 \mid p_2 : e \rightsquigarrow \sigma}$$

$$\text{PM-EXP}\ \frac{}{e : e \rightsquigarrow \emptyset}$$

Figure 2.1: Inference system inductively defining the judgments $p : e \rightsquigarrow \sigma$ and
$p \nmid e$.
Part 1: rules defining successful pattern matching.

system under scrutiny) against user-defined event patterns; the result
will be either a substitution or a failure.

Definition 6. (Event pattern matching) An event pattern p is said to
match an event expression e with substitution σ when the judgement $p{:}e \rightsquigarrow \sigma$
holds. p is said *not to match e* when the judgement $p \nmid e$ holds. Both the
judgements are inductively defined by the inductive interpretation of
the inference system in Figure 2.1.

Successful pattern matching. $p : e \rightsquigarrow \sigma$ is inductively defined on the
structure of patterns.

Rule PM-OBJ allows event objects to have more fields beyond the ones
required by the pattern, though all of the substitutions produced by matching
single key-value pairs must be merge-able for the rule to be applicable.

Rule PM-LIST works on lists in a similar way, except the exact number of
elements is expected, while the more flexible pattern with final more (rule
PM-LIST-MORE) simply discards unnecessary elements.

The wildcard _ matches any event expression (rule PM-WILDCARD), just
like variables, though when the latter is encountered a substitution for that
variable is produced (rule PM-VAR).

The choice pattern $p_1 \mid p_2$ matches the event either when the left pattern does
(rule PM-LEFT) or when the left pattern does not match but the right one does
(rule PM-RIGHT); this keeps the pattern matching procedure deterministic.

Finally, an event expression can only match itself (rule PM-EXP), as neither
variables nor patterns are present.

$$\text{PMF-}{\scriptsize\text{OBJ}} \quad \frac{[p_1, \dots, p_n] \nmid [e_1, \dots, e_n]}{\{k_1 : p_1, \dots, k_n : p_n\} \nmid \{k_1 : e_1, \dots, k_{n+k} : e_{n+k}\}}$$

$$\text{PMF-}{\scriptsize\text{OBJ-OTHER}} \quad \frac{}{\{k_1 : p_1, \dots, k_n : p_n\} \nmid e} \ e \neq \{k_1 : e_1, \dots, k_{n+k} : e_{n+k}\}$$

$$\text{PMF-}{\scriptsize\text{LIST-MERGE}} \quad \frac{p_1 : e_1 \rightsquigarrow \sigma_1 \quad \cdots \quad p_n : e_n \rightsquigarrow \sigma_n}{[p_1, \dots, p_n] \nmid [e_1, \dots, e_n]} \ \nexists \sigma . \sigma = \sigma_1 \cup \cdots \cup \sigma_n$$

$$\text{PMF-}{\scriptsize\text{LIST-ELEM}} \quad \frac{p_i \nmid e_i}{[p_1, \dots, p_n] \nmid [e_1, \dots, e_n]} \ i \leq n$$

$$\text{PMF-}{\scriptsize\text{LIST-OTHER}} \quad \frac{}{[p_1, \dots, p_n] \nmid e} \ e \neq [e_1, \dots, e_n]$$

$$\text{PMF-}{\scriptsize\text{MORE}} \quad \frac{[p_1, \dots, p_n] \nmid [e_1, \dots, e_n]}{[p_1, \dots, p_n, \text{more}] \nmid [e_1, \dots, e_{n+k}]}$$

$$\text{PMF-}{\scriptsize\text{MORE-OTHER}} \quad \frac{}{[p_1, \dots, p_n, \text{more}] \nmid e} \ e \neq [e_1, \dots, e_{n+k}]$$

$$\text{PMF-}{\scriptsize\text{CHOICE}} \quad \frac{p_1 \nmid e \quad p_2 \nmid e}{p_1 \mid p_2 \nmid e} \qquad\qquad \text{PMF-}{\scriptsize\text{EXP}} \quad \frac{}{e \nmid e'} \ e \neq e'$$

Figure 2.1: Inference system inductively defining the judgments $p : e \rightsquigarrow \sigma$ and $p \nmid e$.
Part 2: rules defining failed pattern matching.

Failed pattern matching. $p \nmid e$ is also defined closely following the structure of event patterns, though some patterns and event expressions can fail to match in more than one way.

List patterns can fail for different reasons: one of the patterns does not match the corresponding expression (rule PMF-LIST-ELEM); all the patterns match but the resulting substitutions cannot be merged (rule PMF-LIST-MERGE); the expression is not a list of the correct length, or not a list at all (rule PMF-LIST-OTHER).

The same holds for objects (rules PMF-OBJ and PMF-OBJ-OTHER), and we reuse pattern matching on lists by discarding unneeded key-value pairs.

Flexible list patterns can fail because of the inner patterns not matching event elements (rule PMF-MORE) or because the event expression is not a list with enough elements, or not a list in the first place.

Finally, choice patterns do not match events when both the sub-patterns do not (rule PMF-CHOICE), and event expressions do not match other expressions different from themselves (rule PMF-EXP).

Note that, even if the two set of rules in Figure 2.1 are kept separated for the sake of clarity, formally they are part of the same inference system. This is necessary since the two judgements are mutually recursive and cannot be truly defined separately.

Example 4. (Function call event pattern matching) With the given rules for event pattern matching we can now formalize the generation of the substitution informally given in Example 3.

$$p = \{\text{funcName: ``}foo\text{''}, \text{args: } [x], \text{result: } [\_, \text{more}]\}$$
$$o = \{\text{funcName: ``}foo\text{''}, \text{args: } [7], \text{result: } [1, 45]\}$$

$$\text{PM-OBJ} \; \dfrac{\text{PM-EXP} \; \dfrac{}{\text{``}foo\text{''} : \text{``}foo\text{''} \rightsquigarrow \emptyset} \quad \text{PM-LIST} \; \dfrac{\text{PM-VAR} \; \dfrac{}{x : 7 \rightsquigarrow \{x \mapsto 7\}}}{[x] : [7] \rightsquigarrow \{x \mapsto 7\}} \quad \text{PM-LIST-MORE} \; \dfrac{\text{PM-WILDCARD} \; \dfrac{}{\_ : 1 \rightsquigarrow \emptyset}}{[\_, \text{more}] : [1, 45] \rightsquigarrow \emptyset}}{p : o \rightsquigarrow \{x \mapsto 7\}}$$

2.2.4 *Event type declaration*

We now describe the main abstraction mechanism over events used in RML. Event types are a crucial mechanism in RML for many reasons:

- *Set of events*: an event type can conveniently represent an arbitrary set of events at once, even an infinite one.

- *Abstraction*: while events are the abstraction level chosen for the instrumentation, event types can select the data that is relevant for the specification being written, enabling the user to reason at the desired level of abstraction.

- *Data-oriented properties*: event types with variables can be used to capture data from the observed event to be later used in subsequent verification steps, supporting data-oriented properties.

- *Reuse*: event types can be defined on top of other event types, bringing modularity and reusability.

Definition 7. (Event type) Given a set of event type names Θ and variables $\mathcal{X}$, an *event type* is a term $\theta(x_1, \dots, x_n)$, with $\theta \in \Theta$ and $x_1, \dots, x_n \in \mathcal{X}$.

Event types can be declared as follows.

Definition 8. (Event type declaration) The set of *event type declarations* $\mathcal{D}$ over a set of event type names Θ, event patterns $\mathcal{P}$ and variables $\mathcal{X}$ is inductively defined by the following context-free grammar:

$$
\begin{array}{llll}
\mathcal{D} & ::= & \Theta(\mathcal{X}, \dots, \mathcal{X}) \text{ matches } \mathcal{P} & \text{(positive declaration)} \\
& | & \Theta(\mathcal{X}, \dots, \mathcal{X}) \text{ not matches } \mathcal{P} & \text{(negative declaration)} \\
& | & \Theta(\mathcal{X}, \dots, \mathcal{X}) \text{ matches } \widetilde{\mathcal{T}_1} | \cdots | \widetilde{\mathcal{T}_n} & \text{(positive derived decl.)} \\
& | & \Theta(\mathcal{X}, \dots, \mathcal{X}) \text{ not matches } \widetilde{\mathcal{T}_1} | \cdots | \widetilde{\mathcal{T}_n} & \text{(negative derived decl.)} \\
& & \qquad\qquad\qquad (n > 0) & \\
\widetilde{\mathcal{T}} & ::= & \Theta(\mathcal{P}, \dots, \mathcal{P}) & \text{(event type pattern)}
\end{array}
$$

The set of *event type patterns* is denoted as $\widetilde{\mathcal{T}}$.

Event type declarations specify one or more patterns: note that these should be object event patterns, since we expect events to be objects with key-value pairs (and not, for instance, lists). Patterns with choices (which can be nested at any level) allow event types to describe objects with different structures. It is also possible to define event types that do *not* match a given pattern: this is especially useful to denote a set of events we are not interested in, to be later filtered out with RML specific operators.

Derived declarations support modularity in the definition of event types, reusing other declarations. Note that event type patterns ($\widetilde{\mathcal{T}}$) may contain not only variables but arbitrary patterns: this gives the flexibility to reuse existing event type declarations while (partially) instantiating *some* arguments.

Event substitution is lifted to event type patterns in the natural way:

$$\sigma\theta(p_1, \dots, p_n) = \theta(\sigma p_1, \dots, \sigma p_n)$$

Example 5. (Data structure event declaration) The following event type declarations are an example for a possible data structure specification:

$$
\begin{array}{rl}
add(e) \text{ matches} & \{\text{funcName: } ``add'', \text{args: } [e]\} \\
remove(e) \text{ matches} & \{\text{funcName: } ``remove'', \text{args: } [e]\} \\
size(l) \text{ matches} & \{\text{funcName: } ``size'', \text{result: } l\} \\
isEmpty(b) \text{ matches} & \{\text{funcName: } ``isEmpty'', \text{result: } b\} \\
structural \text{ matches} & add(\_) \mid remove(\_) \\
nonStructural \text{ not matches} & structural
\end{array}
$$

Finally, since declarations already include a notion of negation, we can define a simple translation from a declaration to its negated version.

Definition 9. (Declaration negation) Given an event type declaration $d \in \mathcal{D}$, its *negation* $\overline{d} \in \mathcal{D}$ is defined as follows:

$$\overline{\theta(x_1, \dots, x_n) \text{ matches } p} = \overline{\theta}(x_1, \dots, x_n) \text{ not matches } p$$

$$\overline{\theta(x_1, \dots, x_n) \text{ not matches } p} = \overline{\theta}(x_1, \dots, x_n) \text{ matches } p$$

$$\overline{\theta(x_1, \dots, x_n) \text{ matches } \tau_1 | \cdots | \tau_m} = \overline{\theta}(x_1, \dots, x_n) \text{ not matches } \tau_1 | \cdots | \tau_m$$

$$\overline{\theta(x_1, \dots, x_n) \text{ not matches } \tau_1 | \cdots | \tau_m} = \overline{\theta}(x_1, \dots, x_n) \text{ matches } \tau_1 | \cdots | \tau_m$$

$\overline{\theta}$ is assumed to be a fresh, unused event type name.

Intuitively, the negation of a declaration d is meant to describe all the events that do *not* match d. See the next section for its use.

Finally, we are now able to specify the semantics of event types, the building blocks of TC. Since a specification contains many event type declarations, and such declarations may depend on other ones (derived declarations in Definition 8), we must take all of them into account in order to decide whether an event type matches an observed event.

Definition 10. (Event type matching) Given a list of event type declarations $\Gamma \in \mathcal{D}^*$, an event type pattern $\tau \in \mathcal{T}$ *matches* an event $e \in \mathcal{E}$ with substitution $\sigma \in \Sigma$ if and only if $\Gamma \vdash \Gamma ; \tau : e \rightsquigarrow \sigma$ holds. Conversely, τ *does not match e* if and only if $\Gamma \vdash \Gamma ; \tau \nmid e$. Both the judgements are inductively defined by the inference system in Figure 2.2.

In order to define event type matching, both a successful and a failed judgement are needed, since event type declarations can either be positive or negative (Definition 8). This is also the reason why event pattern matching was previously defined in a similar way (Definition 6). The two predicates need to be defined simultaneously, as they are mutually recursive and cannot be stratified. The judgement definitions includes:

- the list Γ of *all* event type declarations available (left-hand-side of $\vdash$);

- a list of event type declarations $d_1, \dots, d_n$ left to consider (right-hand-side of $\vdash$);

- the event type pattern τ we are actually trying to match against;

- the event expression e that comes from the instrumentation;

- successful matching produces the event substitution σ resulting from the underlying pattern matching operation, instantiating event type parameters.

Successful event type matching

The rule ETM-STEP allows to "forget" about the first of the declarations under consideration if it does not make τ match e as needed, and move to the next ones. This makes the process deterministic, as we go forward in the list of declarations only if needed, ensuring the first suitable declaration is used, if any. Indeed, all the other rules always use the first declaration of the list.

In direct declarations (ETM-DIRECT) the event has to match the pattern given in the declaration ($p : e \rightsquigarrow \sigma'$), and the parameters from the chosen declaration needs to be substituted with the arguments, as they could partially instantiate event type parameters.

Negated direct declarations (ETM-DIRECT-NOT) are handled in a similar way, though the pattern matching is expected to fail ($p \nmid e$), and no substitution is produced: a failed matching should not bind any variable. Negated declarations are meant to be used only when all the variables involved, if any, have been instantiated in previous verification steps.

To preserve determinism also at the level of event type matching, derived declarations (rule ETM-DERIVED) only move to a certain event type pattern

$$\text{ETM-STEP} \quad \frac{\Gamma \vdash d_1 \, ; \tau \,/\, e \qquad \Gamma \vdash d_2, \dots, d_n \, ; \tau : e \rightsquigarrow \sigma}{\Gamma \vdash d_1, \dots, d_n \, ; \tau : e \rightsquigarrow \sigma}$$

$$\text{ETM-DIRECT} \quad \frac{}{\Gamma \vdash d_1, \dots, d_n \, ; \theta(p_1, \dots, p_m) : e \rightsquigarrow \sigma'} \quad \begin{array}{c} d_1 = \theta(x_1, \dots, x_m) \text{ matches } p \\ \sigma = \{x_1 \mapsto p_1, \dots, x_m \mapsto p_m\} \\ \sigma p : e \rightsquigarrow \sigma' \end{array}$$

$$\text{ETM-DIRECT-NOT} \quad \frac{}{\Gamma \vdash d_1, \dots, d_n \, ; \theta(p_1, \dots, p_m) : e \rightsquigarrow \emptyset} \quad \begin{array}{c} d_1 = \theta(x_1, \dots, x_m) \text{ not matches } p \\ \sigma = \{x_1 \mapsto p_1, \dots, x_m \mapsto p_m\} \\ \sigma p \,/\, e \end{array}$$

$$\text{ETM-DERIVED} \quad \frac{\begin{array}{c} \Gamma \vdash \Gamma \, ; \sigma \tau_1 \,/\, e \\ \vdots \\ \Gamma \vdash \Gamma \, ; \sigma \tau_{j-1} \,/\, e \\ \Gamma \vdash \Gamma \, ; \sigma \tau_j : e \rightsquigarrow \sigma' \end{array}}{\Gamma \vdash d_1, \dots, d_n \, ; \theta(p_1, \dots, p_m) : e \rightsquigarrow \sigma'} \quad \begin{array}{c} d_1 = \theta(x_1, \dots, x_m) \text{ matches } \tau_1 | \cdots | \tau_q \\ \sigma = \{x_1 \mapsto p_1, \dots, x_m \mapsto p_m\} \\ 1 \le j \le q \end{array}$$

$$\text{ETM-DERIVED-NOT} \quad \frac{\begin{array}{c} \Gamma \vdash \Gamma \, ; \sigma \tau_1 \,/\, e \\ \vdots \\ \Gamma \vdash \Gamma \, ; \sigma \tau_n \,/\, e \end{array}}{\Gamma \vdash d_1, \dots, d_n \, ; \theta(p_1, \dots, p_m) : e \rightsquigarrow \emptyset} \quad \begin{array}{c} d_1 = \theta(x_1, \dots, x_m) \text{ not matches } \tau_1 | \cdots | \tau_q \\ \sigma = \{x_1 \mapsto p_1, \dots, x_m \mapsto p_m\} \end{array}$$

Figure 2.2: Multiple inference system inductively defining the two (mutually recursive) judgements $\Gamma \vdash d_1, \dots, d_n ; \tau : e \rightsquigarrow \sigma$ and $\Gamma \vdash d_1, \dots, d_n ; \tau \,/\, e$ Part 1: rules defining the successful event type matching predicate $\Gamma \vdash d_1, \dots, d_n \, ; \tau : e \rightsquigarrow \sigma$.

ETFM-MAIN $\dfrac{}{\Gamma \vdash \emptyset \,;\, \tau \not| e}$

ETFM-NAME-AR $\dfrac{\Gamma \vdash d_1, \dots, d_{i-1}, d_{i+1}, \dots, d_n \,;\, \theta(p_1, \dots, p_m) \not| e}{\Gamma \vdash d_1, \dots, d_n \,;\, \theta(p_1, \dots, p_m) \not| e}$ $\begin{array}{l} d_i = \theta'(x_1, \dots, x_n) \dots \\[4pt] \theta \neq \theta' \vee m \neq n \end{array}$

ETFM-DIRECT $\dfrac{\Gamma \vdash d_1, \dots, d_{i-1}, d_{i+1}, \dots, d_n \,;\, \theta(p_1, \dots, p_m) \not| e}{\Gamma \vdash d_1, \dots, d_n \,;\, \theta(p_1, \dots, p_m) \not| e}$ $\begin{array}{l} d_i = \theta(x_1, \dots, x_m) \text{ matches } p \\[4pt] \sigma = \{x_1 \mapsto p_1, \dots, x_m \mapsto p_m\} \\[4pt] \sigma p \not| e \end{array}$

ETFM-DIRECT-NOT $\dfrac{\Gamma \vdash d_1, \dots, d_{i-1}, d_{i+1}, \dots, d_n \,;\, \theta(p_1, \dots, p_m) \not| e}{\Gamma \vdash d_1, \dots, d_n \,;\, \theta(p_1, \dots, p_m) \not| e}$ $\begin{array}{l} d_i = \theta(x_1, \dots, x_m) \text{ not matches } p \\[4pt] \sigma = \{x_1 \mapsto p_1, \dots, x_m \mapsto p_m\} \\[4pt] \sigma p : e \rightsquigarrow \sigma' \end{array}$

ETFM-DERIVED $\dfrac{\begin{array}{c} \Gamma \vdash \Gamma \,;\, \sigma \tau_1 \not| e \\ \vdots \\ \Gamma \vdash \Gamma \,;\, \sigma \tau_q \not| e \\ \Gamma \vdash d_1, \dots, d_{i-1}, d_{i+1}, \dots, d_n \,;\, \theta(p_1, \dots, p_m) \not| e \end{array}}{\Gamma \vdash d_1, \dots, d_n \,;\, \theta(p_1, \dots, p_m) \not| e}$ $\begin{array}{l} d_i = \theta(x_1, \dots, x_m) \text{ matches } \tau_1| \cdots |\tau_q \\[4pt] \sigma = \{x_1 \mapsto p_1, \dots, x_m \mapsto p_m\} \end{array}$

ETFM-DERIVED-NOT $\dfrac{\begin{array}{c} \Gamma \vdash \Gamma \,;\, \sigma \tau_j : e \rightsquigarrow \sigma' \\ \Gamma \vdash d_1, \dots, d_{i-1}, d_{i+1}, \dots, d_n \,;\, \theta(p_1, \dots, p_m) \not| e \end{array}}{\Gamma \vdash d_1, \dots, d_n \,;\, \theta(p_1, \dots, p_m) \not| e}$ $\begin{array}{l} d_i = \theta(x_1, \dots, x_m) \text{ not matches } \tau_1| \cdots |\tau_q \\[4pt] \sigma = \{x_1 \mapsto p_1, \dots, x_m \mapsto p_m\} \\[4pt] 1 \leq j \leq q \end{array}$

Figure 2.2: Multiple inference system inductively defining the two (mutually recursive) judgements $\Gamma \vdash d_1, \dots, d_n ; \tau{:}e \rightsquigarrow \sigma$ and $\Gamma \vdash d_1, \dots, d_n ; \tau \not| e$ Part 2: rules defining the failed event type matching predicate $\Gamma \vdash d_1, \dots, d_n \,;\, \tau \not| e$.

if all the previous one failed to match the event. It is important to go back to the whole list of available declarations Γ since the chosen event type could be defined by any declaration. In negated derived declarations (rule ETM-DERIVED-NOT) all the possible event type patterns are expected not to match the event.

Failed event type matching

In this case, the matching procedure must fail according to *all* of the declarations, in order to assert that the observed event really does not match the event type. This, and the fact that no substitution is produced, means that, in every rule, the declaration to use can be chosen without restrictions, as they all need to be analyzed (though still one at a time).

The base case ETFM-MAIN is reached when there are no more declarations left to consider, and we can conclude τ does not match e according to declarations Γ.

The four rules ETFM-DIRECT, ETFM-DIRECT-NOT, ETFM-DERIVED, and ETFM-DERIVED-NOT are similar to the ones we just described, with success and failure swapped. Since $\tau_1|\cdots|\tau_q$ is essentially a union, it also behave differently when negated. The rule for derived declarations ensures none of the available patterns match the event, while the rule for negated derived declarations only needs to find one pattern that actually matches the event.

All those rules pick one declaration and proves that matching fails, and the premise ensures that also all the other declarations agree with it.

Double negations can be confusing. When we want to show that a pattern does *not* match an event, and we do so by using a *negated* declaration, we have to prove that the event actually matches the inner pattern provided by the declaration.

As opposed to successful matching, failure has an additional base case, that is, the declaration taken into account defines an event type with different name or arity, thus matching trivially fails (rule ETFM-NAME-AR).

Example 6. (Matching) Example 1 shows a possible event domain for function calls, which we now use, augmented with return values. On top of that, some examples of event type declarations follow (*fd* is meant to identify a file descriptor[2]):

> $d_1 = open(fd)$ matches {funcName: "*open*", result: *fd*}
>
> $d_2 = rw(fd)$ matches {funcName: "*read*" | "*write*", args: [*fd*, more]}
>
> $d_3 = close(fd)$ matches {funcName: "*close*", args: [*fd*, more]}
>
> $d_4 = relevant$ matches $open(\_) | rw(\_) | close(\_)$
>
> $d_5 = ignored$ not matches *relevant*

Let us consider the following events:

> $o_1 = $ {funcName: "*open*", args: ["*log.txt*"], result: 42}
>
> $o_2 = $ {funcName: "*random*", args: [], result: [25634]}
>
> $o_3 = $ {funcName: "*write*", args: [42, 25634], result: *true*}

Assuming a C-like syntax, the three events above would be generated by the instrumentation to encode the following function calls:

---

2 File descriptors are identifiers used in Unix and related operating systems to indicate files and other I/O resources.

```
int fd = open("log.txt");
int n = random();
write(fd, n);
```

As expected, event type semantics leads to o_1 matching $open(fd)$ with substitution $\sigma = \{fd \mapsto 42\}$ ($\Gamma = d_1, \ldots, d_5$):

$$\text{ETM-\textsc{direct}} \; \frac{\begin{array}{c} p = \{\text{funcName: "}open\text{", result: } fd\} \\ p : o_1 \rightsquigarrow \sigma \end{array}}{\Gamma \vdash d_1, \ldots, d_5 \, ; open(fd) : o_1 \rightsquigarrow \sigma}$$

$$\text{PM-\textsc{obj}} \; \frac{\text{PM-\textsc{exp}} \; \dfrac{}{\text{"}open\text{"} : \text{"}open\text{"} \rightsquigarrow \emptyset} \qquad \text{PM-\textsc{var}} \; \dfrac{}{fd : 42 \rightsquigarrow \sigma} \quad \sigma = \{fd \mapsto 42\}}{\{\text{funcName: "}open\text{", result: } fd\} : o_1 \rightsquigarrow \sigma} \quad \sigma = \emptyset \cup \sigma$$

As shown above, the semantics is clearly stratified: event type matching deals with declarations and their dependences, while pattern matching handles the lower-level term manipulation and generates substitutions. Note that, in rule PM-\textsc{obj}, only the fields required by the pattern are taken into account, while *args* is ignored.

Similarly, o_1 also matches the event type *relevant* (the same holds for o_3):

$$\text{ETM-\textsc{step}} \; \frac{\text{ETMF-\textsc{name-ar}} \; \dfrac{\text{ETMF-\textsc{main}} \; \dfrac{}{\Gamma \vdash \emptyset \, ; relevant \nmid o_1}}{\Gamma \vdash d_1 \, ; relevant \nmid o_1} \quad \begin{array}{c} d_1 = open(fd) \ldots \\ relevant \neq open \end{array} \quad \text{ETM-\textsc{step}} \; \frac{\ddots \quad \text{ETM-\textsc{derived}} \; \dfrac{\vdots}{\Gamma \vdash d_4, d_5 \, ; relevant : o_1 \rightsquigarrow \emptyset}}{\begin{array}{c} \vdots \\ \Gamma \vdash d_2, \ldots, d_5 \, ; relevant : o_1 \rightsquigarrow \emptyset \end{array}}}{\Gamma \vdash d_1, \ldots, d_5 \, ; relevant : o_1 \rightsquigarrow \emptyset}$$

o_2 on the other hand only matches the event type *ignored*.

2.4 TC SYNTAX

This Section is devoted to the syntax of TC, the calculus underlying RML, to which the latter is compiled to. At this level, an important design decision is how to support recursion.

2.4.1 *Regular terms*

Many formalisms support recursion through an explicit fixpoint operator; these are often called least or greatest fixed point operators, depending on whether they are meant to be understood according to an inductive or coinductive interpretation. The modal μ-calculus, for instance, is a fixed-point logic including both features (Kozen, 1983).

A different approach is based on *regular terms* (a.k.a. rational or cyclic terms) (Ancona and Corradi, 2014, 2016; Courcelle, 1983; Frisch, Castagna, and Benzaken, 2008). Traditionally, syntactic terms are inductive, well-founded entities that can be seen as finite trees; this is the idea behind abstract syntax trees used in programming languages semantics. A regular term is a possibly infinite tree, though the set of its subtrees must be finite. They can be finitely represented as graphs; term ϕ depicted in Figure 2.3[3], for instance, is an infinite tree with the following subtrees:

---

3 Prefix operator ':' and union operator '$\vee$' are part of TC and will be presented in the rest of this Section; their semantics, however, is not relevant for the example being discussed, they could be any binary operators.

1. the whole infinite tree rooted at '∨', encoding the term ϕ itself;

2. the tree only containing ϵ, encoding term ϵ;

3. the infinite tree rooted at ':', encoding the term $\theta : \phi$;

4. the tree only containing θ, encoding term θ.
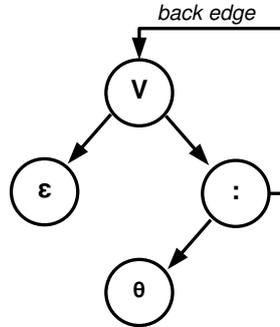
Figure 2.3: Syntax graph of the unique term ϕ such that $\phi = \epsilon \vee (\theta : \phi)$.

In order to have a finite *syntactic* representation of regular terms, syntactic equations can be used: this is the approach used in this work. The term ϕ from Figure 2.3, for instance, can be described as "the (unique) term satisfying the recursive syntactic equation $\phi = \epsilon \vee (\theta : \phi)$". Note that this is a *coinductive* interpretation of syntactic, recursive equations: the one above does not describe any finite term ϕ.

However, we will only use equations as a way to *describe* regular terms, but at the abstract level, it is arguably easier to consider them as infinite trees (or their equivalent graph). A formal treatment of systems of syntactic equations describing regular terms has been written by Courcelle (1983).

Finally, in order to describe some regular terms, multiple (possibly mutually recursive) equations are needed. As an example, consider the regular term ϕ satisfying the following system of equations:

$$\phi = \phi_1 \vee \phi_2$$
$$\phi_1 = \theta_1 : \phi_1$$
$$\phi_2 = \epsilon \vee (\theta_2 : \phi_2)$$

Equations can also be mutually recursive. In any case, the system of equations must be finite, otherwise the term satisfying them would not be regular.

2.4.2   *Syntax and operators*

We now formalize the syntax of TC and give an intuition about the semantics of the operators. Some syntactic components have been previously defined in this Chapter.

First, the class of expressions about observed data that can be used in specification is presented. This is useful to express data-oriented properties.

Definition 11. (Data expressions) A *data expression* $\xi \in \Xi$ is an arithmetic (over a set of numbers $\mathcal{N}$) and boolean expression inductively defined by the following context-free grammar:

$$
\begin{array}{llll}
\Xi & ::= & \Xi \, op \, \Xi & op \in \{+, -, \times, \div\} \quad \text{(arithmetic expression)} \\
& | & \Xi \, op' \, \Xi & op' \in \{<, \leq, =, \geq, >\} \quad \text{(relational expression)} \\
& | & \Xi \, op'' \, \Xi \quad | \quad \neg\Xi & op'' \in \{\wedge, \vee\} \quad \text{(boolean expression)} \\
& | & \mathcal{X} & \text{(variables)} \\
& | & \mathcal{N} & \text{(numbers)} \\
& | & \mathbb{B} & \text{(booleans)}
\end{array}
$$

Data expressions are entirely standard arithmetic and boolean expressions. The supported operators are the ones currently implemented in our prototype, though it would be easy to add new ones without affecting the rest of the implementation (and semantics).

The following definition presents the syntax of the calculus.

Definition 12. (TC syntax) A *TC term* $\phi \in \Phi$ is a *regular* term over the operators described by the following grammar:

$$
\begin{array}{llll}
\Phi & ::= & \epsilon & \text{(empty)} \\
& | & \mathbf{0} & \text{(none)} \\
& | & \mathbf{1} & \text{(all)} \\
& | & \mathcal{I} : \Phi & \text{(prefix)} \\
& | & \Phi \cdot \Phi & \text{(concatenation)} \\
& | & \Phi \vee \Phi & \text{(union)} \\
& | & \Phi \wedge \Phi & \text{(intersection)} \\
& | & \Phi \, | \, \Phi & \text{(shuffle)} \\
& | & \{\mathcal{X}; \Phi\} & \text{(parametric term)} \\
& | & g\langle \Xi, \dots, \Xi \rangle & \text{(generic application)} \\
& | & \text{if } \Xi \text{ then } \Phi \text{ else } \Phi & \text{(conditional term)} \\
g & ::= & \langle \mathcal{X}, \dots, \mathcal{X} \rangle.\Phi & \text{(generic term)}
\end{array}
$$

Though context-free grammars, as the one given above, are usually meant to define languages inductively, in this case we abuse such notation to give a concise representation of the language operators, despite terms are allowed to be regular.

A TC term encodes a low-level specification whose semantics is the set of traces that are correct with respect to such specification. While the semantics will be fully formalized in the next section, the intuitive meaning of the operators above is as follows:

EMPTY (ϵ) the empty trace is expected;

NONE ($\mathbf{0}$) no trace is valid, an error has been encountered;

ALL ($\mathbf{1}$) any trace is valid, requested property has already been proved to hold;

PREFIX ($\theta(p_1, \dots, p_n) : \phi$) an event matching $\theta(p_1, \dots, p_n)$ is expected, followed by a trace according to ϕ;

CONCATENATION ($\phi_1 \cdot \phi_2$) a trace that is valid according to ϕ_1 followed by a trace according to ϕ_2;

UNION ($\phi_1 \vee \phi_2$) traces that are valid either with respect to ϕ_1 or to ϕ_2 are accepted (a deterministic approach to this operator will be formalized in this Chapter);

INTERSECTION ($\phi_1 \wedge \phi_2$) in order to be valid, a trace must be correct with respect to both ϕ_1 and ϕ_2;

SHUFFLE ($\phi_1 \mid \phi_2$) the set of traces resulting from the shuffle of those from ϕ_1 and from ϕ_2 (again, determinism needs to be taken into account);

PARAMETRIC TERM ($\{x; \phi\}$) variable x is introduced for use in ϕ (and it will be assigned a value during matching) but its scope is limited by the delimiters, and the semantics is that of ϕ;

GENERIC APPLICATION ($g\langle \xi_1, \dots, \xi_n \rangle$) the generic term $g = \langle x_1, \dots, x_n \rangle.\phi$ is instantiated with arguments $\xi_1, \dots, \xi_n$, and the semantics is given by ϕ after the appropriate substitution has been applied;

CONDITIONAL TERM (if ξ then ϕ_1 else ϕ_2) if ξ evaluates to *true* then the semantics of the whole expression is that of ϕ_1, while if it evaluates to *false* then the semantics is that of ϕ_2;

GENERIC TERM ($\langle x_1, \dots, x_n \rangle.\phi$) ϕ contains variables $x_1, \dots, x_n$ that are meant to be instantiated by a generic application, then the semantics is that of ϕ after the appropriate substitution has been applied.

Example 7. (Stack specification term) The following TC term encodes a possible specification of a stack (also known as last-in-first-out queue):

$$\phi = \epsilon \vee ((push : \phi) \cdot pop \cdot \phi)$$

This example is meant to show the calculus syntax. More significant examples will be thoroughly discussed after the semantics will be presented.

2.4.3 *Parametric and generic specifications*

Two different kinds of parametricity are supported, and they are kept separated as they have different meanings. Parametric terms allow one to write parametric specifications, that is, specifications depending on values that will only be known at runtime. As an example, consider a protocol specifying that an arbitrary natural number n is expected, followed by exactly n events matched by a given event type θ. Such a simple protocol needs parametricity support as only after knowing the value of n we are able to detect the correct number of subsequent events.

Generic terms, on the other hand, are parametric with respect to variables that are meant to be instantiated when that portion of the specification comes into play. On one hand, generic terms improves modularity: the same (generic) specification can be used in different contexts by applying different arguments when it is employed. On the other hand, the combination of generics, conditionals and data expressions allows one to *compute* values during the verification process, greatly enhancing expressivity. This is different from parametric specifications in the form $\{x; \phi\}$, where x is just a placeholder for a value that will be soon discovered with the next events observed. Although different,

the two kinds of parametricity are quite useful when combined together, especially when the arguments of a generic specification are known at runtime, as happens in the simple protocol described above that can be specified by $\{x; \theta'(x) : g\langle x\rangle\}$, where g is the generic and regular term identified by the equation $g = \langle x'\rangle.\text{if } x' > 0 \text{ then } \theta : g\langle x' - 1\rangle \text{ else } \epsilon$; event type $\theta'(x)$ checks that a natural number x is sent, then $g\langle x\rangle$ requires that exactly x events of type θ are observed.

Finally, the syntax of TC keeps generic specifications and their applications separated to ensure that generic terms and applications match, and no higher-order generic manipulation is allowed, as it would make the verification process quite complex and possibly undecidable; as it will be shown in the examples in Section 4, all the operators above already allow users to express a set of very expressive properties.

2.5 REWRITING SEMANTICS

2.5.1 *Term substitution*

In the following, the previously defined notion of event substitution (Definition 4) is generalized so that variables can also be mapped to data expressions. This will be used in the rewriting semantics to map specification variables not only to observed data, but also to *computed* data.

Substitution and evaluation for data expressions are inductively defined following the expression structure in a straightforward way.

Definition 13. (Data expression substitution) Given a substitution σ and a data expression ξ, the *application* of σ to ξ, denoted as $\sigma\xi$, is inductively defined as follows:

$$\sigma(\neg\xi) = \neg(\sigma\xi)$$
$$\sigma(\xi_1 \; op \; \xi_2) = (\sigma\xi_1) \; op \; (\sigma\xi_2) \qquad (op \in \{+, -, \times, \div, <, \leq, =, \geq, >\})$$
$$\sigma x = \sigma(x) \qquad\qquad\qquad\qquad (x \in dom(\sigma))$$
$$\sigma x = x \qquad\qquad\qquad\qquad\quad (x \notin dom(\sigma))$$

Definition 14. (Data expression evaluation) Given a data expression ξ, the *evaluation* of ξ, denoted as $ev(\xi)$, is inductively defined as follows:

$$ev(\neg\xi) = ev(\neg)(ev(\xi))$$
$$ev(\xi_1 \; op \; \xi_2) = ev(op)(ev(\xi_1), ev(\xi_2)) \quad (op \in \{+, -, \times, \div, <, \leq, =, \geq, >\})$$

For every unary and binary operation symbol op, $ev(op)$ maps the operation symbol into the corresponding standard mathematical function.

Substitution on terms, on the other hand, needs to be defined by coinduction: regular terms are not well-founded objects (we recall they are a particular class of infinite trees) thus induction cannot be fruitfully used to define functions on them.

Definition 15. (Term substitution) The *application of substitution σ to a TC term ϕ*, denoted as $\sigma\phi$, is the function *coinductively* defined as follows:

$$\sigma\epsilon = \epsilon$$
$$\sigma\,\mathbf{0} = \mathbf{0}$$
$$\sigma\,\mathbf{1} = \mathbf{1}$$
$$\sigma(\tau : \phi) = (\sigma\tau) : (\sigma\phi)$$
$$\sigma(\phi_1 \cdot \phi_2) = (\sigma\phi_1) \cdot (\sigma\phi_2)$$
$$\sigma(\phi_1 \vee \phi_2) = (\sigma\phi_1) \vee (\sigma\phi_2)$$
$$\sigma(\phi_1 \wedge \phi_2) = (\sigma\phi_1) \wedge (\sigma\phi_2)$$
$$\sigma(\phi_1 \mid \phi_2) = (\sigma\phi_1) \mid (\sigma\phi_2)$$
$$\sigma\{x; \phi\} = \{x; (\sigma \backslash \{x\})\phi\}$$
$$\sigma(g\langle \xi_1, \ldots, \xi_n \rangle) = (\sigma g)\langle \sigma\xi_1, \ldots, \sigma\xi_n \rangle$$
$$\sigma(\text{if } \xi \text{ then } \phi_1 \text{ else } \phi_2) = \text{if } \sigma\xi \text{ then } \sigma\phi_1 \text{ else } \sigma\phi_2$$
$$\sigma(\langle x_1, \ldots, x_n \rangle.\phi) = \langle x_1, \ldots, x_n \rangle.(\sigma \backslash \{x_1, \ldots, x_n\}\phi)$$
$$\sigma x = \sigma(x) \qquad\qquad (x \in dom(\sigma))$$
$$\sigma x = x \qquad\qquad (x \notin dom(\sigma))$$

In parametric and generic specifications, the declared variable and parameters are removed from the domain of the substitution before propagating its application, to ensure the usual scoping rule on nested variable/parameter declarations: outermost declarations are hidden by innermost ones.

2.5.2   *Empty traces*

Since the system under test can halt at anytime, it is important to define whether a specification (or a term it rewrote to) accepts the empty trace, that is, whether termination is allowed at a specific step, so that the monitor can emit a sensible verdict.

Definition 16. (ϵ acceptance) A specification term ϕ *accepts the empty trace ϵ* iff the judgement $E(\phi)$ holds, according to the inductive interpretation of the inference system in Figure 2.4.

The base cases are the empty set of traces ϵ and the top operator $\mathbf{1}$, whose semantics trivially includes the empty trace (rule E-EMPTY and E-1).

A specification consisting of a concatenation, an intersection, or a shuffle (rule E-BIN) only accepts the empty trace if both the operands do so; otherwise, at least an event is expected, thus the empty trace is not a valid one for the specification.

For union, on the other hand (rule E-OR) it is sufficient that either the left or the right argument accepts the empty trace, since both encode valid behaviors.

Parametricity (rule E-PARAM) is not relevant in this context, thus the variable is simply ignored.

In generic applications (rule E-GENERIC), arguments need to be evaluated and the generic to be applied to check whether the specification accepts the empty trace. Similarly, in conditional terms (rule E-IF-*) the condition is evaluated in order to decide where to look for empty trace acceptance.

Definition 17. (ϵ non-acceptance) A specification term ϕ *does not accepts the empty trace ϵ* iff the judgement $NE(\phi)$ holds, according to the inductive interpretation of the inference system in Figure 2.5.

$$\text{E-\small EMPTY} \ \frac{}{E(\epsilon)} \qquad \text{E-1} \ \frac{}{E(\mathbf{1})} \qquad \text{E-\small BIN} \ \frac{E(\phi_1) \qquad E(\phi_2)}{E(\phi_1 \ op \ \phi_2)} \ op \in \{\cdot, \wedge, |\}$$

$$\text{E-\small OR} \ \frac{E(\phi_i)}{E(\phi_1 \vee \phi_2)} \ i \in \{1, 2\} \qquad \text{E-\small PARAM} \ \frac{E(\phi)}{E(\{x; \phi\})}$$

$$\text{E-\small GENERIC} \ \frac{E(\sigma \phi)}{E((\langle x_1, \dots, x_n \rangle . \phi) \langle \xi_1, \dots, \xi_n \rangle)} \ \sigma = \{x_1 \mapsto ev(\xi_1), \dots, x_n \mapsto ev(\xi_n)\}$$

$$\text{E-\small IF-TRUE} \ \frac{E(\phi_1)}{E(\text{if } \zeta \text{ then } \phi_1 \text{ else } \phi_2)} \ ev(\zeta) = \textit{true}$$

$$\text{E-\small IF-FALSE} \ \frac{E(\phi_2)}{E(\text{if } \zeta \text{ then } \phi_1 \text{ else } \phi_2)} \ ev(\zeta) = \textit{false}$$

Figure 2.4: Inference system inductively defining $E(\phi)$.

$$\text{NE-0} \ \frac{}{NE(\mathbf{0})} \qquad \text{NE-\small PREFIX} \ \frac{}{NE(\tau : \phi)} \qquad \text{NE-\small BIN} \ \frac{NE(\phi_i)}{NE(\phi_1 \ op \ \phi_2)} \ \begin{matrix} i \in \{1, 2\} \\ op \in \{\cdot, \wedge, |\} \end{matrix}$$

$$\text{NE-\small OR} \ \frac{NE(\phi_1) \qquad NE(\phi_2)}{NE(\phi_1 \vee \phi_2)} \qquad \text{NE-\small PARAM} \ \frac{NE(\phi)}{NE(\{x; \phi\})}$$

$$\text{NE-\small GENERIC} \ \frac{NE(\sigma \phi)}{NE((\langle x_1, \dots, x_n \rangle . \phi) \langle \xi_1, \dots, \xi_n \rangle)} \ \sigma = \{x_1 \mapsto ev(\xi_1), \dots, x_n \mapsto ev(\xi_n)\}$$

$$\text{NE-\small IF-TRUE} \ \frac{NE(\phi_1)}{NE(\text{if } \zeta \text{ then } \phi_1 \text{ else } \phi_2)} \ ev(\zeta) = \textit{true}$$

$$\text{NE-\small IF-FALSE} \ \frac{NE(\phi_2)}{NE(\text{if } \zeta \text{ then } \phi_1 \text{ else } \phi_2)} \ ev(\zeta) = \textit{false}$$

Figure 2.5: Inference system inductively defining $NE(\phi)$.

The base cases for $NE(\phi)$ are the empty set of traces $\mathbf{0}$ (rule NE-o) and the prefix operator $\tau : \phi$ (rule NE-PREFIX) which expects an event matched by τ, hence it cannot accept the empty trace.

Binary operators behave in a specular way to ϵ-acceptance judgement, which makes sense since we are essentially asking the opposite question.

The rest of the rules follow the structure of the term in the same way.

2.5.3   *Rewriting system*

The semantics of TC is given as a rewriting system, where steps are labelled by observed events. This has two direct advantages: first, the formal semantics can be used (as it has been done) to drive the implementation; second, it effectively supports online verification, working on one event at a time.

TC originates from trace expressions (Ancona, Ferrando, and Mascardi, 2016) and parametric trace expressions (Ancona, Ferrando, and Mascardi, 2017), both with a non-deterministic semantics. RML, and the underlying formalism TC, not only extends trace expressions with generics and conditionals (among other things) but also move to a deterministic semantics (Ancona, Franceschini, Ferrando, et al., 2019). This design choice was driven by our experience in applying our tool: although it is a bit more involved, the deterministic semantics allows a more efficient implementation, since backtracking is not needed; furthermore, as shown in Section 4, some specifications can be expressed more concisely. Our semantics forces a left-to-right evaluation strategy, and to do so, when formalizing the successful rewriting step, we also need to formalize a failed rewriting step for the observed event.

Definition 18. (TC rewriting semantics) Given a list of event type declarations Γ, a TC term ϕ and an event object o, ϕ *rewrites to* ϕ' *w.r.t.* o ($\phi \xrightarrow[\Gamma]{o} \phi'$) according to the inductive interpretation of the inference system in Figures 2.6 and 2.7.

When the event type declarations Γ is clear from the context or not relevant, it will be left implicit for the sake of readability.

The main rewriting predicate $\phi \xrightarrow{o} \phi'$ entirely relies on the auxiliary predicate $\phi \xrightarrow{o} \phi'$; σ, which additionally computes a substitution. At the top-level, however, we do not expect free variables (i. e., unbound variables that are not declared as $\{x; ...\}$), as specified in rule R-MAIN.

When a prefix is seen ($\tau : \phi$) the semantics relies on event type matching as defined in Definition 10 (rule R-PREFIX). This is the step where variables from parametric specifications are instantiated with data included in the event object. Matching and rewriting are kept strictly separated: this gives our system a great flexibility towards different kinds of systems and domains, as the interface to the instrumentation is specified by event type declaration, without affecting the semantics of the calculus.

Two different things can happen with union ($\phi_1 \vee \phi_2$). If the specification on the left accepts the event, than that term is chosen and the right one is forgotten (rule R-OR-L): the semantics makes no backtracking on the rewriting steps. Otherwise, if the left one does not accept the event but the right one does, then the latter is chosen (rule R-OR-R). The premise $\phi_1 \xrightarrow{o}\!\!\!\!\!/\;$ effectively makes the semantics of union deterministic by enforcing the left-to-right strategy.

Intersection (rule R-AND) requires both the specifications to accept the event, and furthermore the produced substitutions must be mergeable. This is to avoid that the same variable is bound to two different values in the two branches.

$$\text{R-MAIN} \quad \frac{\phi \xrightarrow{o}_\Gamma \phi' \; ; \; \emptyset}{\phi \xrightarrow{o}_\Gamma \phi'} \qquad\qquad \text{R-1} \quad \frac{}{\mathbf{1} \xrightarrow{o}_\Gamma \mathbf{1} \; ; \; \emptyset}$$

$$\text{R-PREFIX} \quad \frac{}{\tau : \phi \xrightarrow{o}_\Gamma \phi \; ; \; \sigma} \quad \Gamma \vdash \Gamma \; ; \; \tau : o \rightsquigarrow \sigma$$

$$\text{R-OR-L} \quad \frac{\phi_1 \xrightarrow{o}_\Gamma \phi_1' \; ; \; \sigma}{\phi_1 \vee \phi_2 \xrightarrow{o}_\Gamma \phi_1' \; ; \; \sigma} \qquad\qquad \text{R-OR-R} \quad \frac{\phi_1 \xrightarrow{o}_\Gamma \not\rightarrow \qquad \phi_2 \xrightarrow{o}_\Gamma \phi_2' \; ; \; \sigma}{\phi_1 \vee \phi_2 \xrightarrow{o}_\Gamma \phi_2' \; ; \; \sigma}$$

$$\text{R-AND} \quad \frac{\phi_1 \xrightarrow{o}_\Gamma \phi_1' \; ; \; \sigma_1 \qquad \phi_2 \xrightarrow{o}_\Gamma \phi_2' \; ; \; \sigma_2}{\phi_1 \wedge \phi_2 \xrightarrow{o}_\Gamma \phi_1' \wedge \phi_2' \; ; \; \sigma} \quad \sigma = \sigma_1 \cup \sigma_2$$

$$\text{R-SHUF-L} \quad \frac{\phi_1 \xrightarrow{o}_\Gamma \phi_1' \; ; \; \sigma}{\phi_1 \mid \phi_2 \xrightarrow{o}_\Gamma \phi_1' \mid \phi_2 \; ; \; \sigma} \qquad \text{R-SHUF-R} \quad \frac{\phi_1 \xrightarrow{o}_\Gamma \not\rightarrow \qquad \phi_2 \xrightarrow{o}_\Gamma \phi_2' \; ; \; \sigma}{\phi_1 \mid \phi_2 \xrightarrow{o}_\Gamma \phi_1 \mid \phi_2' \; ; \; \sigma}$$

$$\text{R-CAT-L} \quad \frac{\phi_1 \xrightarrow{o}_\Gamma \phi_1' \; ; \; \sigma}{\phi_1 \cdot \phi_2 \xrightarrow{o}_\Gamma \phi_1' \cdot \phi_2 \; ; \; \sigma} \qquad \text{R-CAT-R} \quad \frac{\phi_1 \xrightarrow{o}_\Gamma \not\rightarrow \qquad \phi_2 \xrightarrow{o}_\Gamma \phi_2' \; ; \; \sigma}{\phi_1 \cdot \phi_2 \xrightarrow{o}_\Gamma \phi_2' \; ; \; \sigma} \quad E(\phi_1)$$

$$\text{R-VAR-1} \quad \frac{\phi \xrightarrow{o}_\Gamma \phi' \; ; \; \sigma}{\{x ; \phi\} \xrightarrow{o}_\Gamma \sigma' \phi' \; ; \; \sigma \setminus \{x\}} \quad \begin{array}{l} x \in dom(\sigma) \\ \sigma' = \sigma \upharpoonright \{x\} \end{array}$$

$$\text{R-VAR-2} \quad \frac{\phi \xrightarrow{o}_\Gamma \phi' \; ; \; \sigma}{\{x ; \phi\} \xrightarrow{o}_\Gamma \{x ; \phi'\} \; ; \; \sigma} \quad x \notin dom(\sigma)$$

$$\text{R-GENERIC} \quad \frac{\sigma' \phi \xrightarrow{o}_\Gamma \phi' \; ; \; \sigma}{(\langle x_1, \dots, x_n \rangle . \phi) \langle \xi_1, \dots, \xi_n \rangle \xrightarrow{o}_\Gamma \phi' \; ; \; \sigma} \quad \sigma' = \{x_1 \mapsto ev(\xi_1), \dots, x_n \mapsto ev(\xi_n)\}$$

$$\text{R-IF-TRUE} \quad \frac{\phi_1 \xrightarrow{o}_\Gamma \phi_1' \; ; \; \sigma}{\text{if } \xi \text{ then } \phi_1 \text{ else } \phi_2 \xrightarrow{o}_\Gamma \phi_1' \; ; \; \sigma} \quad ev(\xi) = true$$

$$\text{R-IF-FALSE} \quad \frac{\phi_2 \xrightarrow{o}_\Gamma \phi_2' \; ; \; \sigma}{\text{if } \xi \text{ then } \phi_1 \text{ else } \phi_2 \xrightarrow{o}_\Gamma \phi_2' \; ; \; \sigma} \quad ev(\xi) = false$$

Figure 2.6: Inference system inductively defining TC rewriting $\phi \xrightarrow{o}_\Gamma \phi'$ and its auxiliary predicates.

$$\text{NR-EMPTY} \ \dfrac{}{\epsilon \not\xrightarrow[\Gamma]{o}} \qquad\qquad \text{NR-PREFIX} \ \dfrac{}{\tau : \phi \not\xrightarrow[\Gamma]{o}} \ \Gamma \vdash \Gamma \, ; \tau \,/\, o \qquad \text{NR-0} \ \dfrac{}{\mathbf{0} \not\xrightarrow[\Gamma]{o}}$$

$$\text{NR-OR} \ \dfrac{\phi_1 \not\xrightarrow[\Gamma]{o} \qquad \phi_2 \not\xrightarrow[\Gamma]{o}}{\phi_1 \vee \phi_2 \not\xrightarrow[\Gamma]{o}} \qquad\qquad \text{NR-AND-I} \ \dfrac{\phi_i \not\xrightarrow[\Gamma]{o}}{\phi_1 \wedge \phi_2 \not\xrightarrow[\Gamma]{o}} \ i \in \{1, 2\}$$

$$\text{NR-SHUFFLE} \ \dfrac{\phi_1 \not\xrightarrow[\Gamma]{o} \qquad \phi_2 \not\xrightarrow[\Gamma]{o}}{\phi_1 \mid \phi_2 \not\xrightarrow[\Gamma]{o}} \qquad\qquad \text{NR-VAR} \ \dfrac{\phi \not\xrightarrow[\Gamma]{o}}{\{x ; \phi\} \not\xrightarrow[\Gamma]{o}}$$

$$\text{NR-AND-MERGE} \ \dfrac{\phi_1 \xrightarrow[\Gamma]{o} \phi_1' \, ; \sigma_1 \qquad \phi_2 \xrightarrow[\Gamma]{o} \phi_2' \, ; \sigma_2}{\phi_1 \wedge \phi_2 \not\xrightarrow[\Gamma]{o}} \ \nexists \sigma . \sigma = \sigma_1 \cup \sigma_2$$

$$\text{NR-CAT-L} \ \dfrac{\phi_1 \not\xrightarrow[\Gamma]{o}}{\phi_1 \cdot \phi_2 \not\xrightarrow[\Gamma]{o}} \ NE(\phi_1) \qquad\qquad \text{NR-CAT-R} \ \dfrac{\phi_1 \not\xrightarrow[\Gamma]{o} \qquad \phi_2 \not\xrightarrow[\Gamma]{o}}{\phi_1 \cdot \phi_2 \not\xrightarrow[\Gamma]{o}}$$

$$\text{NR-IF-TRUE} \ \dfrac{\phi_1 \not\xrightarrow[\Gamma]{o}}{\text{if } \zeta \text{ then } \phi_1 \text{ else } \phi_2 \not\xrightarrow[\Gamma]{o}} \ ev(\zeta) = true$$

$$\text{NR-IF-FALSE} \ \dfrac{\phi_2 \not\xrightarrow[\Gamma]{o}}{\text{if } \zeta \text{ then } \phi_1 \text{ else } \phi_2 \not\xrightarrow[\Gamma]{o}} \ ev(\zeta) = false$$

$$\text{NR-GENERIC} \ \dfrac{\sigma \phi \not\xrightarrow[\Gamma]{o}}{(\langle x_1, \ldots, x_n \rangle . \phi) \langle \zeta_1, \ldots, \zeta_n \rangle \not\xrightarrow[\Gamma]{o}} \ \sigma = \{x_1 \mapsto ev(\zeta_1), \ldots, x_n \mapsto ev(\zeta_n)\}$$

Figure 2.7: Inference system inductively defining TC rewriting auxiliary predicate $\phi \not\xrightarrow[\Gamma]{o}$.

Shuffle is handled similarly to union (rules R-SHUF-L and R-SHUF-R), meaning the term on the left is given precedence. However, the shuffle encodes interleaving, rather than a choice point, thus both the terms must be preserved by the rewriting steps.

In concatenation, as expected, the left term can be chosen without further conditions (rule R-CAT-L). The right side, however, is slightly trickier (rule R-CAT-R): not only we can use it only if the left cannot be rewritten (with respect to the given event), but it is also requested that the left term accepts the empty trace. Intuitively, concatenation $\phi_1 \cdot \phi_2$ first consumes all the events expected from ϕ_1, and only after that, a trace satisfying ϕ_2 can be considered valid. Concatenation allows specifications to ensure an order of expected events.

Parametric specifications in the form $\{x; \phi\}$ are rewritten using rules R-VAR-1 and R-VAR-2. If the rewriting step produces a substitution for x, then such variable is instantiated and removed from the computed substitution (rule R-VAR-1); note that $\sigma \backslash \{x\}$ is produced, rather than the empty set, since a single rewriting step may instantiate more than one variable. Otherwise, if the variable is not mapped, the inner term ϕ is still rewritten but the variable scope block is mantained, so that no free, undeclared variables appear in the specification (rule R-VAR-2). This is necessary to correctly rewrite specifications where variables are instantiated some rewriting steps later.

When generic terms and their application are found (as it is forced by Definition 12), all the parameters are instantiated with the given evaluated arguments before proceeding with the rewriting (rule R-GENERIC).

The semantics of conditional specifications simply depends on the outcome of the evaluation of the condition (rules R-IF-TRUE and R-IF-FALSE). This means that such condition must not contain free variables at this point, otherwise no further rewriting steps can be done.

The negated predicate $\phi \overset{o}{\nrightarrow}$ rules are specular to those described above.

The reflexive and transitive closure of $\phi \overset{o}{\to} \phi'$ results in the trace semantics of TC specifications.

Definition 19. (TC semantics) The *semantics* of a TC term ϕ, with respect to a list of event type declarations Γ, is the set of traces $\llbracket \phi \rrbracket_\Gamma \in \wp(O^\infty)$ *coinductively* defined as follows:

$$\text{S-EMPTY} \ \frac{}{\epsilon \in \llbracket \phi \rrbracket_\Gamma} \ E(\phi) \qquad \text{S-STEP} \ \frac{t \in \llbracket \phi' \rrbracket_\Gamma}{ot \in \llbracket \phi \rrbracket_\Gamma} \ \phi \overset{o}{\underset{\Gamma}{\to}} \phi'$$

Event type declarations Γ will be left implicit when not relevant or clear from the context.

The formal semantics of a TC term belongs to the powerset of all traces. This means that it is a possibly infinite set of possibly infinite traces. Here infinite traces encode the behavior of systems for which non-termination is allowed, while an infinite set of traces means there are infinitely many valid behaviors for the monitored system. While the former is only relevant for non-terminating systems (like a web server or some safety-critical software), an infinitary semantics is much more common: it arises every time an operation can be repetead an arbitrary number of times, or when parametric specifications can be instantiated by an infinite number of values.

Example 8. (TC rewriting) Let us continue from Example 6 and use the event types declared there. The correct usage of a file is specified by the TC term ϕ, where ϕ is the regular term uniquely identified by the following equations:

$$\phi = \epsilon \vee \{fd; open(fd) : \phi'\}$$
$$\phi' = (rw(fd) : \phi') \vee (close(fd) : \epsilon)$$

According to ϕ, either no events are observed (ϵ) or the trace starts with an event matching $open(fd)$, for a given file descriptor fd that can be only known at runtime, followed by a trace verifying ϕ' ($open(fd) : \phi'$). Specification ϕ' admits two behaviors: on the left, a read or write operation is observed for the file identified by fd, and the verification process recursively returns to ϕ'; on the right, the file identified by fd is closed and no more events are observed.

As such, the empty trace belongs to the semantics of ϕ ($\epsilon \in [\![\phi]\!]$):

$$\text{S-empty} \ \frac{}{\epsilon \in [\![\epsilon \vee \{fd; open(fd) : \phi'\}]\!]} \ E(\epsilon \vee \{fd; open(fd) : \phi'\})$$

$$\text{E-or} \ \frac{\text{E-empty} \ \frac{}{E(\epsilon)}}{E(\epsilon \vee \{fd; open(fd) : \phi'\})} \ i = 1$$

Another, less trivial trace that is correct with respect to the same specification is $o_1 o_2 o_3$ ($\Gamma = d_1, \ldots, d_5$, see Example 6):

$$o_1 = \{\text{funcName: "}open\text{", args: ["}log.txt\text{"], result: } 42\}$$
$$o_2 = \{\text{funcName: "}write\text{", args: [}42\text{, "}Hello, world!\text{"]}\}$$
$$o_3 = \{\text{funcName: "}close\text{", args: [}42\text{], result: } 42\}$$

$$\text{S-step} \ \frac{\text{S-step} \ \frac{\text{S-step} \ \frac{\text{S-empty} \ \frac{}{\epsilon \in [\![\phi_3]\!]} \ E(\phi_3)}{o_3 \in [\![\phi_2]\!]} \ \phi_2 \xrightarrow{o_3} \phi_3}{o_2 o_3 \in [\![\phi_1]\!]} \ \phi_1 \xrightarrow{o_2} \phi_2}{o_1 o_2 o_3 \in [\![\epsilon \vee \{fd; open(fd) : \phi'\}]\!]} \ \epsilon \vee \{fd; open(fd) : \phi'\} \xrightarrow{o_1} \phi_1$$

$$\text{R-main} \ \frac{\text{R-or-r} \ \frac{\text{NR-empty} \ \frac{}{\epsilon \not\xrightarrow{o_1}} \quad \text{R-var-1} \ \frac{\text{R-prefix} \ \frac{}{open(fd) : \phi' \xrightarrow{o_1} \phi' ; \sigma} \ \frac{\Gamma \vdash \Gamma ; open(fd) : o_1 \rightsquigarrow \sigma}{\sigma = \{fd \mapsto 42\}}}{\{fd; open(fd) : \phi'\} \xrightarrow{o_1} \sigma\phi' ; \emptyset}}{\epsilon \vee \{fd; open(fd) : \phi'\} \xrightarrow{o_1} \sigma\phi' ; \emptyset}}{\epsilon \vee \{fd; open(fd) : \phi'\} \xrightarrow{o_1} \sigma\phi'}$$

The term ϕ_1 obtained after the first rewriting step is $\sigma\phi'$, that is:

$$\phi_1 = (rw(42) : \phi_1) \vee (close(42) : \epsilon)$$

The rewriting process then proceeds similarly for the rest of the trace.

It is worth noting that the purely coinductive approach of TC leads to the infinite trace $o_1 o_2 o_2 \cdots$ formally being a valid trace, though it cannot be observed in practice.

Rewriting-based semantics are quite common in runtime verification: a considerable amount of formalisms and tools employs process calculi-inspired

semantics, labeled transition system (of which our semantics can be seen as an example), and rule-based systems. These techniques are similar in many aspects, and though not investigated in this work, a comparison with TC semantics is an interesting future work direction. For relevant work with respect to this, see Aceto, Achilleos, Francalanza, and Ingólfsdóttir (2017, 2018), Aceto, Achilleos, Francalanza, Ingólfsdóttir, and Lehtinen (2019a,b), Francalanza (2016), and Francalanza, Aceto, and Ingólfsdóttir (2017).

2.6 ALGEBRAIC PROPERTIES

In this section we focus on some fundamental algebraic laws that are exploited for optimization purposes to make monitoring more efficient; we leave for future work the full development of the equational theory of TC.

Certain standard algebraic properties (with respect to the intuitive semantics of operators like union or shuffle) does *not* hold for TC semantics: commutativity, for instance, is violated due to the deterministic left-to-right semantics. Fortunately, there are still several algebraic properties relevant for optimization purposes that hold: identity and absorbing elements exist for concatenation, union, intersection, and shuffle.

Identity elements for TC binary operators, together with their associated laws, are listed below:

$$\llbracket \epsilon \cdot \phi \rrbracket = \llbracket \phi \cdot \epsilon \rrbracket = \llbracket \phi \rrbracket$$
$$\llbracket \mathbf{0} \vee \phi \rrbracket = \llbracket \phi \vee \mathbf{0} \rrbracket = \llbracket \phi \rrbracket$$
$$\llbracket \mathbf{1} \wedge \phi \rrbracket = \llbracket \phi \wedge \mathbf{1} \rrbracket = \llbracket \phi \rrbracket$$
$$\llbracket \epsilon \mid \phi \rrbracket = \llbracket \phi \mid \epsilon \rrbracket = \llbracket \phi \rrbracket$$

Proofs of laws for identity elements

- $\llbracket \epsilon \cdot \phi \rrbracket = \llbracket \phi \cdot \epsilon \rrbracket = \llbracket \phi \rrbracket$: by rule E-EMPTY and NR-EMPTY $E(\epsilon)$ and $\epsilon \overset{o}{\not\rightarrow}$ are derivable, and by the rewriting rules, $\epsilon \overset{o}{\rightarrow} \phi' ; \sigma$ is not derivable for any o, ϕ', σ; therefore from rule E-BIN we deduce that $E(\epsilon \cdot \phi)$ is derivable iff $E(\phi)$ is derivable iff $E(\phi \cdot \epsilon)$ is derivable.

  From the facts above, for $\epsilon \cdot \phi$ the only applicable rule is R-CAT-R and directly from that rule we deduce that $\epsilon \cdot \phi \overset{o}{\rightarrow} \phi' ; \sigma$ iff $\phi \overset{o}{\rightarrow} \phi' ; \sigma$, while for $\phi \cdot \epsilon$ the only applicable rule is R-CAT-L and directly from that rule we deduce that $\phi \cdot \epsilon \overset{o}{\rightarrow} \phi' \cdot \epsilon ; \sigma$ iff $\phi \overset{o}{\rightarrow} \phi' ; \sigma$.

  Finally, from the facts above, from Def. 19 and rule R-MAIN we have:

  - $\epsilon \in \llbracket \epsilon \cdot \phi \rrbracket$ iff $\epsilon \in \llbracket \phi \rrbracket$ iff $\epsilon \in \llbracket \phi \cdot \epsilon \rrbracket$;
  - $ot \in \llbracket \epsilon \cdot \phi \rrbracket$ iff $ot \in \llbracket \phi \rrbracket$;
  - $ot \in \llbracket \phi \cdot \epsilon \rrbracket$ iff $ot \in \llbracket \phi \rrbracket$ by coinduction.

  Therefore we can conclude $\llbracket \epsilon \cdot \phi \rrbracket = \llbracket \phi \cdot \epsilon \rrbracket = \llbracket \phi \rrbracket$.

- $\llbracket \epsilon \mid \phi \rrbracket = \llbracket \phi \mid \epsilon \rrbracket = \llbracket \phi \rrbracket$: similarly as the previous case, we deduce that $\epsilon \overset{o}{\not\rightarrow}$ is derivable, $\epsilon \overset{o}{\rightarrow} \phi' ; \sigma$ is not derivable for any o, ϕ', σ, and $E(\epsilon \mid \phi)$ is derivable iff $E(\phi)$ is derivable iff $E(\phi \mid \epsilon)$ is derivable.

  From the facts above, for $\epsilon \mid \phi$ the only applicable rule is R-SHUF-R and directly from that rule we deduce that $\epsilon \mid \phi \overset{o}{\rightarrow} \epsilon \mid \phi' ; \sigma$ iff $\phi \overset{o}{\rightarrow} \phi' ; \sigma$, while for $\phi \mid \epsilon$ the only applicable rule is R-SHUF-L and directly from that rule we deduce that $\phi \mid \epsilon \overset{o}{\rightarrow} \phi' \mid \epsilon ; \sigma$ iff $\phi \overset{o}{\rightarrow} \phi' ; \sigma$.

  Finally, from the facts above, from Def. 19 and rule R-MAIN we have:

     – $\epsilon \in [\![\epsilon \mid \phi]\!]$ iff $\epsilon \in [\![\phi]\!]$ iff $\epsilon \in [\![\phi \mid \epsilon]\!]$;

     – $ot \in [\![\epsilon \mid \phi]\!]$ iff $ot \in [\![\phi]\!]$ iff $ot \in [\![\phi \mid \epsilon]\!]$ by coinduction.

Therefore we can conclude $[\![\epsilon \mid \phi]\!] = [\![\phi \mid \epsilon]\!] = [\![\phi]\!]$.

- $[\![\mathbf{0} \vee \phi]\!] = [\![\phi \vee \mathbf{0}]\!] = [\![\phi]\!]$: by the rules for $E(-)$ we know that $E(\mathbf{0})$ is not derivable, while by rule NR-0 $\mathbf{0} \overset{o}{\not\to}$ is derivable, and by the rewriting rules, $\mathbf{0} \overset{o}{\to} \phi'$; σ is not derivable for any o, ϕ', σ; therefore from rule E-OR we deduce that $E(\mathbf{0} \vee \phi)$ is derivable iff $E(\phi)$ is derivable iff $E(\phi \vee \mathbf{0})$ is derivable.

  From the facts above, for $\mathbf{0} \vee \phi$ the only applicable rule is R-OR-R and directly from that rule we deduce that $\mathbf{0} \vee \phi \overset{o}{\to} \phi'$; σ iff $\phi \overset{o}{\to} \phi'$; σ, while for $\phi \vee \mathbf{0}$ the only applicable rule is R-OR-L and directly from that rule we deduce that $\phi \vee \mathbf{0} \overset{o}{\to} \phi'$; σ iff $\phi \overset{o}{\to} \phi'$; σ.

  Finally, from the facts above, from Def. 19 and rule R-MAIN we have:

       – $\epsilon \in [\![\mathbf{0} \vee \phi]\!]$ iff $\epsilon \in [\![\phi]\!]$ iff $\epsilon \in [\![\phi \vee \mathbf{0}]\!]$;

       – $ot \in [\![\mathbf{0} \vee \phi]\!]$ iff $ot \in [\![\phi]\!]$ iff $ot \in [\![\phi \vee \mathbf{0}]\!]$.

  Therefore we can conclude $[\![\mathbf{0} \vee \phi]\!] = [\![\phi \vee \mathbf{0}]\!] = [\![\phi]\!]$.

- $[\![\mathbf{1} \wedge \phi]\!] = [\![\phi \wedge \mathbf{1}]\!] = [\![\phi]\!]$: by rules E-1 and R-1 we know that $E(\mathbf{1})$ and $\mathbf{1} \overset{o}{\to} \mathbf{1}$; $\emptyset$ are derivable for any o; therefore from rule E-BIN we deduce that $E(\mathbf{1} \wedge \phi)$ is derivable iff $E(\phi)$ is derivable iff $E(\phi \wedge \mathbf{1})$ is derivable.

  From the facts above, from rule R-AND and from the fact that $\sigma = \emptyset \cup \sigma = \sigma \cup \emptyset$ we deduce that $\mathbf{1} \wedge \phi \overset{o}{\to} \mathbf{1} \wedge \phi'$; σ iff $\phi \overset{o}{\to} \phi'$; σ iff $\phi \wedge \mathbf{1} \overset{o}{\to} \phi' \wedge \mathbf{1}$; σ.

  Finally, from the facts above, from Def. 19 and rule R-MAIN we have:

       – $\epsilon \in [\![\mathbf{1} \wedge \phi]\!]$ iff $\epsilon \in [\![\phi]\!]$ iff $\epsilon \in [\![\phi \wedge \mathbf{1}]\!]$;

       – $ot \in [\![\mathbf{1} \wedge \phi]\!]$ iff $ot \in [\![\phi]\!]$ iff $ot \in [\![\phi \wedge \mathbf{1}]\!]$, by coinduction.

  Therefore we can conclude $[\![\mathbf{1} \wedge \phi]\!] = [\![\phi \wedge \mathbf{1}]\!] = [\![\phi]\!]$.

While identity laws work as expected, laws for absorbing elements are less intuitive because of determinism and infinite traces.

$$[\![\mathbf{0} \cdot \phi]\!] = [\![\mathbf{0}]\!]$$
$$[\![\mathbf{1} \vee \phi]\!] = [\![\mathbf{1}]\!]$$
$$[\![\mathbf{0} \wedge \phi]\!] = [\![\phi \wedge \mathbf{0}]\!] = [\![\mathbf{0}]\!]$$

Element $\mathbf{0}$ is left-absorbing for concatenation (see the proof below), but is not right-absorbing; consider for instance the term $\mathbf{1} \cdot \mathbf{0}$: all infinite traces belong to the semantics of this specification; indeed, by rules R-CAT-L and R-1 we have that $\mathbf{1} \cdot \mathbf{0} \overset{o}{\to} \mathbf{1} \cdot \mathbf{0}$; $\emptyset$ is derivable for any o, therefore $[\![\mathbf{1} \cdot \mathbf{0}]\!] = O^\omega \neq [\![\mathbf{0}]\!]$.

Analogously, element $\mathbf{1}$ is left-absorbing for union (see the proof below), but is not right-absorbing; consider for instance the term $(\tau : \epsilon) \vee \mathbf{1}$, where τ is an event type s.t. there exist o and σ s.t. $\Gamma \vdash \Gamma$; $\tau : o \rightsquigarrow \sigma$ is derivable; then, from rules R-PREFIX and R-OR-L we have that $(\tau : \epsilon) \vee \mathbf{1} \overset{o}{\to} \epsilon$; σ is derivable; furthermore, since $\tau : \epsilon \overset{o}{\not\to}$ is not derivable, rule R-OR-R is not applicable, therefore the only trace beginning with o which belongs to $[\![(\tau : \epsilon) \vee \mathbf{1}]\!]$ has length 1, hence $[\![(\tau : \epsilon) \vee \mathbf{1}]\!] \subset [\![\mathbf{1}]\!]$.

Because the only rule for intersection requires a rewriting step for both sides, $\mathbf{0}$ is both left- and right-absorbing for intersection.

Shuffle does not have any absorbing element. Element $\mathbf{0}$ does not satisfy the property, as both $\mathbf{0} \mid \phi$ and $\phi \mid \mathbf{0}$ accept infinite traces from $[\![\phi]\!]$, if any. Also $\mathbf{1}$ cannot be such element: both $\mathbf{1} \mid (\tau : \epsilon)$ and $(\tau : \epsilon) \mid \mathbf{1}$ do not include the empty trace.

Proofs of laws for absorbing elements

- $[\![\mathbf{0} \cdot \phi]\!] = [\![\mathbf{0}]\!]$: it suffices to prove that $[\![\mathbf{0} \cdot \phi]\!] = \emptyset$. By the rules for $E(\_)$ we know that $E(\mathbf{0})$ is not derivable, and by the rewriting rules, $\mathbf{0} \xrightarrow{o} \phi'$; σ is not derivable for any o, ϕ', σ; therefore $E(\mathbf{0} \cdot \phi)$ is not derivable because rule E-BIN is not applicable, and there is no ϕ', o and σ s.t. $\mathbf{0} \cdot \phi \xrightarrow{o} \phi'$; σ is derivable because rules R-CAT-L and R-CAT-R are not applicable, hence $[\![\mathbf{0} \cdot \phi]\!] = \emptyset$.

- $[\![\mathbf{1} \vee \phi]\!] = [\![\mathbf{1}]\!]$: by rules E-1 and R-1 we know that $E(\mathbf{1})$ and $\mathbf{1} \xrightarrow{o} \mathbf{1}$; $\emptyset$ are derivable for any o; therefore from rule E-OR we deduce that $E(\mathbf{1} \vee \phi)$ is derivable, hence $\epsilon \in [\![\mathbf{1} \vee \phi]\!]$. Furthermore, by rule R-OR-L $\mathbf{1} \vee \phi \xrightarrow{o} \mathbf{1}$; $\emptyset$ is derivable for any o, therefore we conclude $[\![\mathbf{1} \vee \phi]\!] = [\![\mathbf{1}]\!]$ by Def. 19.

- $[\![\mathbf{0} \wedge \phi]\!] = [\![\phi \wedge \mathbf{0}]\!] = [\![\mathbf{0}]\!]$: it suffices to prove that $[\![\mathbf{0} \wedge \phi]\!] = \emptyset$ (the proof for $[\![\phi \wedge \mathbf{0}]\!] = \emptyset$ is the same). By the rules for $E(\_)$ we know that $E(\mathbf{0})$ is not derivable, and by the rewriting rules, $\mathbf{0} \xrightarrow{o} \phi'$; σ is not derivable for any o, ϕ', σ; therefore $E(\mathbf{0} \wedge \phi)$ is not derivable because rule E-BIN is not applicable, and there is no ϕ', o and σ s.t. $\mathbf{0} \wedge \phi \xrightarrow{o} \phi'$; σ is derivable because rules R-AND is not applicable, hence $[\![\mathbf{0} \wedge \phi]\!] = \emptyset$.

Some expected properties can be restored if we restrict the semantics to finite traces. Assuming such restriction to be denoted as $[\![\phi]\!]^* = [\![\phi]\!] \cap O^*$, the following additional laws can be stated, according to the reasoning above regarding finite and infinite traces:

$$[\![\mathbf{0} \cdot \phi]\!]^* = [\![\phi \cdot \mathbf{0}]\!]^* = [\![\mathbf{0}]\!]^*$$
$$[\![\mathbf{0} \mid \phi]\!]^* = [\![\phi \mid \mathbf{0}]\!]^* = [\![\mathbf{0}]\!]^*$$

For finite traces, laws above hold because, after a finite number of steps, the semantics will either require $\mathbf{0}$ to be rewritten or $E(\mathbf{0})$ to hold, but none of them is valid. For all laws it is straightforward to prove that no finite trace t can belong to the semantics of the terms by induction on the length of t.

The main reason why identity and absorbing elements are very relevant to the proposed semantics is the ability to *optimize*, a possibility that will be exploited when discussing examples and performances in the next chapters. Some recursive TC terms can grow in size with rewriting steps, especially when mixing recursion with operators like intersection and shuffle, which keep both sides after rewriting. In this cases, the specification would grow indefinitely, possibly making the monitor run out of memory or become quite inefficient, regardless of the correctness of the observed system.

The optimizations above allow the implementation to *shrink* the TC term during the process whenever possible, sometimes changing a process that is bound to failure to a sustainable one, reducing the computational complexity of the verification algorithm. Our implementation indeed exploits such laws; this topic is further developed in Chapter 6.

2.7 PROOF OF DETERMINISM

The rewriting semantics presented for TC in Definition 18 is designed to be deterministic. Determinism can greatly improve monitoring performance,

since there is no need to implement back-tracking mechanism or to keep a set of all the terms the specification can be rewritten to. The problem of monitor determinization in runtime verification has been extensively studied in the literature (Aceto, Achilleos, Francalanza, Ingólfsdóttir, and Kjartansson, 2017; Francalanza, 2016, 2017). In the context of RML, while non-determinism of the monitored system is allowed (and can be expressed in TC) the monitoring procedure itself (i. e., the rewriting system) is always deterministic.

This Section is devoted to the proof of such claim. Hence, the goal is to prove that if $\phi \xrightarrow{o} \phi'$; σ' and $\phi \xrightarrow{o} \phi''$; σ'', then $\phi' = \phi''$ and $\sigma' = \sigma''$. The semantics is stratified, and determinism needs to be proven at each level. Furthermore, we need to prove that judgements with their corresponding negated version cannot be derived simultaneously.

Lemma 1. *If $p : e \leadsto \sigma$ holds, then $p \nmid e$ does not hold.*

Proof. The proof goes by induction on the inference rules defining $p : e \leadsto \sigma$ (Figure 2.1).

- Rule PM-obj: by inductive hypotheses, for all $i \in \{1, \dots, n\}$, $p_i \nmid e_i$ does not hold, and from the side condition we also know $\sigma = \sigma_1 \cup \cdots \cup \sigma_n$. The only possibly applicable rules for $p \nmid e$ are PMF-obj-other and PMF-obj. The first one is not applicable due to the side condition. The second one relies on the premise $[p_1, \dots, p_n] \nmid [e_1, \dots, e_n]$, but that can neither be proved from rule PMF-list-merge (because we know $\sigma_1, \dots, \sigma_n$ are mergeable into σ) nor from rule PMF-list-elem (by the initial inductive hypothesis).

- For rule PM-list the same reasoning as above holds.

- The lemma trivially holds for wildcards and variables (rules PM-wildcard and PM-var), as no failed pattern matching is defined on them.

- Rules PM-left and PM-right: the inductive hypothesis on the premises implies that either $p_1 \nmid e$ or $p_2 \nmid e$ cannot hold, therefore $p_1 \mid p_2 \nmid e$ cannot be proved (rule PMF-choice).

- The lemma trivially holds for event expressions.

□

Lemma 2. *Successful event pattern matching is deterministic, that is, if $p : e \leadsto \sigma$ and $p : e \leadsto \sigma'$, then $\sigma = \sigma'$.*

Proof. By induction on the inference rules defining $p : e \leadsto \sigma$ (Figure 2.1).

- Rules PM-obj and PM-list: by inductive hypotheses, all substitutions $\sigma_1, \dots, \sigma_n$ are uniquely determined. From the side conditions it comes that all substitutions agree on the same variables (if any), otherwise substitution merge would not be defined, therefore σ is the only function whose domain $dom(\sigma_1) \cup \cdots \cup dom(\sigma_n)$ mapping those variables according to the single substitutions.

- Rules PM-list-more: the lemma holds by straightforward induction, as the substitution is the same as in the premise.

- Rules PM-wildcard, PM-var, and PM-exp are axioms and there is only one possible substitution.

- Rules PM-LEFT and PM-RIGHT: by Lemma 1, either $p_1 : e \leadsto \sigma$ or $p_1 \nmid e$ hold, but not both, thus only one of the rules is applicable. In both cases, the lemma holds by inductive hypothesis.

□

Lemma 3. *If $\Gamma \vdash d_1, \dots, d_n \mathbin{;} \tau \nmid e$ holds, then $\Gamma \vdash d_1, \dots, d_{i-1}, d_{i+1}, \dots, d_n \mathbin{;} \tau \nmid e$ holds as well.*

Proof. Straightforward induction on the rules in Figure 2.2 part 2.   □

Lemma 4. *If $\Gamma \vdash d_1, \dots, d_n \mathbin{;} \tau : e \leadsto \sigma''$, then $\Gamma \vdash d_1, \dots, d_n \mathbin{;} \tau \nmid e$ does not hold.*

Proof. By induction on the inference rules defining $\Gamma \vdash d_1, \dots, d_n \mathbin{;} \tau : e \leadsto \sigma''$.

- Rule ETM-STEP: by inductive hypothesis $\Gamma \vdash d_2, \dots, d_n \mathbin{;} \tau \nmid e$ does not hold, therefore by lemma 3 $\Gamma \vdash d_1, \dots, d_n \mathbin{;} \tau \nmid e$ does not hold as well.

- Rule ETM-DIRECT: from the side condition $\sigma p : e \leadsto \sigma'$, then by Lemma 1 $\sigma p \nmid e$ does not hold, therefore the only usable rule ETFM-DIRECT is not applicable.

- The same reasoning as above also holds for rule ETM-DIRECT-NOT.

- Rule ETM-DERIVED: from the inductive hypotheses, $\Gamma \vdash \Gamma \mathbin{;} \sigma \tau_j \nmid e$ does not hold for $1 \le j \le q$, therefore the only usable rule ETFM-DERIVED to derive $\Gamma \vdash d_1, \dots, d_n \mathbin{;} \tau \nmid e$ cannot be applied.

- Rule ETM-DERIVED-NOT: from the premises, all $\sigma \tau_j$ do not match e according to any declaration. Hence, it is not possible to apply the only usable rule to prove $\Gamma \vdash d_1, \dots, d_n \mathbin{;} \tau \nmid e$, namely ETFM-DERIVED-NOT.

□

Lemma 5. *Event type matching is deterministic, that is, if $\Gamma \vdash d_1, \dots, d_n \mathbin{;} \tau : e \leadsto \sigma$ and $\Gamma \vdash d_1, \dots, d_n \mathbin{;} \tau : e \leadsto \sigma'$, then $\sigma = \sigma'$.*

Proof. The proof goes by induction on the inference rules defining $\Gamma \vdash d_1, \dots, d_n \mathbin{;} \tau : e \leadsto \sigma$. The (only) rule applicable depends on the shape of d_1 (see the side conditions in Figure 2.2).

- Rule ETM-STEP: from the hypothesis $\Gamma \vdash d_1 \mathbin{;} \tau \nmid e$ and lemma 4 we have that $\Gamma \vdash d_1 \mathbin{;} \tau : e \leadsto \sigma''$ cannot hold for any substitution σ'', therefore the only applicable rule for $\Gamma \vdash d_1, \dots, d_n \mathbin{;} \tau : e \leadsto \sigma'$ is ETM-STEP; we can then conclude by applying the inductive hypothesis to the other hypothesis $\Gamma \vdash d_2, \dots, d_n \mathbin{;} \tau : e \leadsto \sigma$.

- Rule ETM-DIRECT: the resulting substitution is computed by the pattern matching side condition, which is deterministic by Lemma 2.

- Rules ETM-DIRECT-NOT and ETM-DERIVED-NOT: the produced substitution is always $\emptyset$.

- Rule ETM-DERIVED: by applying the same rule with the same hypotheses we can derive $\Gamma \vdash d_1 \mathbin{;} \tau : e \leadsto \sigma$, hence by Lemma 4 $\Gamma \vdash d_1 \mathbin{;} \tau \nmid e$ does not hold, therefore rule ETM-STEP is not applicable; this means that the only applicable rule for $\Gamma \vdash d_1, \dots, d_n \mathbin{;} \tau : e \leadsto \sigma'$ is ETM-DERIVED; we can then conclude by applying the inductive hypothesis to the hypothesis $\Gamma \vdash \Gamma \mathbin{;} \sigma'' \tau_j : e \leadsto \sigma$.

□

Lemma 6. *If $E(\phi)$, then $NE(\phi)$ does not hold.*

Proof. By induction on the inference rules defining $E(-)$ (see Figure 2.4 and 2.5 for $E(-)$ and $NE(-)$, respectively), and case analysis on the structure of the term.

- Rules E-EMPTY and E-1: the opposite judgement is not derivable as no rule consequence matches $NE(\epsilon)$ or $NE(\mathbf{1})$.

- Rule E-BIN: by inductive hypothesis, neither $NE(\phi_1)$ nor $NE(\phi_2)$ hold, hence rule NE-BIN cannot be applied.

- Rule E-OR: $NE(\phi_i)$ does not hold by inductive hypothesis, for some $i \in \{1, 2\}$, thus rule NE-OR cannot be applied.

- The lemma holds for the other rules by straightforward induction.

□

Note that the lemma above does not imply that no ϕ exists such that neither judgements holds. An if-then-else term with an expression that does not evaluate neither to *true* nor to *false* (because of unbound variables, for instance) is an example of such a TC term.

Lemma 7. *If $\phi \xrightarrow[\Gamma]{o} \phi' \; ; \sigma$, then $\phi \xnrightarrow[\Gamma]{o}$ does not hold.*

Proof. By induction on the rules defining $\phi \xrightarrow[\Gamma]{o} \phi' \; ; \sigma$ in Figure 2.6 (rules for the negated judgement are in Figure 2.7).

- Rule R-1: no rule exists to derive $\mathbf{1} \xnrightarrow{o}$.

- Rule R-PREFIX: by application of Lemma 4 to the side condition $\Gamma \vdash \Gamma \; ; \tau : o \rightsquigarrow \sigma$, it comes that $\Gamma \vdash \Gamma \; ; \tau \nmid o$. Rule NR-PREFIX cannot be applied then.

- Rules R-OR-L and R-OR-R: by inductive hypothesis, $\phi_i \xnrightarrow{o}$ for some $i \in \{1, 2\}$. By rule NR-OR, $\phi \xnrightarrow{o}$ does not hold.

- The same reasoning as above holds for rules R-SHUF-L, R-SHUF-R, and R-CAT-L, by considering the appropriate rules for $\phi \xnrightarrow{o}$.

- Rule R-CAT-R: the rules for $\phi_1 \cdot \phi_2 \xnrightarrow{o}$ are NR-CAT-L and NR-CAT-R. The former requires $NE(\phi_1)$, but from rule R-CAT-R we know $E(\phi_1)$, and by Lemma 6, $NE(\phi_1)$ cannot hold. The latter requires $\phi_2 \xnrightarrow{o}$ as a premise, but by inductive hypothesis it does not hold.

- For all the other rules the lemma comes from the inductive hypotheses and inspection of the applicable rules in Figure 2.7.

□

Lemma 8. *Data expression evaluation is deterministic, that is, if $ev(\xi) = r$ and $ev(\xi) = r'$, then $r = r'$.*

Proof. The lemma is a direct consequence of Definition 14, which uses deterministic mathematical operators. □

Theorem 1. (*TC rewriting determinism*) *If $\phi \xrightarrow[\Gamma]{o} \phi_1 \; ; \sigma_1$ and $\phi \xrightarrow[\Gamma]{o} \phi_2 \; ; \sigma_2$, then $\phi_1 = \phi_2$ and $\sigma_1 = \sigma_2$.*

Proof. The proof proceeds by induction on the rules defining $\phi \xrightarrow{o}_{\Gamma} \phi_1 \; ; \sigma_1$ in Figure 2.6.

- Rule R-1: **1** can only rewrite in itself with the empty substitution, thus $\phi_1 = \phi_2 = \mathbf{1}$ and $\sigma_1 = \sigma_2 = \emptyset$.

- Rule R-PREFIX: $\phi = \tau : \phi'$. From the rule structure, $\phi_1 = \phi_2 = \phi'$, and by Lemma 5 $\sigma_1 = \sigma_2$.

- Rule R-OR-L: $\phi = \phi' \vee \phi''$. From the premise, $\phi' \xrightarrow{o} \phi_1 \; ; \sigma_1$ and $\phi' \xrightarrow{o} \phi_2 \; ; \sigma_2$, and by Lemma 7, ϕ can only be rewritten using R-OR-L. By inductive hypothesis $\phi_1 = \phi_2$ and $\sigma_1 = \sigma_2$. Similar reasoning holds for rule R-OR-R and for the other binary operator rules.

- Rule R-AND: $\phi = \phi' \wedge \phi''$. By inductive hypothesis on the premises and by definition of substitution merge (Section 2.2.2).

- Rules R-VAR-1 and R-VAR-2: from the premise of the rule, $\phi = \{x; \phi'\}$ and $\phi' \xrightarrow{o} \phi'' \; ; \sigma'$. By inductive hypothesis, ϕ' can only rewrite to ϕ'' with substitution σ', so also $\phi \xrightarrow{o} \phi_2 \; ; \sigma_2$ comes from the same premise. Since there is only one substitution possible for the premises of both rules, and the side conditions $x \in dom(\sigma')$ and $x \notin dom(\sigma')$ of the two rules are incompatible, we deduce that the same rule has been applied, and the conclusion directly follows from the inductive hypothesis.

- Rules R-IF-TRUE and R-IF-FALSE: by inductive hypothesis and Lemma 8.

- Rule R-GENERIC: since the substitution instantiating generic parameters is unique and only depends on ϕ, the claim holds by applying the inductive hypothesis.

$\square$

RML

This Chapter introduces RML, a high-level, system-agnostic specification language. RML provides more operators than TC, though the additional ones can be understood as derived operators from an expressivity point of view. While the aim of TC is to be a small foundational formalism, RML is meant to be a user-friendly higher level language.

3.1 SYNTAX

The syntax of RML is made of standard inductive terms, no regular terms are involved. Since RML has to be implemented directly, it is easier to work with structures that can be translated to standard abstract syntax trees. Recursion is encoded through syntactic definition, though this time they are explicitly represented.

Definition 20. (RML syntax) Given a set of identifiers $\mathcal{I}$ and variables $\mathcal{X}$, the syntax of an *RML specification Spec* is defined according to the following context-free grammar:

$$
\begin{array}{llll}
Spec & ::= & \mathcal{D}; \cdots ; \mathcal{D}; \quad \text{main} = Exp; \quad Def; \cdots ; Def; & \text{(specification)} \\
Def & ::= & \mathcal{I}\langle \mathcal{X}, \dots, \mathcal{X} \rangle = Exp & \text{(definition)} \\
Exp & ::= & \text{empty} & \text{(empty)} \\
& | & \text{all} & \text{(all)} \\
& | & \text{none} & \text{(none)} \\
& | & \mathcal{T} & \text{(event type)} \\
& | & \mathcal{I}\langle \Xi, \dots, \Xi \rangle & \text{(recursion)} \\
& | & Exp\ Exp & \text{(concatenation)} \\
& | & Exp \wedge Exp & \text{(intersection)} \\
& | & Exp \vee Exp & \text{(union)} \\
& | & Exp\ |\ Exp & \text{(shuffle)} \\
& | & \{\mathcal{X}; Exp\} & \text{(parametric)} \\
& | & Exp! & \text{(prefix closure)} \\
& | & Exp? & \text{(optional)} \\
& | & Exp+ & \text{(repeat)} \\
& | & Exp* & \text{(Kleene star)} \\
& | & \mathcal{T} \gg Exp & \text{(single filter)} \\
& | & \mathcal{T} \gg Exp\ ; Exp & \text{(double filter)} \\
& | & \text{if } \Xi \text{ then } Exp \text{ else } Exp & \text{(conditional)}
\end{array}
$$

When there is no generic parameter/argument, angular brackets $\langle \rangle$ can be omitted altogether, both at definition and use sites. Furthermore, unproductive definitions of shape $id\langle x_1, \dots, x_n \rangle = id'\langle \xi_1, \dots, \xi_m \rangle$ are not allowed.

An RML specification has the following components:

- a list of event type declarations $d_1, \dots, d_n$ describing the shape of the events that are relevant to the specification;

- a definition main = *Exp* working as the entry point of the verification procedure;

- a list of more definitions $Def_1; \cdots ; Def_m$; to describe the rest of the specifications, if more than one definition is needed.

Note that RML generics work at the definition level: rather than making arbitrary expression generic, only single definitions can have generic parameters, and they are instantiated when a definition refers to another one (or to itself).

RML provides three *constants*. Constant empty is only verified by the empty trace, and says no more events are expected, just like TC ϵ. Constant all can be understood as the set of all traces, and it is used to describe a condition where any behavior is accepted from there on (for instance because the property being verified already holds). Finally, none is dual to all and signals an error, regardless of more events being observed or not (this is the case where a property has been violated).

TC makes a clear distinction between concatenation and prefix, as the latter is where the event matching happens, and it is the base case of TC semantics (Definition 18). Such distinction is however more relevant for the development of a formal semantics, rather than for the final user. In RML there is only one operator, *concatenation*, and event type patterns are simply expressions. The translation to TC will take care of this. RML concatenation is syntactically represented by juxtaposition.

Exp! encodes *prefix closure*. It is sometimes useful to specify a set of traces, and then to accept any possible prefix of any of those traces, rather than explicitly account for termination at every step. Let us consider, for instance, a specification of a data structure like a stack or a queue. A stricter specification could force the program to empty the data structure before terminating, while a more relaxed one would only ensure observed operations return the correct values, letting the program the possibility to terminate at any time. Following this example, by applying the prefix closure operator to the stricter specification we would obtain the more relaxed semantics.

Regular expressions are a standard tool in runtime verification, and their standard operators are often useful and well-understood by developers and computer scientists. RML supports typical regular expressions operators like *Exp*?, *Exp*+, and *Exp*∗ (beyond concatenation and union). Their semantics is the standard one, augmented with infinite traces: *Exp*+ and *Exp*∗ allows both finite and infinite concatenations of traces that are correct with respect to *Exp*.

When writing specifications in a modular way, it is often the case that different parts of the specification verify different properties, and they are then combined with appropriate operators to get the desired global property. When this happens, not all the events observed by the instrumentation are relevant for all those parts, and the approach fails to be truly modular and quickly leads to cumbersome specifications. The *filter* operators serve this exact purpose: $\tau \gg Exp$ ignores all the events not matching τ, and check those matching it against *Exp*. The extended version ($\tau \gg Exp ; Exp'$) verifies also events not matching τ, but against Exp', rather than ignoring them.

The other operators have the same syntax and semantics of TC, namely: intersection, union, shuffle, parametric expressions, and if-then-else.

3.2 PREFIX CLOSURE

Most of RML operators can be expressed in terms of TC operators in a compositional way; unfortunately, the prefix closure described in the previous section is an exception.

We recall that prefix closure adds the possibility of terminating after every step, or, in other words, its semantics is the set of all prefixes of all traces in the original semantics (hence the name). Such an operation will now be defined on TC terms, and it will be used when formalizing the semantics of RML.

Definition 21. (Prefix closure) Given a TC term ϕ, its prefix closure $pc(\phi)$ is coinductively defined by the following equations:

$$pc(\epsilon) = \epsilon$$
$$pc(\mathbf{0}) = \epsilon$$
$$pc(\mathbf{1}) = \mathbf{1}$$
$$pc(\tau : \phi) = \epsilon \vee (\tau : pc(\phi))$$
$$pc(\phi_1 \cdot \phi_2) = pc(\phi_1) \vee (\phi_1 \cdot pc(\phi_2))$$
$$pc(\phi_1 \vee \phi_2) = pc(\phi_1) \vee pc(\phi_2)$$
$$pc(\phi_1 \wedge \phi_2) = pc(\phi_1) \wedge pc(\phi_2)$$
$$pc(\phi_1 \mid \phi_2) = pc(\phi_1) \mid pc(\phi_2)$$
$$pc(\{x; \phi\}) = \{x; pc(\phi)\}$$
$$pc(g\langle \xi_1, \dots, \xi_n \rangle) = pc(g)\langle \xi_1, \dots, \xi_n \rangle$$
$$pc(\langle x_1, \dots, x_n \rangle.\phi) = \langle x_1, \dots, x_n \rangle.pc(\phi)$$
$$pc(\text{if } \xi \text{ then } \phi_1 \text{ else } \phi_2) = \text{if } \xi \text{ then } pc(\phi_1) \text{ else } pc(\phi_2)$$

Since no traces are valid with respect to $\mathbf{0}$, we only need to add the empty one, therefore $pc(\mathbf{0}) = \epsilon$. ϵ and $\mathbf{1}$ already accepts the empty trace, so nothing needs to be changed.

Prefix operator expects at least one event, so we need to explicitly allow termination by adding ϵ. Concatenation, on the other hand, is less intuitive. If we simply apply the transformation to both branches, then the resulting specification could accept a prefix of a trace in $[\![\phi_1]\!]$ followed by a prefix of a trace in $[\![\phi_2]\!]$; this however is *not* a prefix of a trace belonging to $[\![\phi_1 \cdot \phi_2]\!]$.

The transformation is entirely compositional on all the other operators and strictly follows the structure of the specification.

Example 9. (Stack prefix closure) Example 7 shows a TC specification ϕ for a stack. One limitation of that specification is that it expects the data structure to be empty at the end of monitoring. While this can be sensible in some cases, it could also make sense to relax the specification and allow elements to be in the structure at the end of the program. Rather than changing the specification, we can simply consider $pc(\phi)$ instead, and get the desired behavior with minimal effort.

3.3 TRANSLATION SEMANTICS

Since RML is defined at a higher-level w.r.t. TC, and inspired by it, its semantics is given by *translation*: an RML specification is compiled down to a TC term, and the specification semantics (as a set of traces) is that of such a term.

Definition 22. (RML compilation) Given a list of RML definitions $Def_1, \dots, Def_n$, an RML expression *Exp* is translated to a TC term ϕ according to the coinductive interpretation of judgement $Def_1, \dots, Def_n \vdash Exp \leadsto \phi$ in Figure 3.1.

The compilation rules are meant to be understood *coinductively*: when an identifier is found, the relevant definition is picked from the specification, and due to recursion between definitions, this can lead to cycles. While in an inductive system this would lead to invalid infinite proof trees, the coinductive interpretation "closes the loop" and produces a regular TC term. Due to this, the constraint imposed in Definition 20 is crucial, as an invalid definition like main = main could otherwise be translated to *any* regular term with the coinductive interpretation (we recall that coinduction interpretation of an inference system allows all infinite proof trees):

$$
\text{C-ID} \ \cfrac{\text{C-ID} \ \cfrac{\vdots}{(\text{main} = \text{main}) \vdash \text{main} \leadsto \phi}}{(\text{main} = \text{main}) \vdash \text{main} \leadsto \phi}
$$

The compilation of constants empty, all, and none is trivial as equivalent constants exist in TC (see rules C-EMPTY, C-ALL, and C-NONE). The same holds for intersection, union, shuffle and variable blocks, and the compilation just needs to co-recursively compile the subterms (rules C-BIN-OP and C-VAR).

When an event type is found, it needs to be translated to a valid TC term (event types by themselves are not TC terms), so a prefix is generated, with the event type followed by the empty trace (rule C-EVTYPE).

Since generic parameters in RML are declared in definitions, when an identifier is found the compilation also needs to take care of generic arguments, if any. Rule C-ID retrieves the relevant definition from the context (see the side condition), compiles the expression on the right, and turns the resulting TC term in a generic one with the appropriate variables ($\langle x_1, \dots, x_m \rangle . \phi$). Such parameters are then instantiated with the arguments used in the RML expression, producing the generic application ($\langle x_1, \dots, x_m \rangle . \phi \rangle \langle \zeta_1, \dots, \zeta_m \rangle$).

Concatenation is translated in two different ways depending on the shape of the expression on the left: if it is an event type, then a prefix is produced (rule C-CAT-1); otherwise, TC concatenation is used (rule C-CAT-2). This way, RML concatenation expressions in the shape *evType exp* are translated to *evType*:*exp*, rather than (*evType* : ϵ) · *exp*, as it is expected. Note that rule C-EVTYPE is still needed, since in RML event types are proper expression and can be used anywhere, even outside concatenation expressions.

Prefix closure compilation does not work in a compositional way, and it resorts to the transformation formalized in Definition 21 (rule C-PC).

Compilation of regular expressions operators needs to change the structure of the resulting term. For the optional operator ?, the compiled term ϕ is inside a union operation together with the empty trace ϵ, so that termination is allowed, but the semantics is otherwise unchanged from that of ϕ (rule C-OPT). Kleene star is implemented producing a regular term that either stops (ϵ) or uses the compiled term and then recursively start again from the initial choice, effectively making the verification process iterate over ϕ' as many times as needed (rule C-STAR). Plus operator is compiled in a similar way, with the exception that at least one iteration is forced to happen in $\phi' \cdot \phi$ (rule C-PLUS). Indeed, in regular expressions, $e{+} = e\, e{*}$.

$$\text{C-EMPTY} \; \frac{}{Def_1, \dots, Def_n \vdash \text{empty} \rightsquigarrow \epsilon} \qquad \text{C-ALL} \; \frac{}{Def_1, \dots, Def_n \vdash \text{all} \rightsquigarrow \mathbf{1}}$$

$$\text{C-NONE} \; \frac{}{Def_1, \dots, Def_n \vdash \text{none} \rightsquigarrow \mathbf{0}} \qquad \text{C-EVTYPE} \; \frac{}{Def_1, \dots, Def_n \vdash \tau \rightsquigarrow \tau : \epsilon}$$

$$\text{C-ID} \; \frac{Def_1, \dots, Def_n \vdash Exp \rightsquigarrow \phi}{Def_1, \dots, Def_n \vdash id\langle \xi_1, \dots, \xi_m \rangle \rightsquigarrow (\langle x_1, \dots, x_m \rangle . \phi)\langle \xi_1, \dots, \xi_m \rangle} \; Def_i = (id\langle x_1, \dots, x_m \rangle = Exp)$$

$$\text{C-CAT-1} \; \frac{Def_1, \dots, Def_n \vdash Exp \rightsquigarrow \phi}{Def_1, \dots, Def_n \vdash \tau \, Exp \rightsquigarrow \tau : \phi}$$

$$\text{C-CAT-2} \; \frac{Def_1, \dots, Def_n \vdash Exp_1 \rightsquigarrow \phi_1 \qquad Def_1, \dots, Def_n \vdash Exp_2 \rightsquigarrow \phi_2}{Def_1, \dots, Def_n \vdash Exp_1 \, Exp_2 \rightsquigarrow \phi_1 \cdot \phi_2} \; Exp_1 \neq \tau$$

$$\text{C-BIN-OP} \; \frac{Def_1, \dots, Def_n \vdash Exp_1 \rightsquigarrow \phi_1 \qquad Def_1, \dots, Def_n \vdash Exp_2 \rightsquigarrow \phi_2}{Def_1, \dots, Def_n \vdash Exp_1 \, op \, Exp_2 \rightsquigarrow \phi_1 \, op \, \phi_2} \; op \in \{\wedge, \vee, |\}$$

$$\text{C-VAR} \; \frac{Def_1, \dots, Def_n \vdash Exp \rightsquigarrow \phi}{Def_1, \dots, Def_n \vdash \{x; Exp\} \rightsquigarrow \{x; \phi\}}$$

$$\text{C-PC} \; \frac{Def_1, \dots, Def_n \vdash Exp \rightsquigarrow \phi'}{Def_1, \dots, Def_n \vdash Exp! \rightsquigarrow \phi} \; \phi = pc(\phi')$$

$$\text{C-OPT} \; \frac{Def_1, \dots, Def_n \vdash Exp \rightsquigarrow \phi}{Def_1, \dots, Def_n \vdash Exp? \rightsquigarrow \epsilon \vee \phi}$$

$$\text{C-STAR} \; \frac{Def_1, \dots, Def_n \vdash Exp \rightsquigarrow \phi'}{Def_1, \dots, Def_n \vdash Exp* \rightsquigarrow \phi} \; \phi = \epsilon \vee (\phi' \cdot \phi)$$

$$\text{C-PLUS} \; \frac{Def_1, \dots, Def_n \vdash Exp \rightsquigarrow \phi'}{Def_1, \dots, Def_n \vdash Exp+ \rightsquigarrow \phi' \cdot \phi} \; \phi = \epsilon \vee (\phi' \cdot \phi)$$

$$\text{C-FILTER-1} \; \frac{Def_1, \dots, Def_n \vdash Exp \rightsquigarrow \phi}{Def_1, \dots, Def_n \vdash \theta(p_1, \dots, p_m) \gg Exp \rightsquigarrow (\phi' \wedge \phi) \, | \, \phi''} \; \begin{array}{l} \phi' = \epsilon \vee (\theta(p_1, \dots, p_m) : \phi') \\ \phi'' = \epsilon \vee (\overline{\theta}(p_1, \dots, p_m) : \phi'') \end{array}$$

$$\text{C-FILTER-2} \; \frac{Def_1, \dots, Def_n \vdash Exp_1 \rightsquigarrow \phi_1 \qquad Def_1, \dots, Def_n \vdash Exp_2 \rightsquigarrow \phi_2}{Def_1, \dots, Def_n \vdash \theta(p_1, \dots, p_m) \gg Exp_1 \, ; Exp_2 \rightsquigarrow \phi} \; \begin{array}{l} \phi' = \epsilon \vee (\theta(p_1, \dots, p_m) : \phi') \\ \phi'' = \epsilon \vee (\overline{\theta}(p_1, \dots, p_m) : \phi'') \\ \phi = (\phi' \wedge \phi_1) \, | \, (\phi'' \wedge \phi_2) \end{array}$$

$$\text{C-CONDITIONAL} \; \frac{Def_1, \dots, Def_n \vdash Exp_1 \rightsquigarrow \phi_1 \qquad Def_1, \dots, Def_n \vdash Exp_2 \rightsquigarrow \phi_2}{Def_1, \dots, Def_n \vdash \text{if } \xi \text{ then } Exp_1 \text{ else } Exp_2 \rightsquigarrow \text{if } \xi \text{ then } \phi_1 \text{ else } \phi_2}$$

Figure 3.1: Inference system coinductively defining RML compilation $Def_1, \dots, Def_n \vdash Exp \rightsquigarrow \phi$.

Filter operators are the most complex ones to translate (rules C-FILTER-1 and C-FILTER-2). Incoming events need to be verified against different expressions depending whether they match the condition, expressed by the event type θ. TC terms ϕ' and ϕ'' allow any number of events matching θ or its negation (see Def. 9), respectively. Then, shuffle and intersection are combined to correctly validate events. In the TC term produced by rule C-FILTER-1, events matching the condition can only be consumed by the left side of the shuffle, and they are forced to be valid with respect to the specification inside the filter (ϕ). Other events are ignored by being consumed on the right through ϕ''. A similar translation is used in rule C-FILTER-2, though additionally events not matching the condition also need to satisfy property Exp_2, namely the one expressed in the second branch of the filter.

Conditional expressions (if-then-else) are translated to the equivalent TC construct. Note that filters select specifications depending on event type matching, while conditional expressions work on data expressions, possibly using the result of arithmetic and boolean computations.

Finally, note that data expressions, which are used in generic instantiations (rule C-ID) and conditionals (rule C-CONDITIONAL) do not need to be compiled, as they are the same as in TC.

On top of the translation above, we can define the semantics of RML specifications.

Definition 23. (RML semantics) Given an RML specification *Spec* in the following shape:

$$d_1;$$
$$\vdots$$
$$d_n;$$
$$\text{main} = Exp;$$
$$Def_1;$$
$$\vdots$$
$$Def_m;$$

the *semantics* of *Spec* is $[\![Spec]\!] = [\![\phi]\!]_{d_1,\dots,d_n,\overline{d_1},\dots,\overline{d_n}}$, where ϕ is the TC term such that the following judgment is derivable:

$$(\text{main} = Exp), Def_1, \dots, Def_m \vdash Exp \rightsquigarrow \phi$$

In the definition above, first the main RML expression *Exp* is compiled to a TC term ϕ using the translation formalized in Figure 3.1 (all definitions need to be carried as context). Then, we take the TC semantics of ϕ, and we do so according not only to the given event type declarations, but also to their negations (see Def.9). This is needed because compilation rules C-FILTER-1 and C-FILTER-2 from Figure 3.1 exploit such additional declarations.

EXAMPLES AND PATTERNS

The previous chapters formalized the low-level calculus TC and the high-level specification language RML. In Chapter **2** some examples of how to declare and use event types have been given, so this part of the thesis will be mainly focused on the actual specification layer, assuming sensible event types. Indeed, event type declarations are the interface to instrumentation, while here we will focus on high-level, reusable specifications and patterns to show the features of RML and how they play together to specify complex real-world properties.

All the examples presented here have been tested with our tool. These examples (and many others) can be found on our website[1], and the implementation of RML is open source and hosted on GitHub[2].

4.1 RESOURCE MANAGEMENT

The task of correctly handling the use of limited resources is ubiquitous in software, and the more concurrent and distributed the system is, the harder it becomes to verify correct usage statically. Therefore, verification of resource management is a good fit for runtime verification of a control-oriented property.

Three main typical operations on resources are *acquisition*, *use*, *release*. Examples include files, I/O channels, concurrency locks, and many others. A first, simple RML specification for resource monitoring is the following:

```
// event types acquire(id), release(id), use(id) to be defined

Main = { let id; acquire(id) (use(id)* release(id) | Main )}?;
```
Listing 4.1: Resource management specification.

Resources are identified by id, and the first expected event is the acquisition of a resource (and this is used to bind id through the event type matching semantics). Then, two sequences of events are expected to happen, in an interleaved way (hence the shuffle operator): on the left, we want to monitor the correct use of the resource that has just been acquired, that is, any number of uses followed by a release operation for *that* resource; on the right, the specification recursively goes back to the starting point, to allow for more resources to be acquired. Finally, the whole specification is followed by the regular expression operator ? to state that the block is optional, and the empty trace is also accepted (no use of resources is a correct bahavior).

The specification above is simple and compact yet powerful, as it allows monitoring of a dynamically changing set of resources, used in a concurrent way. When n resources are simultaneously acquired, the underlying TC term is recursively unfolded n times and the variable id is instantiated every time with the new identifier. The block limits the scope and allows the standard hiding of variables that is implemented in programming languages. Furthermore,

---

when resources are released, TC optimizations (see laws for identity of shuffle in Section 2.6) allow the monitor to shrink the rewritten term, keeping the time and space complexity linear with respect to the maximum number of resources *simultaneously acquired*, rather than all resources acquired since the start of the execution.

As an example of RML compilation, the specification in Figure 4.1 is compiled to the TC term ϕ, where

$$\phi = \{id; acquire(id) : ((\phi' \cdot (release(id) : \epsilon)) \mid \phi)\} \vee \epsilon$$
$$\phi' = \epsilon \vee (use(id) : \phi')$$

RML specifications can often be used in different systems thanks to event type declarations. For instance, the resource specification above can be "instantiated" for the file example provided in Chapter 2[3]:

```
acquire(id) matches {funcName: 'open', result: id};
use(id) matches {funcName: 'read'|'write', args: [id, ...]};
release(id) matches {funcName: 'close', args: [id, ...]};
relevant matches use(_) | acquire(_) | release (_);

Main = relevant >> { let id;
    acquire(id) (use(id)* release(id) | Main )
}?;
```

Listing 4.2: Adaptation of resource management specification to files.

One limitation of the resource specifications presented so far is that they check for correct usage, but not necessarily exclusive: two subsequent acquire operations on the same resource identifier would not raise an error. While this may be acceptable if we trust the identifier assignment system (for instance because it is part of the operating system), in some cases we may also want to enforce this additional constraint.

The following RML specification verifies exclusive and correct use of a dynamic set of resources.

```
// acquire(id), use(id), release(id) to be defined
acqRel(id) matches acquire (id) | release(id);

Main = {let id; acquire(id)
    ((Main | use(id)* release(id))
    /\ (acqRel(id) >> release(id) all))
}?;
```

Listing 4.3: Resource management specification for exclusive use.

This extended specification adds another constraint to the specification: after having acquired a resource `id`, any further acquire/release operation on `id` must verify `release(id)` `all`, which means that the trace has to start with a release of that resource (`all` can be understood as no more requirements need to be verified). In other words, it is now enforced that a resource needs to be released before it can be acquired again.

---

3 In Chapter 2, the pattern used to ignore elements at the end of a list is more, but the concrete syntax implemented in our tool is `...`; we chose a different mathematical notation to avoid confusion at the meta-syntax level, but in the example the implemented syntax will be used.

This example also shows how the intersection operator can be used to employ a divide and conquer approach to specifications, when different, loosely related properties can be specified independently and then joined together. The filter operator is very useful to restrict the domain of events to those that are relevant to a specific piece of RML code.

4.2 STATE VARIABLES

RML generics can be used to carry values during the verification process, and data expressions allow arithmetic and boolean computations on them. Furthermore, with conditionals, decisions during the verification procedure can be taken depending on such values.

The following example (Listing 4.4) shows how to use state variables to ensure that the use of limited resources does not exceed a given total amount.

```
// available(total), use(total, used) to be defined

Use<total> = if (total > 0)
    { let used; use(total, used) Use<total-used> }
    else empty;
Main = { let total; (available(total) Use<total>)! };
```

Listing 4.4: Specification for operations on limited resources.

First, an event matching `available(total)` lets us know the total amount of available resources. Then, `available(total)` is matched by operations checking the availability of resources, with the parameter being the maximum usable amount. Events that actually consume resources match `use(total, used)`, so that the specification both captures the total amount currently available and the amount that has been used.

Generics, variables and conditionals can be used together to specify behaviors that do not directly depend on observed data, require some computation over such data.

4.3 LIFO PROPERTIES

RML can can be successfully employed to verify also data-oriented properties to check the correct implementation of data types. One of the most commonly used data type are *stacks*. Besides the data type itself, LIFO properties are often useful in concurrent and distributed programming. Nested locks, for instance, can be verified with LIFO poperties (Kahlon, Ivancic, and Gupta, 2005), but this poses undecidability problems for static verification and model checking (Atig et al., 2017). However, as it will be shown, the problem can be successfully tackled with runtime verification and RML.

Stacks must support at least *push* (insertion) and *pop* (removal) of values. A first example of stack specification follows.

```
// push(val) and pop(val) to be defined

Main = Stack!;
```

```
Stack = { let val; push(val) Stack pop(val) }*;
```

Listing 4.5: Stack specification.

First, let us focus on the `Stack` definition. The specification expects that the stack is initially empty; after declaring variable `val` to capture the actual value that will be pushed, the specification expects a push operation, followed by the behavior of a stack (recursively), followed by a pop operation of *the same* value that was pushed before. This way we allow an unbounded number of elements to be pushed, but we expect their removal in the opposite order; intuitively, the `Stack` between `push(val)` and `pop(val)` is unfolded as much as needed.

The whole variable block is followed by a Kleene star operator: this allows to freely interleave push and pop operations (as long as the LIFO policy is verified).

Finally, `Main` adds the prefix closure on the top, so that any prefix of a valid trace with respect to `Stack` is also accepted. This way a program using a stack and terminating before emptying it will not be rejected.

Data structures usually offer methods to get the current size, that is, the number of elements inside. To take into account this functionality, we need to extend the specification and exploit generics and conditional expressions; see the following example.

```
// push(val), pop(val) and size(s) to be defined

Main = Stack<0>!;
Stack<s> = size(s)* { let val;
    push(val) Stack<s+1> pop(val) Stack<s>
}?;
```

Listing 4.6: Stack specification monitoring size.

The core of the specification, `Stack<s>`, is now *generic*: it is parametric with respect to a *computed* (rather than observed) value. In particular, the new parameter is the size of the stack.

The first difference is that zero or more calls to the size method are now allowed, and they have to return exactly `s`. Inside the specification, the size is increased at the recursive occurence because a push operation has just been observed. Note that after the pop operation the size used is again `s`.

The ability to compute values along the way adds a layer of parametricity that is not limited to the information collected from observing the execution of the system under scrutiny, and this can increase both expressivity and decoupling between the monitor and the instrumentation.

A possible improvement to the given specification is a further *decomposition*. From the specification it is not immediately clear when and how `size(s)` events are allowed, and if we were about to add support for more operations, the proposed approach would not scale.

By exploiting again filters and intersection, we can rewrite Listing 4.6 to another specification with the same semantics, but more modular and ready to be extended with more operations.

```
// push(val), pop(val) and size(s) to be defined

push matches push(_);
```

```
pop matches pop(_);
notSize not matches size(_);

Main = ((notSize >> Stack) /\ Size<0>)!;
Stack = { let val; push(val) Stack pop(val) }*;
Size<s> = (size(s) Size<s>
    \/ pop Size<s-1>
    \/ push Size<s+1>)?;
```

Listing 4.7: Modular stack specification with size monitoring.

The new specification has the same semantics as the previous one, though now the monitoring of structural modifications and that of the stack size are clearly decoupled. In Size<s> no check on the size s is required to accept a pop event, because Stack already forbids pop operations on the empty stack.

More pieces could be easily added with the intersection operator as above.

The specification can also be scaled to handle multiple stacks, with augmented event types taking an additional argument to identify the stack. Event types new(id) and free(id) will be introduced, encoding dynamic allocation.

```
// push(id, val), pop(id, val) and size(id, s) to be defined
// new(id) and free(id) to be defined

push matches push(_, _);
pop matches pop(_, _);
notSize not matches size(_, _);

Main = { let id; new(id) (Main | (Single<id> free(id))) }?;

Single<id> = ((notSize >> Stack<id>) /\ Size<id, 0>)!;
Stack<id> = { let val; push(id, val) Stack<id> pop(id, val) }*;
Size<id, s> = (size(id, s) Size<id, s>
    \/ pop Size<id, s-1>
    \/ push Size<id, s+1>)?;
```

Listing 4.8: Stack specification monitoring size and multiple objects.

Stack<id> and Size<id, s> are straightforward generalization of the previous definitions to also match identifiers (definitions can have more generic parameters). The top-level definition Main ensures that objects are created and deallocated correctly, and the optimizations presented in Chapter 2 will shrink the verification term after stack deallocation.

4.4   FIFO PROPERTIES

From an expressivity point of view, LIFO properties are not very hard to be specified. Context-free grammars are enough to match corresponding push and pop operations (if we ignore parametricity). If we shift our attention to First-In-First-Out properties (FIFO) then specifications become harder to be defined, as they require context-sensitive capabilities. A FIFO policy is what it is used in FIFO *queues*, another very common data type in computer science.

Let us consider the formal language $L_{FIFO} = \{\, a^m b^n c^m d^n \mid m > 0 \wedge n > 0 \,\}$. Just like non-regular languages can be proved to be so exploiting the pumping

lemma for regular languages, it can be shown that a language is not context-free using the pumping lemma for context-free languages, also called Bar-Hillel lemma (Bar-Hillel, Perles, and Shamir, 1961). The latter is however more involved as it requires to divide string in five parts, rather than three as in the case of regular languages.

Lemma 9. (*Bar-Hillel*) *For any context-free language L, there exists a pumping length $p > 0$ such that every string $s \in L$ of length $|s| = n$ (such that $n \geq p$) can be decomposed as $s = u\,v\,w\,x\,y$ so that:*

1. $|v\,x| > 0$;

2. $|v\,w\,x| \leq p$;

3. $u\,v^n w\,x^n\,y \in L$, for all $n \geq 0$.

Lemma 10. $L_{FIFO} = \{\,a^m b^n c^m d^n \mid m > 0 \land n > 0\,\}$ *is not context-free.*

Proof. By contradiction. Assuming L_{FIFO} is context-free, consider the string $a^p b^p c^p d^p$, with p being a valid pumping length for L_{FIFO} according the the Bal-Hillel lemma. The string can be decomposed as $a^p b^p c^p d^p = u\,v\,w\,x\,y$, and according to condition 2 of the lemma, $|v\,w\,x| \leq p$. Since the starting string $a^p b^p c^p d^p$, $v\,x$ cannot contain both a and c, and similarly it cannot contain both b and d (it is not long enough); $v\,x$ cannot also be empty by condition 1. Then, by condition 3, we can choose $n = 0$ and obtain $u\,w\,y$, but $u\,w\,y \notin L_{FIFO}$ (because removing $v\,x$ made either repetitions of a and c, or b and d, unbalanced), contradicting the hypothesis. □

Now consider the set $\{enqueue(0), enqueue(1), dequeue(0), dequeue(1)\}$ encoding possible operations on a queue of binary numbers, and the formal language over this set T_{FIFO} such that it only contains strings corresponding to correct sequences of queue operations. Let us use a, b, c, d as abbreviations for $enqueue(0), enqueue(1), dequeue(0), dequeue(1)$, and consider the regular language $R = a^+ b^+ c^+ d^+$.

From the definitions of L_{FIFO}, T_{FIFO}, and R, $T_{FIFO} \cap R = L_{FIFO}$. By Lemma 10, L_{FIFO} is not context-free, and the intersection of a context-free language and a regular language is still a context-free language (Hopcroft, Motwani, and Ullman, 2007). As a consequence, T_{FIFO} is *not* context-free.

We will now give an example of RML specification for the non-context-free property corresponding to the correct behavior of FIFO queues.

```
// enq(val) and deq(val) event type to be defined

deq matches deq(_);

Main = { let val;
    enq(val) ((deq | Main) /\ (deq >> deq(val) all))
}?;
```

Listing 4.9: Specification for correct FIFO queue usage.

The queue is assumed to be initially empty, thus the only operation that is allowed at the beginning is enqueue, and the inserted value is stored in variable val. Verification then proceeds with the intersection of two properties.

On the left, (deq | Main) allows the removal of an element (since we just inserted one, this is expected to happen) and the possibility to add more

elements by recursively using `Main`. Here we are only concerned to ensure that every time an element is inserted, it will be removed at some point, so we do not care about the specific `val`.

On the right, (`deq >> deq(val)` `all`) states that all dequeue operations must go through `deq(val)` `all`. This is equivalent to say that the first element to be removed must be `val`, according to the FIFO policy.

When more elements are inserted, the rewriting procedure will unfold the recursion and more interesections will be added, but because of the semantics of the operator, *all* of them will have to hold. This effectively ensure FIFO order regardless of how many elements are in the queue.

Note that the two parts, joined with intersection, are both needed: on the left we just match the number of enqueue and dequeue operations, not the values; on the right, we ensure removals happen in the right order, although they are not forced to happen.

The example also shows the importance of RML deterministic semantics (regarding the shuffle in this case). If the semantics would not be constrained to choose the term on the left (in (`deq | Main`) and its unfolding) the specification would also accept incorrect traces like `enq(0), enq(0), enq(1), deq(0), deq(1), deq(0)`

The queue specification can be extended in many ways, for instance by adding a prefix closure operator to allow termination on non-empty queues. A more interesting extension is, as done with stacks, the monitoring of the size.

```
// event types enq(val), deq(val), size(s) to be defined

enq matches enq(_);
deq matches deq(_);
notSize not matches size(_);

Main = (notSize >> Queue) /\ Size<0>;

Queue = { let val;
    enq(val) ((deq | Queue) /\ (deq >> deq(val) all))
}?;

Size<s> = (size(s) Size<s>
    \/ deq Size<s-1>
    \/ enq Size<s+1>)?;
```
<div align="center">Listing 4.10: Queue specification with size.</div>

The specification is extended in a compositional way, similarly as done for the modular stack specification with size monitoring in Listing 4.7. The `Size<s>` specification is the same, except for the fact that event types `pop` and `push` have been replaced with `deq` and `enq`, respectively.

Several queue specifications have been developed that are extensions of the ones proposed here, including randomized queues and queues that does not allow for repeated elements[4].

---

4 https://rmlatdibris.github.io/examples/fifo.html

4.5   ITERATOR PATTERN

The iterator pattern (Gamma et al., 1995) is an object-oriented pattern used in implementations of container objects to give the user the ability to traverse containers without exposing their internal components.

Iterator have two main functionalities:

- hasNext: this method returns a boolean indicating whether there are more elements to get from the container;

- next: returns the next element in the traversal and move the iterator forward, or throws an error if there is no such element.

If used correctly, iterators should never raise errors on calls to next. The correct behavior can be formalized with the following RML specification.

```
// hasNext(b) and next to be defined

Main = (hasNext(true) next)* hasNext(false);
```
Listing 4.11: Iterator pattern specification.

The specification is a regular expression. First, it allows zero or more invocations of hasNext returning true followed by next, and then the trace must end with an invocation of hasNext returning false.

The specification above is quite rigid, as it enforces (arguably) good practices: no multiple subsequent calls to hasNext can be made, and the whole container must be traversed. Of course the specification could be modified to be more flexible according to the verification goals.

The flexible support for parametric specifications of RML is well suited to runtime verification object-oriented programming (Ancona, Dagnino, and Franceschini, 2018; Ancona, Ferrando, Franceschini, et al., 2017), especially if the instrumentation layer exposes object identifiers[5]. This makes it easy to generalize specifications to monitor a container object.

For instance, consider the iterator example with two additional pieces of information:

- when an iterator is created, an event matching iterator(id) is observed, where id is the unique identifier of the newly created objects (this essentially amounts to monitor constructors or factory methods);

- hasNext and next have an additional parameter id encoding the identifier of the target object of the method call.

Then, a statically unknown, dynamically growing set of iterators can be monitored with the following RML specification:

```
// iterator(id), hasNext(id, b) and next(id) to be defined

Main = {let id; iterator(id) (Iterator<id> | Main)}?;
Iterator<id> = (hasNext(id, true) next(id))* hasNext(id, false);
```
Listing 4.12: Multiple iterators specification.

---

5 Depending on the programming language, objects can natively have an identifier, or the instrumentation component can assign one to them.

The technique is based on shuffle and recursion combined, similarly to what has been done for multiple resource management. Indeed, this is the general RML pattern that can be used to lift a property from a single entity to multiple ones, possibly acting concurrently.

Finally, Listing 4.13 specifies the correct behaviors of multiple iterators over the same list. Additionally, this specification monitors events that structurally change the list, i. e., that change the size of the data structure; such events are encoded by event type list.

```
// hasNext(id, b) and next(id) to be defined
// iterator(id), free(id), and list to be defined

hasNextOrNext(id) matches hasNext(id, _) | next(id);
it matches iterator(_) | hasNextOrNext(_) | free(_);
notIterator not matches iterator(_);
listOrIterator(id) matches hasNextOrNext(id) | free(id) | list;

Main = ListSafe /\ it >> Iterators;

ListSafe = notIterator* {let id; iterator(id) (ListSafeIter<id>
    /\ ListSafe)}?;
ListSafeIter<id> = listOrIterator(id) >> hasNextOrNext(id)*
    list* free(id) all;

Iterators = {let id; iterator(id) (Iterator<id> free(id) |
    Iterators)}?;
Iterator<id> = ((hasNext(id, true)+ next(id))* hasNext(id,
    false)+)!;
```
Listing 4.13: Multiple iterators on the same list, with checks for structural modifications.

Creation and disposal of iterators is matched by events iterator(id) and free(id). The specification verifies that events matching next(id) occur after those matching hasNext(id, true), and that the result of hasNext(id, res) can only change after next(id). Here the iterator is not required to necessary consume the whole sequence of events.

The generic ListSafeIter<id> verifies that if the list is structurally modified, then disposal of iterator id is the only possible event that can occur on it after the modification of the list.

5
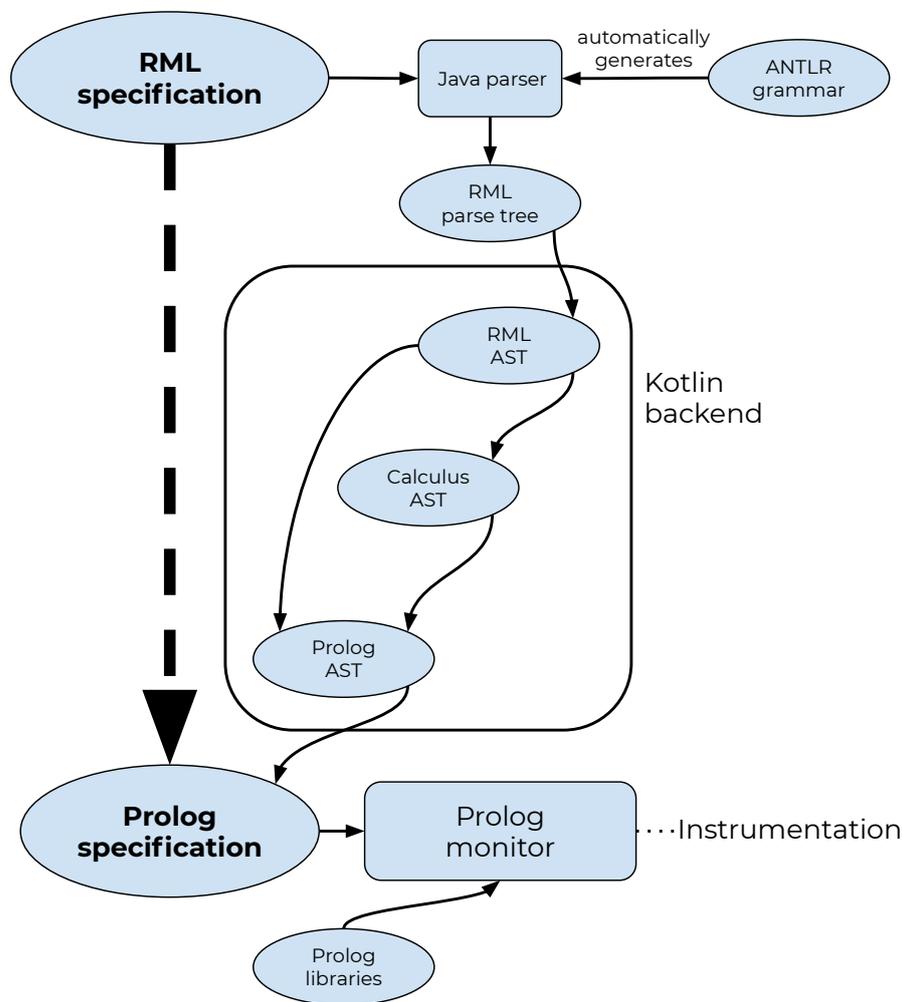

Figure 5.1: Architectural overview of RML implementation.

This chapter provides some details of the implementation of RML hosted on GitHub[1]. The implementation of the RML system consists of many modules using different languages and frameworks, and an overview of the architecture is depicted in Figure 5.1. The final goal is to generate from an RML specification a monitor implemented in Prolog, the language of choice for the implementation of the rewriting semantics presented in Chapter 2.

The GitHub repository includes:

- a *compiler* from RML to Prolog;

- a Prolog *monitor* implementing RML semantics;

---

1 https://github.com/RMLatDIBRIS

- an example of a real-world *instrumentation* developed for Node.js[2] (the cross-platform JavaScript runtime environment).

A website[3] devoted to RML and its usage has also been created, and can be used as an introductory resource to the language.

5.1 COMPILER

As it is customary in language implementation, the first step is lexical and syntactical anlysis of RML source code. This is implemented using *ANTLR*[4] (Parr and Quong, 1995), an advanced, powerful and flexible parser generator. The syntax of RML is formalized as an ANTLR grammar: examples from the lexical and syntactical grammars are shown in Listings 5.1 and 5.2.

```
ELLIPSIS: '...' ;
INT: [0-9]+ ;
FLOAT: INT '.' INT ;
STRING: '\'' [ a-zA-Z0-9_.]* '\'' ;
```

Listing 5.1: Portion of ANTLR lexical grammar for RML.

```
// RML expression syntax

exp: exp '*' # starExp
   | exp '+' # plusExp
   | exp '?' # optionalExp
   | exp '!' # closureExp
   | <assoc=right> exp exp # catExp
   | exp '/\\' exp # andExp
   | exp '\\/' exp # orExp
   | exp '|' exp # shufExp
   | evtype '>>' leftBranch=exp (':' rightBranch=exp)? #
    filterExp
   | 'empty' # emptyExp
   | 'all' # allExp
   | '{' 'let' evtypeVar (',' evtypeVar)* ';' exp '}' # blockExp
   | 'if' '(' dataExp ')' exp 'else' exp # ifElseExp
   | expId ('<' dataExp (',' dataExp)* '>')? # varExp
   | evtype # evtypeExp
   | '(' exp ')' # parenExp
   ;
```

Listing 5.2: Portion of ANTLR parsing grammar for RML.

Lexical rules are mostly based on standard regular expression operators, while ANTLR parsing grammar is more interesting, as it is based on quite a few features that are useful for the implementation of RML:

---

2 https://nodejs.org/en/
3 https://rmlatdibris.github.io/
4 https://www.antlr.org/

- the grammar syntax is based on a widely understood formalism, namely contex-free grammars;

- productions can be decorated with names (introduced by #): this will be used by the parser generator to introduce Java classes with meaningful names to implement the different kinds of nodes of the parse tree in the produced parser;

- associativity can be easily changed (see concatenation);

- left-recursion, which is often a problem in parsing, is automatically handled by ANTLR (the only limitation is related to mutually left-recursive rules);

- when parts of the production are assigned a name (as in `leftBranch=exp`), the generated parser will produce a parse tree whose nodes will be objects with fields named accordingly.

When fed with a grammar, ANTLR generates a parser; many languages are supported, but in our case Java is used. Such parser produces a parse tree supporting the visitor pattern (Gamma et al., 1995), which allows users to traverse the tree according to their needs.

While the automatically generated parser is a Java program, the software we developed for the internals of the compiler is written in *Kotlin*[5]. Kotlin is a modern, concise, statically typed, object-oriented programming language that runs on the JVM and is fully interoperable with Java.

The parse tree generated by ANTLR is a low-level representation of the RML specification, as it still includes many parsing-related information. The next step of the compilation process is to translate the parse tree into a more manageable abstract syntax tree (AST) implemented in Kotlin. The language is well suited to the task, and AST classes are encoded in just a few lines, as shown in Listing 5.3. Such classes are produced by the Kotlin code which interfaces with Java and implements the visit of the parse tree. Sticking to Java for the whole project would have led to a lot of boilerplate code, as Java is much more verbose.

AST classes are implemented with a combination of *data classes* and *sealed classes*. The former feature makes the Kotlin compiler automatically generate all common class methods (`toString`, `equals`, etc) and gives the instances a value (as opposed to reference) semantics, when compared to other instances. The latter ensures that the whole class hierarchy is known at compile-time; this, combined with Kotlin type inference algorithm and its functional programming support, allows the developer to choose a functional style based on pattern matching. This comes in handy when implementing languages and manipulating AST, tasks that are often more suited to a functional programming style.

At this point, the specification is encoded in a compact, high-level Kotlin AST encoding RML structure. Here the compilation pipeline is divided in two parts (see Figure 5.1). Event type declarations are directly compiled to a Prolog AST, while specification definition are converted to another AST encoding TC syntax (Definition 12). The additional step favors modularity by using the TC calculus as intermetidiate representation; in this way optimizations can be implemented more easily.

---

5 https://kotlinlang.org/

```kotlin
sealed class Expression

// regex-like unary operators
data class StarExpression(val exp: Expression): Expression()
data class PlusExpression(val exp: Expression): Expression()
data class OptionalExpression(val exp: Expression): Expression()

data class PrefixClosureExpression(val exp: Expression):
    Expression()

// binary operators
data class ConcatExpression(val left: Expression,
    val right: Expression): Expression()
data class AndExpression(val left: Expression,
    val right: Expression): Expression()
data class OrExpression(val left: Expression,
    val right: Expression): Expression()
data class ShuffleExpression(val left: Expression,
    val right: Expression): Expression()

// conditional operators
data class FilterExpression(val eventType: EventType,
    val filteredExpression: Expression,
    val unfilteredExpression: Expression?): Expression()
data class IfElseExpression(val condition: DataExpression,
    val thenExpression: Expression,
    val elseExpression: Expression): Expression()
```

Listing 5.3: A code snippet from the RML AST written in Kotlin.

The two lines then meet again and produce the AST of the resulting Prolog program. Since the Prolog dialect of choice is fixed (SWI-Prolog[6]) the AST is closely related to its particular syntax, and exploits it whenever convenient. This allows the RML compiler to produce a human-readable Prolog program that can be inspected if needed (see Listing 5.4).

The ASTs for Prolog and TC are structured in the same way as the RML AST (see Listing 5.3).

```prolog
:- module(spec, [(trace_expression/2), (match/2)]).
:- use_module(monitor(deep_subdict)).
:- use_module(library(clpr)).
match(_event, push(Val)) :- deep_subdict(_{event:"func_pre",
    name:"mypush", args:[Val]}, _event).
match(_event, pop(Val)) :- deep_subdict(_{event:"func_post",
    name:"mypop", res:Val}, _event).
match(_event, relevant) :- match(_event, push(_)).
match(_event, relevant) :- match(_event, pop(_)).
match(_event, any) :- deep_subdict(_{}, _event).
match(_event, none) :- not(match(_event, any)).
trace_expression('Main', Main) :- (Main=((relevant>>Stack);1)),
    (Stack=star(var(val,
    ((push(var(val)):eps)*(Stack*(pop(var(val)):eps)))))).
```

Listing 5.4: Example of a RML specification compiled to Prolog with our tool.

5.2 PROLOG SEMANTICS

The main reason why Prolog has been the language of choice for the implementation of the rewriting semantics relies in the underlying programming paradigm. Logic programming is based on rules and unification, thus a logic program is extremely similar to an inference system, and the translation of the rewriting rules is natural. Consider for instance the following rewriting rule:

$$\frac{\phi_1 \xrightarrow{o} \phi_3 \ ; \sigma_1 \qquad \phi_2 \xrightarrow{o} \phi_4 \ ; \sigma_2}{\phi_1 \wedge \phi_2 \xrightarrow{o} \phi_3 \wedge \phi_4 \ ; \sigma} \ \sigma = \sigma_1 \cup \sigma_2$$

In our Prolog implementation this is represented by the following clause:

```prolog
next(T1/\T2, O, T, S) :- !,
                    next(T1, O, T3, S1),
                    next(T2, O, T4, S2),
                    merge(S1, S2, S),
                    conj(T3, T4, T).
```

The atoms in the clause have the following meaning:

- ! (known as "cut" in logic programming) tells the Prolog interpreter not to backtrack in case of failure, since there are no other rules applicable for the intersection operator;

- `next(T1, 0, T3, S1)` recursively implements $\phi_1 \xrightarrow{o} \phi_1' \ ; \sigma_1$;

---

6 https://www.swi-prolog.org/

- `next(T2, 0, T4, S2)` recursively implements $\phi_2 \xrightarrow{o} \phi_2' ; \sigma_2$;

- `merge(S1, S2, S)`: the predicate `merge` implements substitution merging;

- `conj(T3, T4, T)` implements the shrinking optimizations described in Chapter 2 to produce the term $\phi_3 \wedge \phi_4$, and simplify it if possible.

Parametric variables need to be implemented with care, as variables in the specification must not be confused with Prolog variables. In old implementations we tried to encode parametric specifications with Prolog variables, but that clutters the implementation with bookkeeping code and meta-programming features. The current implementation takes a simpler approach and encode a specification variable x as a functor `var(x)` (note that it does not contain any variable, which cannot start with a lowercase letter in Prolog).

Substitutions are implemented with association lists, that is, lists of terms `X=T`. Association lists are widely used in Prolog, and the whole list library can be used to manipulate them, helping with the implementation of substitution application, merge, etc.

Since TC terms are regular, it is extremely useful to have language support for cyclic expressions. SWI-Prolog directly supports cyclic terms by allowing unification of a variable `X` with a term containing `X` itself. For instance, unification `X = f(X)` succeeds and instantiates variable `X` with the (only) regular term satisfying that equation.

One of the main reasons why we chose SWI-Prolog in particular as a dialect, is its native support for *coinductive logic programming* (Simon et al., 2006). Since terms are regular, substitution application is coinductively defined on them (Definition 5). In coinductive logic programming, predicates are labelled either as inductive (by default) or coinductive. When coinductive predicates are involved in the derivation, after each resolution step the interpreter checks whether the current goal unifies with any of the previous ones, and if it does, then the cycle is detected and the resolution concludes producing the substitution resulting from that unification.

Substitution application to prefix terms `ET1:T1`, for instance, is defined by the following Prolog clause:

```
apply_sub_trace_exp(S, ET1:T1, ET2:T2) :- !,
    apply_sub_event_type(S, ET1, ET2),
    apply_sub_trace_exp(S, T1, T2).
```

If we consider the cyclic SWI-Prolog term (corresponding to a valid regular TC term) defined by the unification `T1=ET1:T1`, then the standard inductive resolution would result in an infinite loop, where the interpreter keeps trying to apply substitution `S` to term `T1`.

However, in SWI-Prolog, we can mark the predicate as coinductive:

```
:- use_module(library(coinduction)).
:- coinductive apply_sub_trace_exp/3.
```

This allows us to get the correct coinductive (and, thus, terminating) semantics when substitutions are applied to regular terms.

The event type matching code is automatically generated by the RML compiler. For instance, consider the following event type declaration (from one of the tested examples on our website):

```
operation(fd) matches {
    event: 'func_pre',
```

```
    name: 'fs.write' | 'fs.read',
    args: [fd ...]
};
```

The RML compiler then produces the following clauses:

```
match(Event, operation(Fd)) :- deep_subdict(
    _{ args: [Fd|_], name: "fs.write", event: "func_pre" },
    Event).


match(Event, operation(Fd)) :- deep_subdict(
    _{ args: [Fd|_], name: "fs.read", event: "func_pre" },
    Event).
```

Differently from RML variables in specification terms, here it is correct to translate variables in event type definitions into proper Prolog variables, since we actually want them to unify with (part of) the event sent by the instrumentation and bound to Event.

The last version of SWI-Prolog introduced dictionaries, that is, record-like terms having shape tag{field1: term1, ..., fieldN: termN}. Event patterns are conveniently represented with dictionaries, because of their record-like structure.

Unfortunately, at the moment the SWI-Prolog standard library for dictionaries does not implement a predicate to check whether a dictionary is a "deep" sub-dictionary of another one by recursively traversing down the dictionary structure. We implemented such a feature as part of the library that can be used by the generated Prolog monitor, and exported it as predicate deep_subdict.

Recalling the event type matching semantics, deep_subdict allows an event to match a pattern even if it has *more* fields than requested. The implementation of the predicate in turn relies on SWI-Prolog standard library for JSON (de)serialization and manipulation, which is used to inspect the events (that are expected to be in JSON format) received from the instrumentation.

Choice patterns ('fs.write' | 'fs.write') are implemented by unfolding them to produce a set of patterns covering all the combinations of choices, without containing any choice operator themselves. Since such operator was only used once in the event type declaration above, two Prolog clauses are generated.

The RML compiler, in one of the steps of its pipeline, statically detects those specification terms that would lead to termination problems in the monitor and rejects them. An example would be the (poorly written) recursive specification $\phi = \epsilon \vee \phi$, whose formal semantics is empty (ϕ does not rewrite to any term and does not accepts the empty trace), but it would lead to infinite loops in the implementation.

Other static checks will be implemented in the future, to ease the specification writing process. However, correct RML are fully supported and the features presented in this work are all implemented in the available prototype.

5.3 MONITOR

After an RML specification has been compiled into Prolog, another Prolog module, the *monitor*, loads the Prolog specification and then starts handling events in JSON format received from the instrumentation: by using the developed library mentioned above to implement the rewriting semantics of TC, the monitor checks whether a rewrite step is allowed for the incoming event,

and, if so, updates its state with the Prolog term representing the new TC term obtained by rewriting.

Our tool can generate monitors working according to two different modes.

In *offline* mode, the monitor reads a trace from a file, which is expected to contain a single JSON event per line, then verifies the trace and outputs a verdict. Offline monitoring is still considered to be runtime verification: even if the verification process happens after termination of the program, we are still analyzing a single execution of the program.

In *online* mode, the monitor listens on an input channel for events, and verifies them one at a time, responding every time with a verdict and updating its internal state according to the rewriting semantics. Online mode is especially interesting for runtime verification of distributed and heterogeneous systems, when multiple components (possibly implemented using different technologies) are instrumented to connect to the monitor. At current time, two types of communications are supported:

HTTP   The monitor works as a web server waiting for HTTP requests. HTTP is a widely understood and implemented protocol, thus it is generally easy for instrumented components to send events to the monitor. However, an entirely new HTTP request is made every time, possibly affecting performance for high event flow-rate.

WEBSOCKET   The monitor opens a WebSocket communication channel and waits for events; this mode is highly recommended for high event flow-rate, because in this case communication is more efficient because the connection stays active once established.

Both protocols are supported by the SWI-Prolog standard library.

BENCHMARKS

The previous chapters are devoted to the design, definition, implementation and usage of RML; however, verification needs to be *efficient*, other than effective. This is even more important in runtime verification, since it has to be executed for every run of the program, and possibly at the same time if online monitoring is employed.

6.1 METHODOLOGY

There are two main factors determining the (worst case) computational complexity of RML semantics:

- *trace length*: in order to check the correctness of a trace, all of its events need to be analyzed, putting the trace length as a lower bound on the complexity of the whole process (to equalize the tests and stress the monitor, benchmarks are done on correct traces so that the monitor has to read them all);

- *specification term size*: since the semantics relies on a rewriting system, at each step the term encoding the specification can change in size, and a single rewriting step may need to analyze the whole term, thus fixing a lower bound for the complexity of a single rewriting step.

In order to show that the complexity of RML is indeed optimal, the experiments will change one of the two variables at a time, keeping the other one fixed.

The hardware and software details of the system used for benchmarking follows:

- Intel® Core™ i7-7700HQ CPU (2.8 GHz);

- 16 GB RAM;

- Ubuntu 18.04.2 LTS operating system (64 bit);

- Linux kernel 4.15.0-54-generic;

- SWI-Prolog 8.0.3.

In order to test the tool in a realistic scenario, we used an instrumentation developed for Node.js (Ancona, Franceschini, Delzanno, et al., 2017), whose repository is publicly available[1]. It is based on Jalangi2[2] (Sen et al., 2013), a JavaScript static instrumentation tool that allows the user to set callbacks to be executed before and after specific events during execution, including function/method invocations. The callbacks also have access to all the relevant metadata, that is, the program data involved in the observed operation. The instrumentation we developed collects all the metadata, serializes them in JSON format and writes them on a log file, which is later given to the monitor. The monitor is run in offline mode, because the purpose of the benchmarks is to test the performance of the monitor generated by our tool, independently of the overhead of the instrumentation layer.

---

[1] https://github.com/RMLatDIBRIS/instrumentation

[2] https://github.com/Samsung/jalangi2/

A different approach to runtime verification of Node.js programs has also been proposed. Node-MOP (Schiavio et al., 2019) brings a monitor oriented programming approach (see Section 7.4) to Node.js. By being less system-agnostic and more closely-related to the runtime environment, Node-MOP has a tighter coupling between the monitor and the program, and allow the verification code to take actions when errors are detected.

Since the benchmarks are executed on real programs, and the instrumentation is independent from the specification, a lot of events will be generated, even those that are not relevant for the specific property being monitored. Because of this, the actual RML specifications contain a top-level filter that ignores all irrelevant events.

The examples will be focused on events encoding function calls. Invocation `foo(42)` returing `true`, for instance, is represented by the following event:

```
{ event: 'func_post', name: 'foo', args: [42], res: true }
```

Events marked with `'func_pre'` and `'func_post'` are generated before and after function execution, respectively. The main difference is that the latter also includes the result.

The following sections will show and discuss the results of benchmarks testing the performance of the tool with examples from Chapter 4.

6.2   RESOURCE MANAGEMENT

Listings 4.1 and 4.3 presented RML specifications for acquisition and release of resources, checking correct acquire/release patterns and exclusive access, respectively. A variation of the pattern can be seen as a specification for a set data structure, with two operations for adding and removing elements.

More precisely, the set specification is based on the pattern used for exclusive resource access to multiple resources: here the "resources" are the elements of the set, with acquisition corresponding to the addition of a new element, release corresponding to removal of an element from the set, and usage of allocated resources corresponding to addition of an element that already is in the set. Set APIs usually allow one to detect if the operation actually changed the set by returning a boolean. There is a set operation that is not quite equivalent to any resource operation discussed so far, namely, removal of an element that is not in the set; this corresponds to all other resource uses that are only allowed when the resourse is not allocated. We will use event type `noUse` for such case.

The set specification follows.

```
// acquire(el), use(el), release(el), no_use(el) to be defined

toCheck(el) matches acquire(el) | release(el) | noUse(el);
noUse matches noUse(_);

Main = (noUse* { let el;
    acquire(el) ((use(el)* release(el) | Main)
    /\ (toCheck(el) >> (release(el) all)))
})?;
```

The tested (and instrumented) Node.js script iterates over the following steps several times:

1. all available resources are acquired;

2. acquired resources are used;

3. all acquired resources are released in reversed order.

The three steps make the specification term grow to the maximum size (since the parametric specification captures resources, the term grows with their number), and then, thanks to the shrinking optimizations, allow it to reduce to its original size.
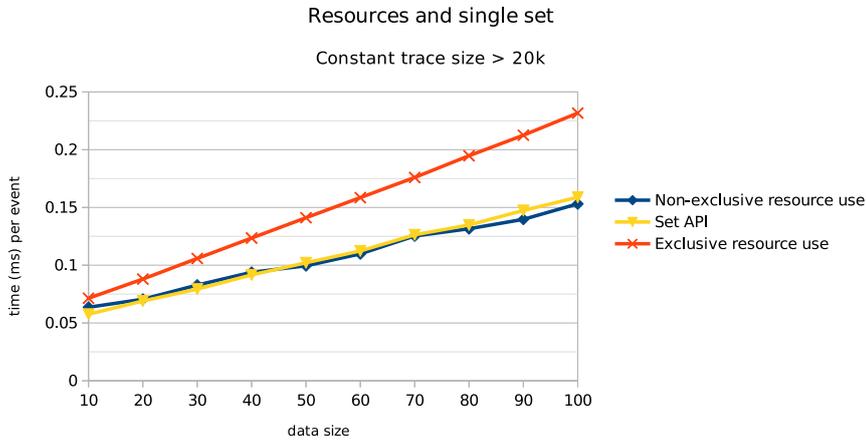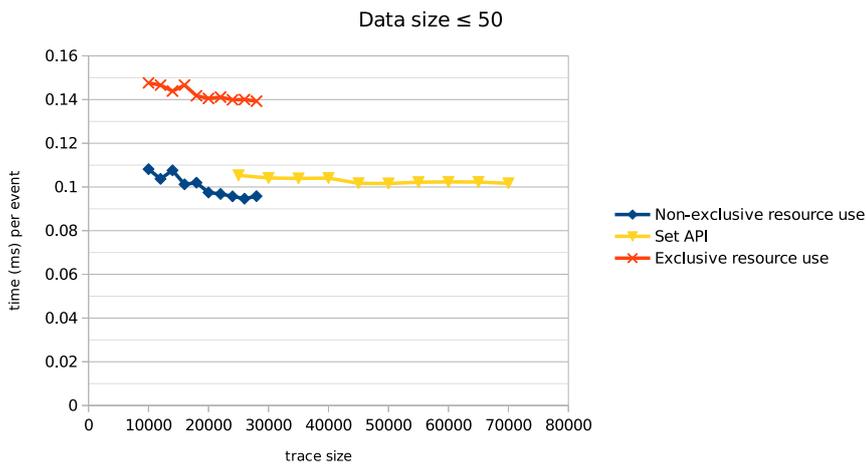

Figure 6.1: Resource-like specification patterns. Average time in milliseconds per event (Y) as a function of the maximum number of resources allocated simultaneously (X), keeping trace length constant.

The chart in Figure 6.1 shows that the optimizations work as expected: all three traces have the same length, therefore we can compare the average time spent on each event by dividing the total time by the trace length. The pattern is linear: the time needed for the monitor to process an event grows linearly with respect to the size of that term, as expected,

Figure 6.2: Resource-like specification patterns. Average time in milliseconds per event (Y) as a function of the trace length (X), keeping the maximum number of resources/elements constant.

Figure 6.2, on the other hand, shows the average time needed to process
an event as a function of the trace length, while the maximum size of the
term is fixed. Since here the trace size is part of the test, we cannot cut it such
that all three scripts produces traces of equal length. Indeed, it is completely
normal that different programs produce traces of different length, even if they
abstractly implement the same (or similar) operations. This is because the
number of generated events depends on the implementation (function calls,
in this case) which of course can change quite significantly.

The important thing to note is that the average time is constant: it is not
affected by the trace size. This means that, when the size of the term is bounded,
the total verification time will grow linearly with respect to the trace length.
This is also the best we can do in the general case: to verify a correct trace, it is
necessary to scan it all.

6.3   STACKS

Listings 4.5 and 4.6 describe the correct behavior of a stack, respectively with-
out and with size monitoring. The Node.js script used for this benchmark is a
typical stack implementation supporting push, pop and size operations. The
script main loop follows the same use pattern as the one for resources: stacks
are first added elements to their maximum capacity (variable "data size") and
emptied by removing all the elements, and these steps are iterated multiple
times. Additionally, the script used for testing the specification that includes
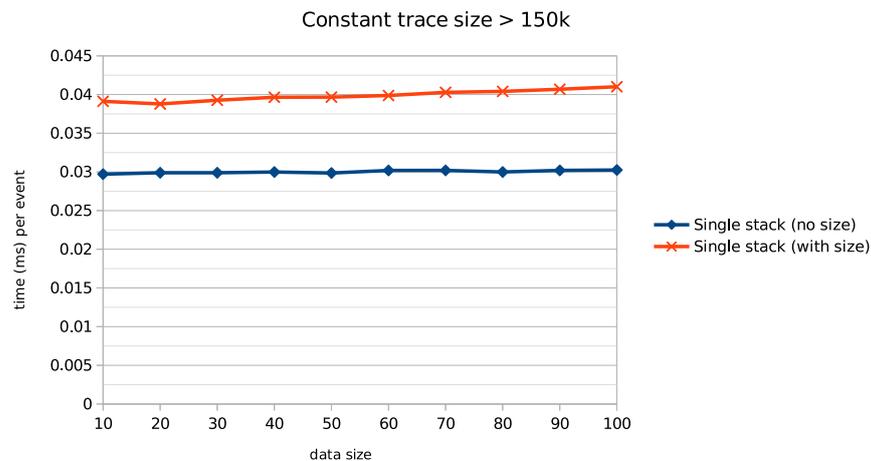the size, retrieves the size from the stack after each push and pop operation.

Figure 6.3: Single stack specification with and without size monitoring. Av-
erage time in milliseconds per event (Y) as a function of the max-
imum number of elements allowed (X), while keeping the trace
length constant.

Figure 6.3 shows the average time per event as the size of the stack increases,
which appears to be constant. For stacks, the monitor only has to inspect a
constant top-level part of the specification term to check for the size and the
last pushed element, thus the size of the whole specification term is almost
irrelevant.

The specification with size monitoring seems to show a very slowly increas-
ing average time that is not observed with the simpler specification. This may

be more related to a memory issue: the size monitoring specification is considerably more complex, thus the specification term can become quite big when the stack is full, leading to a less efficient use of memory, and more frequent invocations of the garbage collector.
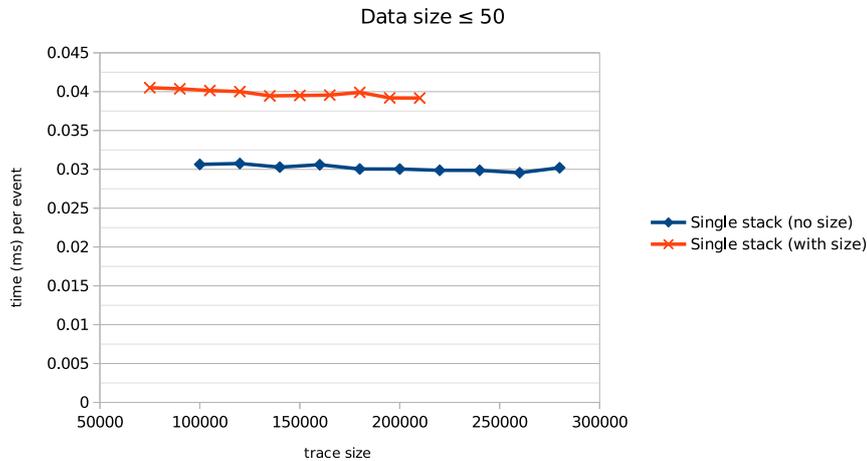

Figure 6.4: Single stack specification with and without size monitoring. Average time in milliseconds per event (Y) as a function of the trace length (X), while keeping the maximum stack size constant.

Figure 6.4 shows the average event processing time as a function of trace length. Again, the time does not increment with the trace length, confirming that the shrinking optimizations are effective.

The test for the specification monitoring multiple stacks with size operations (Listing 4.8) has been kept separated as the results are not directly comparable. With specifications monitoring only one stack, the "data size" variable determines the maximum size of the stack. With the specification monitoring multiple stacks, the variable determines the *sum* of all the lengths of the stacks involved.

Figures 6.5 and 6.6 show the results for the same experiments repeated on a script using multiple stacks. In this case the script is slightly more complex as there is also an outer loop allocating and deallocating entire stacks.

The only appreciable difference can be observed in Figure 6.5, where the time is actually increasing with the data size. This is due to the fact that the specification contains multiple stacks and the monitor needs to traverse the term until the right one is found.

6.4 QUEUES

RML specifications in Listings 4.9 and 4.10 formalize the correct behaviors of FIFO queues without and with size monitoring, respectively. Beyond them, in this benchmark three more variants of queue specifications will be tested: queue with no repetitions (Listing 6.1), random queues (Listing 6.2), and random queues with no repetitions (Listing 6.3). We recall that random queues does not obey the FIFO policy, but instead dequeue elements from the queue in random order.

Figures 6.7 and 6.8 show the experiment results for all the different queue implementations. The results are similar to those observed for resource man-
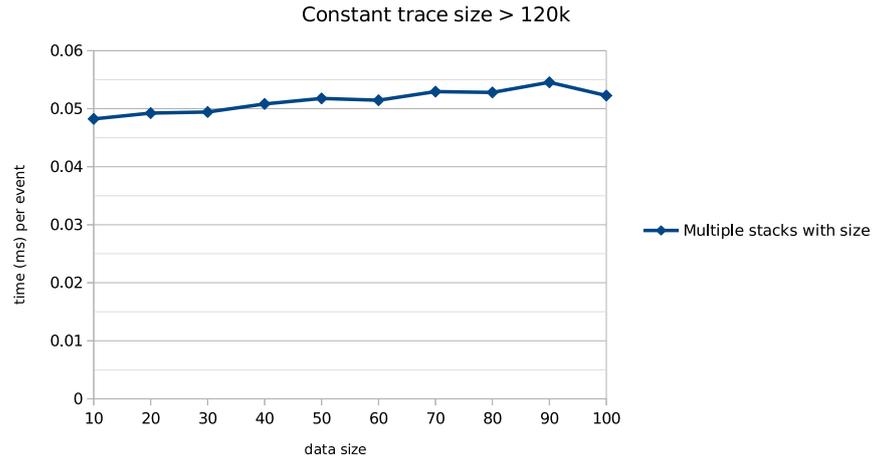
Constant trace size > 120k

Figure 6.5: Multiple stack specification with size monitoring. Average time in milliseconds per event (Y) as a function of the maximum number of elements allowed over all stacks (X), while keeping the trace length constant.
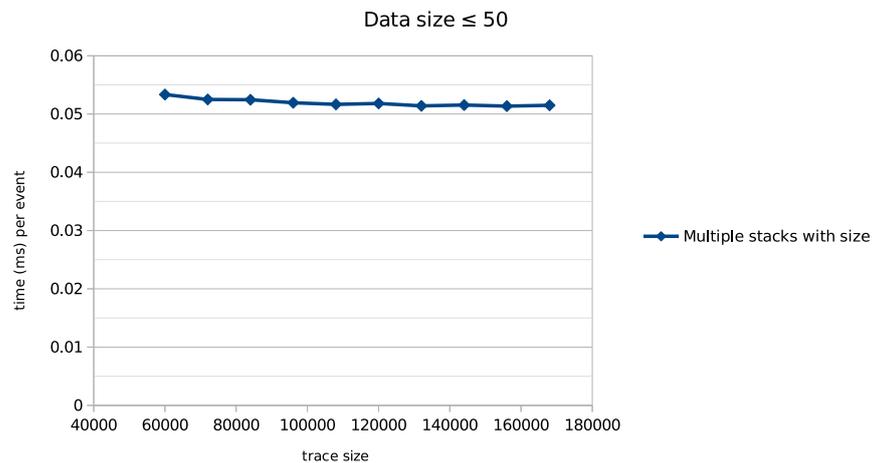
Data size ≤ 50

Figure 6.6: Multiple stack specification with size monitoring. Average time in milliseconds per event (Y) as a function of the trace length (X), while keeping the sum of all stack sizes constant.

```
// enq(val), deq(val) to be defined

deq matches deq(_);

Main = { let val;
    enq(val) ((enq(val)* deq | Main )
    /\ (deq >> deq(val) all))
}?;
```

Listing 6.1: Queue with no repetitions allowed: elements are not enqueued if already contained in the queue.

```
// event types enq(val), deq(val)

Main = { let val; enq(val) (deq(val) | Main) }?;
```

Listing 6.2: Random queue.

```
// event types enq(val), deq(val)

Main = { let val; enq(val) (enq(val)* deq(val) | Main) }?;
```

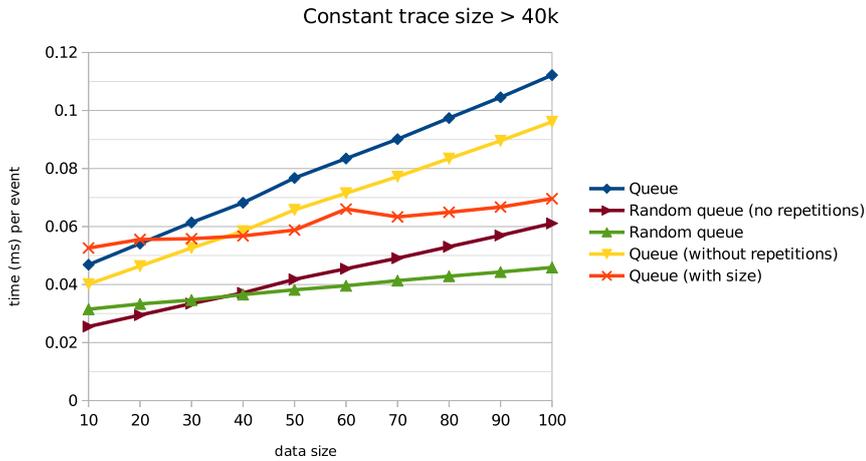Listing 6.3: Random queue with no repetitions allowed.

Figure 6.7: Different queue implementations. Average time in milliseconds per event (Y) as a function of the maximum size of the queue (X), while keeping the trace length constant.
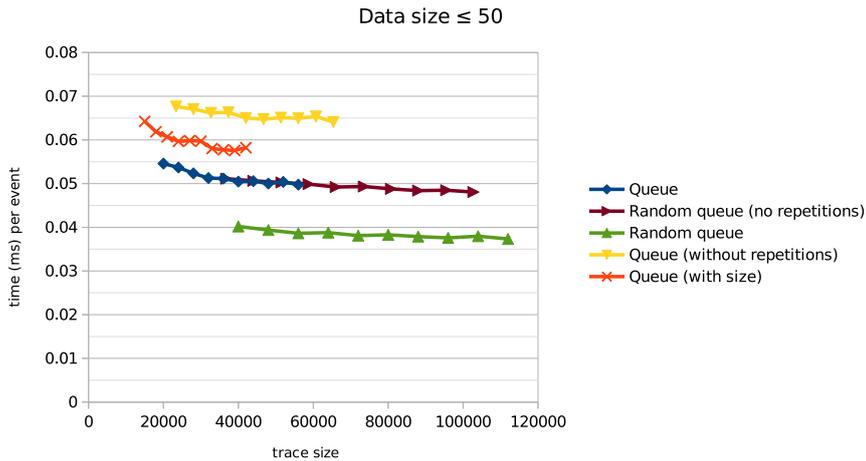

Figure 6.8: Different queue implementations. Average time in milliseconds per event (Y) as a function of the trace length (X), while keeping the maximum size of the queue constant.

agement: Figure 6.7 shows that the time spent for monitoring a single event increases in a linear way with respect to the maximum queue length; Figure 6.8 on the other hand shows how the total monitoring time linearly increases with the trace length, when the size of the queue is bounded.

6.5 FINAL REMARKS

From an absolute perspective, the experiments show satisfying and encouraging performances, as the average time needed to process an event is in the order of magnitude of 0.1 or 0.01 milliseconds, depending on the experiment. This would allow an event throughput of 10,000 to 100,000 events per second, which suggests that the monitoring itself is suitable for real-time verification and analysis of very long traces.

Of course the total overhead of a runtime verification system also depends on the instrumentation, and on how that sends events to the monitor. Implementing an efficient instrumentation, however, is a completely system-dependent task, while the goal of this Chapter was to show the performance of RML itself.

RELATED WORK

This Chapter describes the main approaches to runtime verification that have been proposed in the scientific literature. Even if the field is still relatively young (almost all contributions to runtime verification have been published in the last 20 years), a considerable number of languages, tools, architectures and views have been proposed. A precise comparison between RML and every other tool would be both extremely hard and possibly not really useful, due to many profound differences: application domains, generality of the approach, and abstraction levels are only some of the identifying characteristics of runtime verification tools.

The following sections will briefly describe the main techniques, their pros and cons, some state-of-the-art tools and their difference with respect to RML, comparing their design choices and their consequences. This is not meant to be a precise, complete classification of runtime verification tools. This is a complex research task by itself; such a taxonomy has been recently proposed by Falcone, Krstic, et al. (2018), and a graphical representation is given in Figure 7.1. With respect to such taxonomy, we can identify RML as follows.

SPECIFICATION   The *system model*, i. e., the abstraction level at which the program is observed, is defined by the intrumentation; RML specifications are always *explicit*, as they are written by users and there are no implicit properties checked by the tool.

MONITOR   RML monitors employ an operational, rewriting-based *decision procedure* (the rewriting semantics); such monitors are *explicitly generated* by our compiler, and *directly executed* as SWI-Prolog code.

DEPLOYMENT   The current implementation of RML monitors can work both *offline* and *online*; regarding the latter, the monitor itself works in a synchronous way, though the instrumentation can handle the communication with the monitor also in an asynchronous manner. The monitor is placed *outline*, meaning it is a different program executed in a different address space; the instrumentation can be inline, as it happens with our Node.js instrumentation, which is an instance of a *software instrumentation* (as opposed to hardware ones). Finally, RML monitors are *centralized*, as properties globally describe the correct behavior of a system.

REACTION   Our monitors are *passive*, since they only provide specification outputs (verdicts). However, such outcomes are delivered to the instrumentation layer, which may be either passive or active; the latter would affect the execution of the program under scrutiny to some extent.

TRACE   Traces both have a *role* in the theoretical model (where they are possibly infinite sequences of events) and in the practical observation (where they are necessarily finite sequences); everything else about the kind of information encoded in events entirely depends on the instrumentation (the *evaluation* can either refer to points or intervals in time, though in all of our examples we considered events about single execution points).
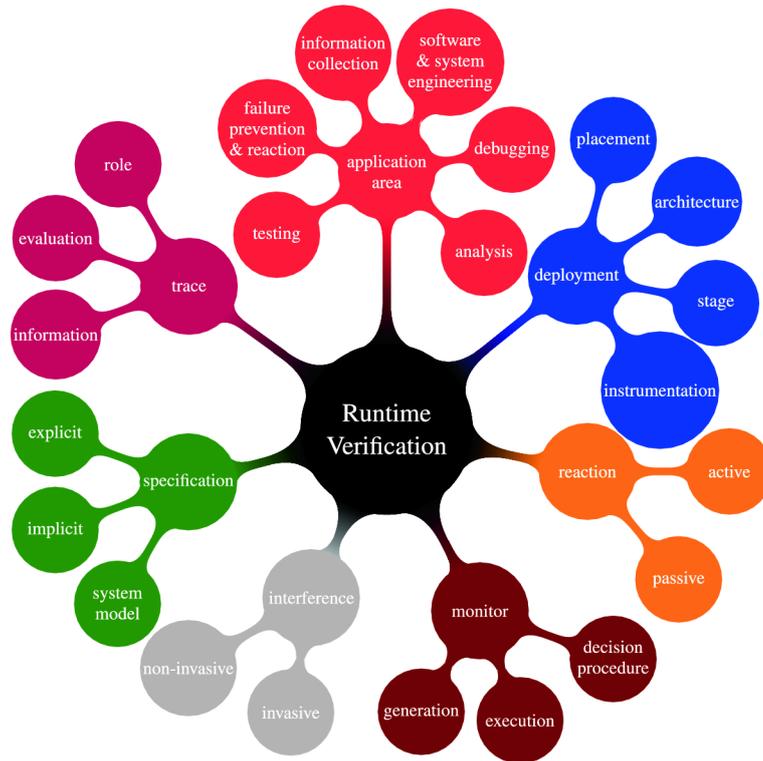
Figure 7.1: "A Taxonomy for Classifying Runtime Verification Tools" (Falcone, Krstic, et al., 2018).

INTERFERENCE   Again, this depends on the instrumentation. For our Node.js instrumentation, the interference is semantically *non-invasive*, as the code instrumentation step does not change the behavior of the program. Regarding performances, however, the overhead cannot be zero when such techniques; Chapter 6 is devoted to performance evaluation.

APPLICATION AREA   RML is designed as an *analysis* technique, complementary to testing, static analysis and possibly other tools.

The distinction between monitors and instrumentation layers is not common in runtime verification literature: indeed even an authoritative work as the taxonomy above freely mixes the two concepts. However, we believe the distinction is worth the additional effort for maximal flexibility.

Other general, comprehensive and authoritative surveys on runtime verification have been done by Bartocci et al. (2018), Delgado, Gates, and Roach (2004), Falcone, Havelund, and Reger (2013), Havelund and Goldberg (2005), Leucker and Schallhart (2009), and Sokolsky, Havelund, and Lee (2012).

7.1   TEMPORAL LOGICS

Since runtime verification has its roots in model checking, it is not surprising that logic-based formalism previously introduced in the context of the latter have been applied to the former.

Temporal logics (first studied by Prior, 1957) are a family of modal logics where modalities are used to qualify propositions in terms of time, like "property ϕ will always/eventually hold" or "property ϕ (always) held". Some

temporal logics only refer to the past or to the future, while others allow for both.

Linear Temporal Logic (*LTL*, Pnueli, 1977), is one of the most used formalism in verification. Its modalities refer to events in the future; the two fundamental operators are $X \phi$ ("ϕ has to hold at the next state") and $\phi \cup \psi$ ("ϕ has to hold at least until ψ does, which must happen at the current or a future time"). Note that the standard semantics of LTL is defined on infinite traces only. Finally, "linear" comes from the fact that only one possible path is taken into account.

To account for the incremental nature of runtime verification monitors (as opposed to static formal verification), a three-valued semantics for LTL, named LTL_3 has been proposed (Bauer, Leucker, and Schallhart, 2006b, 2011). Beyond the basic "true" and "false" truth values, a third "inconclusive" one is considered (LTL specification syntax is unchanged, only the semantics is modified to take into account the new value). This allows one to distinguish the satisfaction/violation of the desired property ("false") from the lack of sufficient evidence among the events observed so far ("inconclusive"), making this semantics more suited to runtime verification. Differently from LTL, the semantics of LTL_3 is defined on finite prefixes, making it more suitable for comparison with other runtime verification formalisms. Further development of LTL_3 led to *RV-LTL* (Bauer, Leucker, and Schallhart, 2007), a 4-valued semantics.

The expressive power of LTL is the same as of star-free ω-regular languages (Pnueli and Zuck, 1993). When restricted to finite traces, RML is much more expressive than LTL as any regular expression can be trivially translated to it. On infinite traces, the comparison is slightly more intricate and results in RML and LTL_3 having incomparable expressiveness (Ancona, Ferrando, and Mascardi, 2016).

Since RML can describe non-regular languages, it can express some properties LTL cannot. On the other hand, the U operator has no precise counterpart in RML, as happens in general in runtime verification where monitors can inspect only a finite prefix of the execution trace. A naive translation of $\phi \cup \psi$ would be $\phi^* \psi$, however, the first one expects ψ to happen in the future, while the second can accept an infinite sequence of events matching ϕ.

Considering the LTL formula is not monitorable with respect to the infinite sequence of events matching ϕ (meaning it will be considered neither valid nor invalid, Pnueli and Zaks, 2006), this expressiveness feature is not useful for runtime verification purposes.

There exist many extensions of LTL that deal with time in a more quantitative way (as opposed to the strictly qualitative approach of standard LTL) without increasing the expressive power, like *interval temporal logic* (Cau and Zedan, 1997), *metric temporal logic* (Thati and Rosu, 2005) and *timed LTL* (Bauer, Leucker, and Schallhart, 2011). Other proposals go beyond regularity (Alur, Etessami, and Madhusudan, 2004) and even context-free languages (Bollig, Decker, and Leucker, 2012).

Regardless of the formal expressivity, RML and temporalized logics are essentially different: RML has no direct support for time. However, if the instrumentation provides timestamps as part of the monitored events, then some time-related properties can still be expressed exploiting parametricity, generics and conditionals.

More insights on the use of temporal logics in runtime verification can be found in the survey papers of Leucker and Schallhart (2009) and Bartocci et al. (2018).

LTL and other temporal logics have been embedded in a more expressive framework, namely *recHML* (Larsen, 1990), which is a variant of the *modal μ-calculus* (Kozen, 1983). This allowed to formally study issues such as monitorability (Aceto, Achilleos, Francalanza, Ingólfsdóttir, and Lehtinen, 2019a) in a general framework, and derive results for free about any formalism that can be embedded as well. It would be interesting to embed TC in a more expressive calculus and get theoretical results that are missing from this presentation. Unfortunately, it is not clear whether TC and recHML are comparable at all. For instance, recHML is a fixed-point logic including both least and greatest fixpoint operators, while TC implicitly uses a greatest fixpoint semantics for recursion. On the other hand, recHML does not include a shuffle operator, and we are not aware of a way to derive it from other operators (indeed it is part of our core calculus).

7.2 ASSERTIONS

Assertions in programming are predicates over program variables, used to ensure some expected property holds. Their use in verification has been introduced with *Hoare logic* (Hoare, 1969), where imperative statements are associated with preconditions and postconditions that are expected and need to hold in order for the program to be proved correct. Today, assertions are a common programming tool that developers use to document, test and ensure expected properties in a specific point of the execution. They can be supported either at the language level or as libraries.

Further development led to *design by contract*, first introduced in the context of the Eiffel[1] language, which embraces it (Meyer, 1992). In this methodology, software components explicitly state, at the code level, which preconditions are expected from client code, which postconditions are ensured after usage, and which invariants are always held true[2]. In an object-oriented setting, for instance, method signatures would be decorated with preconditions and postconditions, and classes would explicitly contain their invariants.

When they are not statically implemented (via typical formal methods techniques), assertion-based frameworks can be understood as an instance of runtime verification, effectively monitoring and verifying the program at execution time. This approach has indeed been followed by some runtime verification tools.

Java Modeling Language (*JML*, Leavens, Baker, and Ruby, 2006) is a verification tool specifically tailored for Java programs, where methods and classes are decorated with assertions, using first-order logic extended with arithmetic and boolean expressions, plus some other features. The *OpenJML*[3] implementation (Cok, 2011) supports runtime assertion checking. From a runtime verification perspective, the tool is focused on data-oriented properties, as it allows one to assess the correctness of manipulated data in specific code locations. The implementation relies on Java bytecode[4] instrumentation.

---

[1] https://www.eiffel.org/

[2] More precisely, invariants need to hold after construction and both before and after routine calls from outside the component. It is possible for them not to hold during the execution of code from the component itself.

[3] http://www.openjml.org/

[4] The bytecode is the target language of the Java compiler, and the instruction set of the Java Virtual Machine (JVM).

In order for RML to support runtime assertion checking of pre/post-conditions and invariants in object-oriented programming, the instrumentation layer needs to monitor at least all method calls (and returns), and possibly constructors (to first evaluate the invariants). Given such infrastructure, one can validate parameters and results of methods, as previously shown in the object-oriented examples in Chapter 4, thanks to `with` guard expressions in match declarations. This idea suggests that simple assertions[5] in the design-by-contract style could be automatically translated into RML match declarations, while the main specification is used to verify control-oriented properties, by allowing only some valid sequences of method calls. Since in RML the specification is separated from the code, integration with assertion checking of pre/post-conditions would require appropriate match declarations capturing the relevant variables from the program state (i. e., from the JSON-serialized received event). This (arguably small) additional complexity seems unavoidable for language-independent tools, as properties are not connected to any particular code location. Examples in Chapter 4 on stacks and queues have shown that for some data types one can specify in RML data-oriented properties more abstractly at the interface level, independently of the specific implementation defined in the corresponding classes.

The notion of ghost variable in JML (and other tools) refers to variables only existing in the specification and not in the observed program. While this notion makes sense when one needs to distinguish between program variables and specification variables, all RML variables can be understood as ghost ones. However, first-order logic based formalisms like JML also supports universal and existential quantification over ghost variables, though with restriction to avoid undecidability. RML does not support this as it would require quantification over new variables in guard expressions of event type declarations. This keeps RML matching implementation reasonably fast, since only pattern matching and basic expressions need to be supported.

On the other hand, with RML it is easy to monitor control-oriented properties and correct API usage (Ancona, Dagnino, and Franceschini, 2018; Ancona, Ferrando, Franceschini, et al., 2017). Furthermore, if the instrumentation works at a finer granularity, RML can be used to verify properties when specific types of statements are executed. Essentially, the abstraction level entirely depends on the instrumentation layer.

Another object-oriented runtime verification tool, *SAGA*[6] (De Boer and De Gouw, 2014), targets Java programs with a different formalism based on attribute grammars. Context-free grammars are well suited for specifying protocols and control-oriented properties, while attributes are used to also support data-oriented specifications. From such attributes, assertions to be checked at runtime are generated, and the code is instrumented using Rascal (Klint, Storm, and Vinju, 2009), a DSL devoted to source code manipulation.

The choice of context-free grammars as fundamental building blocks may look similar to what is done with RML recursive definitions; however, SAGA does not directly support the shuffle and intersection operators; intersection can be recovered by associating more specifications with the same class (or other entity), but it is not clear how an RML recursive specification involving the intersection operator could be translated into SAGA.

---

5 For instance, assertions that do not require first-order quantification or ghost variables; see more related comments below.

6 https://www.cwi.nl/innovation/software/saga-a-run-time-verifier-for-java-programs/saga-a-run-time-verifier-for-java-programs

SAGA formalism requires to declare in the source code which events (method calls, for instance) need to be monitored. This resembles RML event types, though they are more general in nature and are parametric, while in SAGA data is accessed by mixing code with the specification.

7.3   STATE MACHINES

High-level (runtime) verification languages are often translated to low-level, more directly executable ones. This is the approach of RML as well, that is compiled down to the trace calculus, for which a rewriting semantics is defined, that in turn drives the implementation. This step is sometimes called monitor synthesis. While this allows one to write down the specification at a higher abstraction level, it clearly needs more work and tools to be developed (and optimized).

A different approach consists in formalizing the specification using *state machines* (a.k.a. automata or finite-state machines). Though the core concept of a finite set of states and a (possibly input-driven) transition function between them is always there, in the field of automata theory different formalizations and extensions bring the expressivity anywhere from simple deterministic finite automata to Turing machines. From a runtime verification perspective, they can be understood as executable specifications (this is also true for rewriting based approaches like RML).

An example of such formalisms is *DATE* (*Dynamic Automata with Timers and Events*, Colombo, Pace, and Schneider, 2008), an extension of the finite-state automata computational model based on communicating automata with timers and transitions triggered by observed events. This is the basis of *LARVA*[7] (Colombo, Pace, and Schneider, 2009), a Java runtime verification tool focused on control-flow and real-time properties, exploiting the expressivity of the underlying system (DATE). The tool also supports other logics, like *QDDC* (*Quantified Discrete-time Duration Calculus*, Pandya, 2000), *LUSTRE* (Halbwachs, Lagnier, and Ratel, 1992), and *counterexample traces* (Hoenicke, 2006); still, all the supported logics are internally translated to it.

The main feature of LARVA that is missing in RML is the support for temporalized properties, as observed events can trigger timers for other expected events. Currently the only way RML can monitor real-time properties is with the support of an instrumentation adding timestamps to serialized events; then, with arithmetic and boolean expression, it is possible to ensure the time distance between two events is acceptable. This approach has some limitation: such a monitor could only emit a negative verdict after the expected event (or another one) is actually observed, and not as soon as the time has expired. In the worst case, this could postpone the final verdict indefinitely, unless the instrumentation layer send clock-based events to the monitor, but this would lead to efficiency and synchronization problems, especially in distributed systems.

On the other hand, the parametric verification support of RML is more general. LARVA scope mechanism works at the object level, thus parametricity is based on *trace slicing* and implemented by spawning new monitors and associating them with different objects. The RML approach is different as specifications are always global, and they can be parametric with respect to any observed data: this makes it easier to write down parametric specifications

---

7 http://www.cs.um.edu.mt/svrg/Tools/LARVA/

that are still related to the whole system, or to a family of otherwise unrelated (and possibly dynamically discovered) objects. Limitations of the parametric trace slicing approach described above, as well as possible generalizations to overcome them, have been explored by Barringer, Falcone, et al. (2012), Chen and Rosu (2009), and Reger, Cruz, and Rydeheard (2015). In RML we decided not to reason on trace slicing but rather to keep just one (global) specification and syntactically instantiate parameters as soon as they are discovered, a task simplified by our design choice of having variable declarations together with a notion of scope for them.

Finally, the goals of the two tools are very different to start with. While RML strives to be system-independent, LARVA is devoted to Java verification, and the implementation relies on AspectJ[8] (Kiczales, Hilsdale, et al., 2001) as an "instrumentation" layer, which allows one to inject code (the monitor) to be executed at specific locations in the program under scrutiny. Aspect-oriented programming (Kiczales, Lamping, et al., 1997) is a paradigm that allows the developer to add new behavior to existing code without modifying it, enhancing modularity and separation of concerns; AspectJ brings these concepts to Java.

7.4 MONITOR-ORIENTED PROGRAMMING

Similarly as RML, which does not depend on the monitored system (not even on its instrumentation), other proposals exist in the literature that introduce different levels of separation of concerns. *Monitor-oriented programming*[9] (*MOP*, Chen and Rosu, 2007) is an infrastructure for runtime verification that is neither tied to any particular programming language nor to a single specification language; indeed, it has been instantiated on a number of formalisms. In order to add support for new logics, one has to develop an appropriate plug-in converting specifications to one of the format supported by the MOP instance of the language of choice (e. g. *JavaMOP*, Chen and Rosu, 2005), so that the infrastructure can synthesize an appropriate monitor.

MOP can also be understood as a software development methodology where the program to be verified is actually aware of the monitoring process, and the two entities can interact with each other. This approach has also been suggested in the context of *runtime reflection* (Bauer, Leucker, and Schallhart, 2006a). This is fundamentally different from RML, where the monitor basically sits on top of the existing program and it is passive in nature.

A brief discussion of the main formalisms implemented in existing MOP instances follows. Support for finite state machines has limited expressivity (they are known to be as expressive as regular languages), and they can easily be translated to RML. Extended regular expressions include intersection and, more interestingly, complement; while this does not increase the formal expressive power (regular languages are closed under both intersection and complement), negation of arbitrary expressions is not supported in RML. However, the complement (with respect to matching) of existing event type is supported through negative match declarations. Context-free grammars ara again easily embedded in RML using recursion, concatenation, union, and the empty trace. Finally, temporal logics have been discussed previously in this section.

---

8 https://www.eclipse.org/aspectj/
9 http://fsl.cs.illinois.edu/index.php/Monitoring-Oriented_Programming

State machine systems are sometimes employed in mixed system together with other types of formalism. *TraceContract*[10] (Barringer and Havelund, 2011), for instance, offers a mix of finite state machines and temporal logic features. It is implemented and available as a Scala internal DSL.

As expressive as finite state machines, regular expressions have been used in runtime verification for their simple semantics and their well-understood meaning among software developers. *Tracematches*[11] (Allan et al., 2005) use regular expressions augmented with variables to get parametricity. The tool is implemented in AspectJ and is devoted to Java programs verification, especially regarding correct usage of APIs. This is similar to what we proposed (Ancona, Dagnino, and Franceschini, 2018; Ancona, Ferrando, Franceschini, et al., 2017) using the precursor of RML, namely parametric trace expressions (Ancona, Ferrando, and Mascardi, 2017). Indeed, quite some common API usage patterns can be formalized using regular languages, though more operators (like shuffle and intersection) allow more flexibility and expressive power, as shown in Chapter 4.

7.5   RULE SYSTEMS

The core idea behind rule-based verification systems is to encode properties in a set of "rules", with rules being composed of conditions that needs to hold in order to apply the rule, events that can trigger the application of the rule, and statements about what holds after such transition. Different formalisms extend this basic conept in different directions.

RuleR (Barringer, Rydeheard, and Havelund, 2010) approach is based on dynamic activation and deactivation of rules: observed events can trigger rules that in turn can change the set of applicable rules in subsequent steps. RuleR has been proposed as a simpler, more essential and easier alternative to Eagle (Barringer, Goldberg, et al., 2004), a highly expressive first-order logic with fixpoint and linear temporal operators, in order to get a more effective runtime verification tool. For this reasons we focus on the successor.

Just like in RML, RuleR supports data parameterization, so that specifications can depend on observed data from the event. Furthermore, it also includes a different form of parametric specifications, namely, rules parameterized over other rules. This form of higher-order specifications are currently not supported in RML; extending the trace calculus in such a way should be done with care to guarantee termination of the algorithm for a single transition step.

Comparing rule-based system to RML is not easy from the usage point of view, as the underlying concepts are very different. Rule systems are arguably lower-level formalisms: the state is explicitly manipulated and one must reason mostly on single event steps. This often leads to longer specifications, as also acknowledged by this method's proponents (Havelund, 2015). On the other hand, RML offers a higher abstraction level, and language operators work at the trace level.

Formal expressive power is conjectured to be the same. Both RML and RuleR can embed context-free grammars and express context-sensitive properties. Furthermore, they both have counting ability over natural numbers, which suggests Turing-completeness when parametricity (in the sense of RML generics) is used.

---

10 https://github.com/havelund/tracecontract
11 https://patricklam.ca/research/tracematch/

Rule systems are connected to rule based artificial intelligence models, as they share the underlying deductive approach, based on deriving new knowledge from known facts. *LogFire* (Havelund, 2015), for instance, uses an algorithm from the artificial intelligence community called *Rete* (Forgy, 1982). By using different inference engines and more efficient way to store information about the observed data, the implementation of such systems can be greatly optimized, also becoming suitable for online runtime verification purposes.

As it happens with other approaches, these systems can either be implemented on their own or embedded in existing languages as libraries. For instance, RuleR is implemented in Java, while LogFire is provided as an internal Scala[12] DSL, a task for which Scala is very well suited thanks to its set of features. The same implementation choice (Scala internal DSL) has ben made for TraceContract (Barringer and Havelund, 2011), which also supports some basic rule-oriented features.

7.6 STATIC AND DYNAMIC ANALYSIS

Effective and efficient combination of static and dynamic analysis techniques is a currently investigated research problem, which poses many challenges and is highly dependent on the formalism used and the kind of system under observation. Despite the challenges, it is a natural step towards software correctness: every property that is too impractical (or impossible) to enforce statically, is postponed to dynamic analysis, ideally getting the best of both worlds. In this view, runtime verification serves as a complementary technique to more traditional ones: static analysis, formal methods, type systems, etc.

PQL[13] (*Program Query Language*, Martin, Livshits, and Lam, 2005) is a tool that allows the user to define patterns of execution to be matched against the flow of the observed program. Statically, the analyzer finds the relevant code points that need to be instrumented, thus minimizing the runtime overhead. At runtime, a monitor generated from the property (as a finite state machine) matches the sequence of observed events against the specification. PQL is mainly concerned with security properties and resource leaks.

Regarding its expressivity, basic queries can be compared to context-free grammars. However, queries are patterns that can be instantiated by an arbitrary number of matching existing objects. While this form of parametricity increses the expressiveness (intersection of context-free grammars, according to Martin, Livshits, and Lam, 2005), it seems to share similar limitations to those of the trace slicing approach previously discussed. In this sense, PQL properties are more local, as opposed to the global approach of RML.

This optimization is not uncommon in runtime verification tools; a similar analysis has also been developed for Tracematches (Bodden, Hendren, and Lhoták, 2007).

Another example of state-of-the-art combination of static and dynamic analysis framework is *Unified Static and Runtime Verification of Object-Oriented Software*[14] (*StaRVOOrS*, Chimento et al., 2015). In this case the combination has the goal of reducing the original specification by simplifying what can be proved statically, and produce a runtime specification for what needs dy-

---

12 https://www.scala-lang.org/
13 http://pql.sourceforge.net/
14 https://starvoors.github.io/

namic analysis. Thus, rather than optimizing the instrumentation, here the specification itself is "optimized".

The framework combines the static deductive verifier *KeY* [15] (Ahrendt, Beckert, et al., 2016) with the runtime verification tool LARVA (Colombo, Pace, and Schneider, 2009). The formalism of choice for the specification is *ppDATE* (Ahrendt, Chimento, et al., 2015), an extension of DATE (Colombo, Pace, and Schneider, 2008) that allows both control- and data-oriented properties to be stated, decorating automaton states with Hoare triples. From this, JML (Leavens, Baker, and Ruby, 2006) annotations are produced, and the deductive verifier uses both to prove correctness, or if it is not possible, only partial proofs. In the latter case, the specification is refined to encode what needs to be checked at runtime, compiled to DATE and used at runtime with LARVA (Colombo, Pace, and Schneider, 2009).

Given the runtime verification flavor of RML, and its independence from the monitored system, it seems hard to refine the specification with static information. However, optimizing the instrumentation for a given specification is very doable, and this could be a future extension of RML. More specifically, from the RML compilation a configuration file could be produced, including the kind of events that are actually relevant; then, such information can be used to minimize the amount of generated and monitored events, gaining more efficiency. This would be especially useful in specifications having a "global filter", as opposed to generate and then ignore most of the observed events.

7.7   PROCESS CALCULI

Process calculi (a.k.a. process algebras) are a formalism family developed to precisely describe the behavior of concurrent systems. The main entities are processes (single computing units) that can both work independently and communicate (and synchronize) with each other. Different choices of operators and different restrictions about dynamic creation of processes and communication channels led to a number of systems. Notable examples include *Communicating Sequential Processes* (*CSP*, Brookes, Hoare, and Roscoe, 1984), the *Calculus of Communicating Systems* (*CCS*, Milner, 1980) and the *π-calculus* (Engberg and Nielsen, 1986).

Despite the different goals and techniques involved, some runtime verification formalisms and tools have been inspired by process calculi, since both the communities try to model software behavior.

Java with Assertions Debugging Architecture[16] (*Jassda*, Brörkens and Möller, 2002) uses a CSP-like syntax to express properties, enriched with a notation to deal with set of events as a whole. This resembles the event types of RML which can be understood, in a sense, as set of events (those matching the event type pattern). As it happens with many other tools in object-oriented runtime verification, parametricity is limited to trace slicing based on instances.

Though process calculi are quite expressive and can model complex interactions, they miss some operators that are useful in verification. Most importantly, intersection is usually not included since it makes little sense in that context. When specifying expected properties however, intersection can be crucial: it can both increase expressiveness (contex-free languages are not closed under

---

intersection, for instance) and support modular specifications, as shown in several RML examples in Chapter 4.

7.8 STREAM RUNTIME VERIFICATION

Beyond ensuring correctness of the observed execution of the program under scrutiny, it is sometimes useful to also collect quantitative, statistical data for analysis. In this case one needs to treat a sequence of events as a whole. This is often called a *stream*, and the specification languages manipulating them are sometimes called stream computation languages, or stream languages for short. More specifically, the observation constitute input streams, which can be used to generate output streams of results. The goal of stream languages is to define the dependencies between streams, and to compute the output stream in an incremental way.

The concept was pioneered by *Lola*[17] (D'Angelo et al., 2005), a monitoring language allowing the user to manipulate streams of events and create new ones by specifying them with equations. The formalism (and algorithm) is explicitly devised for monitoring low-level software and circuits, and allow logical and arithmetical operations on streams. To keep the system efficient, operations are computed in an incremental fashion. The expressivity is higher than temporal logics and finite state machines, and some context-free properties can be encoded. Furtnermore, in *Lola 2.0* (Faymonville et al., 2016) parametricity support is added.

While initially developed in the context of synchronous systems monitoring (a system clock is assumed), stream runtime verification has been generalized to deal with asynchronous systems. *TeSSLa*[18] (Convent et al., 2018; Leucker, Sánchez, et al., 2018) is a temporal stream specification language supporting timestamped events, aiming at monitoring asynchronous, cyber-physical, real-time systems (a global order over events is still assumed).

A lower level formalism named *data transducers* (Alur, Mamouras, and Stanford, 2019) has been recently proposed, with the goal of defining a machine language for stream computations. Thus it is not meant to be directly used to write specifications, but rather to be the target language of compilation from other ones, with a focus on supporting modular compilation of queries over streams of data. The formalism has been used to translate *QRE-past*, a specification language that extends quantitative regular expressions (regular expressions with aggregate operations, Alur, Fisman, and Raghothaman, 2016) with past-time temporal logic.

The goals of ensuring correctness and computing metrics are quite different, and thus the features are not immediately comparable. However, even though RML scope is traditional runtime verification, generics and data expressions allow one to compute some values along the way (though no history of previous events is directly supported). Thanks to parametric event types, such computations can easily be computed from the observed data.

Transducers have also been used in runtime enforcement to anticipate incorrect behavior and allow acting on the system under scrutiny in a timely manner to avoid errors (Aceto, Cassar, et al., 2018).

---

17 https://www.react.uni-saarland.de/tools/lola/
18 https://www.tessla.io/

CONCLUSION AND FUTURE WORK

This thesis proposed RML, a deterministic, system-agnostic DSL for runtime verification. We briefly recap its main features.

SYSTEM-INDEPENDENCE RML does not make any assumption on the kind of system being monitored, and relies on the extremely common and universally supported JSON format to handle events. By being system-agnostic, RML is also applicable to heterogeneous systems, and its specifications can be reused in different contextes.

MODULARITY The instrumentation, the specification and the monitor are kept strictly separated, and this modular, decoupled design allows one to reuse as much of the runtime verification framework as possible when monitoring new kind of systems. Modularity also allows reasoning on a higher level of abstraction.

EXPRESSIVITY This work showed how complex properties, both data- and control-oriented, can be specified using RML, including context-sensitive ones. Key features from the expressivity point of view are parametric and generic specifications, together with basic computing capabilities.

SEMANTICS RML is still a reasonably compact formalism, and relies on a well-defined, directly implementable rewriting semantics formalized for the lower-level calculus it compiles to.

PRACTICAL USABILITY Throughout this work many examples of different specifications have been given, to show that RML can be actually employed in practice.

Though the actual state of RML provides a solid foundation, further development in different directions is still needed. So far RML has been applied to Node.js[1], Internet of Things systems and Node-RED[2] (Leotta, Ancona, et al., 2018), and current effort is ongoing to apply RML to the Robotic Operating System[3]. Nonetheless, more case studies and experiments will further assess the usability and flexibility of the language, and reveal whether the current set of features is appropriate.

Distributed runtime verification (Francalanza, Pérez, and Sánchez, 2018) poses a whole new set of challenges, and application of RML in this context requires further study. The system-independent flavor of RML lends itself well to the monitoring of heterogeneous, distributed systems, often implemented employing many different programming languages and technologies. It is reasonable to expect that the "global" nature of RML specifications is more suited to the specification of a distributed system as a whole, monitored by a central entity, as opposed to monitor spawning on single entities.

An extremely useful next step would be to enrich the compiler with more static analysis tools that would help the user to avoid subtle errors. When specifications are not correct, finding the source of the problem can be hard,

---

1 https://github.com/RMLatDIBRIS/instrumentation

2 https://nodered.org/

3 https://www.ros.org/

especially because we do not know a priori whether the bug is in the specification or in the program. A necessary condition to employ RML for bigger project is the capability to assist the developer in writing the specification.

In this work some formal properties are proved in order to have a sound foundation for the optimizations needed to keep the complexity manageable. However a deeper study of the expressivity of TC and the algebraic properties of its operators still needs to be done. We conjecture that the language is Turing complete, as the presence of recursion, conditional expressions and counting capabilities gives a very expressive framework to work with.

Another direction for future work is related to the concept of monitorability (Aceto, Achilleos, Francalanza, Ingólfsdóttir, and Lehtinen, 2019a,b; Bartocci et al., 2018), that is, which properties can be not only expressed but also effectively monitored at runtime. Before discussing monitorability, two more steps need to be taken. First, a declarative semantics of TC specifications must be defined (we only gave an operational semantics) so that notions of soundness and completeness of the rewriting procedure (from which the monitor is derived) can be given; this may not be trivial because of our coinductive interpretation and the deterministic semantics. Second, the incremental nature of runtime verification needs to be formalized, i. e., observed prefixes of traces must be given a semantics; in order to do so, more than two truth values would be employed (Bauer, Leucker, and Schallhart, 2007). At the very least, beyond "true" and "false", a third, inconclusive, verdict needs to be considered, to be used when the observed prefix is not informative enough to decide whether the execution is correct with respect to the given specification.

Aceto, Luca, Antonis Achilleos, Adrian Francalanza, and Anna Ingólfs-
dóttir (2017). "Monitoring for Silent Actions." In: *37th IARCS Annual
Conference on Foundations of Software Technology and Theoretical Com-
puter Science, FSTTCS 2017, December 11-15, 2017, Kanpur, India*. Ed. by
Satya V. Lokam and R. Ramanujam. Vol. 93. LIPIcs. Schloss Dagstuhl
- Leibniz-Zentrum fuer Informatik, 7:1–7:14. ISBN: 978-3-95977-055-2.
DOI: 10.4230/LIPIcs.FSTTCS.2017.7. URL: https://doi.org/10.
4230/LIPIcs.FSTTCS.2017.7.

Aceto, Luca, Antonis Achilleos, Adrian Francalanza, and Anna Ingólfs-
dóttir (2018). "A Framework for Parameterized Monitorability." In:
*Foundations of Software Science and Computation Structures - 21st Inter-
national Conference, FOSSACS 2018, Held as Part of the European Joint
Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki,
Greece, April 14-20, 2018, Proceedings*. Ed. by Christel Baier and Ugo
Dal Lago. Vol. 10803. Lecture Notes in Computer Science. Springer,
pp. 203–220. ISBN: 978-3-319-89365-5. DOI: 10.1007/978-3-319-
89366-2_11. URL: https://doi.org/10.1007/978-3-319-89366-
2%5C_11.

Aceto, Luca, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir,
and Sævar Örn Kjartansson (2017). "On the Complexity of Deter-
minizing Monitors." In: *Implementation and Application of Automata -
22nd International Conference, CIAA 2017, Marne-la-Vallée, France, June
27-30, 2017, Proceedings*. Ed. by Arnaud Carayol and Cyril Nicaud.
Vol. 10329. Lecture Notes in Computer Science. Springer, pp. 1–13.
ISBN: 978-3-319-60133-5. DOI: 10.1007/978-3-319-60134-2_1. URL:
https://doi.org/10.1007/978-3-319-60134-2%5C_1.

Aceto, Luca, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir,
and Karoliina Lehtinen (2019a). "Adventures in monitorability: from
branching to linear time and back again." In: *PACMPL* 3.POPL, 52:1–
52:29. DOI: 10.1145/3290365. URL: https://doi.org/10.1145/
3290365.

Aceto, Luca, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir,
and Karoliina Lehtinen (2019b). "An Operational Guide to Moni-
torability." In: *Software Engineering and Formal Methods - 17th Interna-
tional Conference, SEFM 2019, Oslo, Norway, September 18-20, 2019, Pro-
ceedings*. Ed. by Peter Csaba Ölveczky and Gwen Salaün. Vol. 11724.
Lecture Notes in Computer Science. Springer, pp. 433–453. ISBN: 978-
3-030-30445-4. DOI: 10.1007/978-3-030-30446-1_23. URL: https:
//doi.org/10.1007/978-3-030-30446-1%5C_23.

Aceto, Luca, Ian Cassar, Adrian Francalanza, and Anna Ingólfsdóttir
(2018). "On Runtime Enforcement via Suppressions." In: *29th Interna-
tional Conference on Concurrency Theory, CONCUR 2018, September 4-7,*

2018, Beijing, China. Ed. by Sven Schewe and Lijun Zhang. Vol. 118. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 34:1–34:17. ISBN: 978-3-95977-087-3. DOI: 10.4230/LIPIcs.CONCUR.2018.34. URL: https://doi.org/10.4230/LIPIcs.CONCUR.2018.34.

Ahrendt, Wolfgang, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, eds. (2016). *Deductive Software Verification - The KeY Book - From Theory to Practice*. Vol. 10001. Lecture Notes in Computer Science. Springer. ISBN: 978-3-319-49811-9. DOI: 10.1007/978-3-319-49812-6. URL: https://doi.org/10.1007/978-3-319-49812-6.

Ahrendt, Wolfgang, Jesús Mauricio Chimento, Gordon J. Pace, and Gerardo Schneider (2015). "A Specification Language for Static and Runtime Verification of Data and Control Properties." In: *FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings*. Ed. by Nikolaj Bjørner and Frank S. de Boer. Vol. 9109. Lecture Notes in Computer Science. Springer, pp. 108–125. ISBN: 978-3-319-19248-2. DOI: 10.1007/978-3-319-19249-9_8. URL: https://doi.org/10.1007/978-3-319-19249-9%5C_8.

Allan, Chris, Pavel Avgustinov, Aske Simon Christensen, Laurie J. Hendren, Sascha Kuzins, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble (2005). "Adding trace matching with free variables to AspectJ." In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*. Ed. by Ralph E. Johnson and Richard P. Gabriel. ACM, pp. 345–364. ISBN: 1-59593-031-0. DOI: 10.1145/1094811.1094839. URL: https://doi.org/10.1145/1094811.1094839.

Alur, Rajeev, Kousha Etessami, and P. Madhusudan (2004). "A Temporal Logic of Nested Calls and Returns." In: *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*. Ed. by Kurt Jensen and Andreas Podelski. Vol. 2988. Lecture Notes in Computer Science. Springer, pp. 467–481. ISBN: 3-540-21299-X. DOI: 10.1007/978-3-540-24730-2_35. URL: https://doi.org/10.1007/978-3-540-24730-2%5C_35.

Alur, Rajeev, Dana Fisman, and Mukund Raghothaman (2016). "Regular Programming for Quantitative Properties of Data Streams." In: *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Ed. by Peter Thiemann. Vol. 9632. Lecture Notes in Computer Science. Springer, pp. 15–40. ISBN: 978-3-662-49497-4. DOI: 10.1007/978-3-662-49498-1_2. URL: https://doi.org/10.1007/978-3-662-49498-1%5C_2.

Alur, Rajeev, Konstantinos Mamouras, and Caleb Stanford (2019). "Modular quantitative monitoring." In: *PACMPL* 3.POPL, 50:1–50:31. DOI: 10.1145/3290363. URL: https://doi.org/10.1145/3290363.

Ancona, Davide and Andrea Corradi (2014). "Sound and Complete Subtyping between Coinductive Types for Object-Oriented Languages." In: *ECOOP 2014*, pp. 282–307.

Ancona, Davide and Andrea Corradi (2016). "Semantic subtyping for imperative object-oriented languages." In: *OOPSLA 2016*, pp. 568–587.

Ancona, Davide, Francesco Dagnino, and Luca Franceschini (2018). "A formalism for specification of Java API interfaces." In: *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops, ISSTA 2018, Amsterdam, Netherlands, July 16-21, 2018*. Ed. by Julian Dolby, William G. J. Halfond, and Ashish Mishra. ACM, pp. 24–26. ISBN: 978-1-4503-5939-9. DOI: 10.1145/3236454.3236476. URL: https://doi.org/10.1145/3236454.3236476.

Ancona, Davide, Angelo Ferrando, Luca Franceschini, and Viviana Mascardi (2017). "Parametric Trace Expressions for Runtime Verification of Java-Like Programs." In: *Proceedings of the 19th Workshop on Formal Techniques for Java-like Programs, Barcelona , Spain, June 20, 2017*. ACM, 10:1–10:6. ISBN: 978-1-4503-5098-3. DOI: 10.1145/3103111.3104037. URL: https://doi.org/10.1145/3103111.3104037.

Ancona, Davide, Angelo Ferrando, and Viviana Mascardi (2016). "Comparing Trace Expressions and Linear Temporal Logic for Runtime Verification." In: *Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*. Ed. by Erika Ábrahám, Marcello M. Bonsangue, and Einar Broch Johnsen. Vol. 9660. Lecture Notes in Computer Science. Springer, pp. 47–64. ISBN: 978-3-319-30733-6. DOI: 10.1007/978-3-319-30734-3_6. URL: https://doi.org/10.1007/978-3-319-30734-3%5C_6.

Ancona, Davide, Angelo Ferrando, and Viviana Mascardi (2017). "Parametric Runtime Verification of Multiagent Systems." In: *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017, São Paulo, Brazil, May 8-12, 2017*. Ed. by Kate Larson, Michael Winikoff, Sanmay Das, and Edmund H. Durfee. ACM, pp. 1457–1459. URL: http://dl.acm.org/citation.cfm?id=3091328.

Ancona, Davide, Luca Franceschini, Giorgio Delzanno, Maurizio Leotta, Marina Ribaudo, and Filippo Ricca (2017). "Towards Runtime Monitoring of Node.js and Its Application to the Internet of Things." In: *Proceedings First Workshop on Architectures, Languages and Paradigms for IoT, ALP4IoT@iFM 2017, Turin, Italy, September 18, 2017*. Ed. by Danilo Pianini and Guido Salvaneschi. Vol. 264. EPTCS, pp. 27–42. DOI: 10.4204/EPTCS.264.4. URL: https://doi.org/10.4204/EPTCS.264.4.

Ancona, Davide, Luca Franceschini, Angelo Ferrando, and Viviana Mascardi (2019). "A Deterministic Event Calculus for Effective Runtime Verification." In: *Proceedings of the 20th Italian Conference on Theoret-*

ical Computer Science, ICTCS 2019, Como, Italy, September 9-11, 2019. Ed. by Alessandra Cherubini, Nicoletta Sabadini, and Simone Tini. Vol. 2504. CEUR Workshop Proceedings. CEUR-WS.org, pp. 248–260. URL: http://ceur-ws.org/Vol-2504/paper28.pdf.

Atig, Mohamed Faouzi, Ahmed Bouajjani, K. Narayan Kumar, and Prakash Saivasan (2017). "Verification of Asynchronous Programs with Nested Locks." In: *37th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2017, December 11-15, 2017, Kanpur, India.* Ed. by Satya V. Lokam and R. Ramanujam. Vol. 93. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 11:1–11:14. ISBN: 978-3-95977-055-2. DOI: 10.4230/LIPIcs.FSTTCS.2017.11. URL: https://doi.org/10.4230/LIPIcs.FSTTCS.2017.11.

Bar-Hillel, Yehoshua, Micha A. Perles, and Eliahu Shamir (1961). "On Formal Properties of Simple Phrase Structure Grammars." In:

Barringer, Howard, Yliès Falcone, Klaus Havelund, Giles Reger, and David E. Rydeheard (2012). "Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors." In: *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings.* Ed. by Dimitra Giannakopoulou and Dominique Méry. Vol. 7436. Lecture Notes in Computer Science. Springer, pp. 68–84. ISBN: 978-3-642-32758-2. DOI: 10.1007/978-3-642-32759-9_9. URL: https://doi.org/10.1007/978-3-642-32759-9%5C_9.

Barringer, Howard, Allen Goldberg, Klaus Havelund, and Koushik Sen (2004). "Rule-Based Runtime Verification." In: *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VM-CAI 2004, Venice, Italy, January 11-13, 2004, Proceedings.* Ed. by Bernhard Steffen and Giorgio Levi. Vol. 2937. Lecture Notes in Computer Science. Springer, pp. 44–57. ISBN: 3-540-20803-8. DOI: 10.1007/978-3-540-24622-0_5. URL: https://doi.org/10.1007/978-3-540-24622-0%5C_5.

Barringer, Howard and Klaus Havelund (2011). "TraceContract: A Scala DSL for Trace Analysis." In: *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings.* Ed. by Michael J. Butler and Wolfram Schulte. Vol. 6664. Lecture Notes in Computer Science. Springer, pp. 57–72. ISBN: 978-3-642-21436-3. DOI: 10.1007/978-3-642-21437-0_7. URL: https://doi.org/10.1007/978-3-642-21437-0%5C_7.

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

Bartocci, Ezio, Yliès Falcone, Adrian Francalanza, and Giles Reger (2018). "Introduction to Runtime Verification." In: *Lectures on Runtime Verification - Introductory and Advanced Topics.* Ed. by Ezio Bartocci and Yliès Falcone. Vol. 10457. Lecture Notes in Computer Science. Springer,

pp. 1–33. ISBN: 978-3-319-75631-8. DOI: 10.1007/978-3-319-75632-5_1. URL: https://doi.org/10.1007/978-3-319-75632-5%5C_1.

Bauer, Andreas, Martin Leucker, and Christian Schallhart (2006a). "Model-based runtime analysis of distributed reactive systems." In: *17th Australian Software Engineering Conference (ASWEC 2006), 18-21 April 2006, Sydney, Australia*. IEEE Computer Society, pp. 243–252. ISBN: 0-7695-2551-2. DOI: 10.1109/ASWEC.2006.36. URL: https://doi.org/10.1109/ASWEC.2006.36.

Bauer, Andreas, Martin Leucker, and Christian Schallhart (2006b). "Monitoring of Real-Time Properties." In: *FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science, 26th International Conference, Kolkata, India, December 13-15, 2006, Proceedings*. Ed. by S. Arun-Kumar and Naveen Garg. Vol. 4337. Lecture Notes in Computer Science. Springer, pp. 260–272. ISBN: 3-540-49994-6. DOI: 10.1007/11944836_25. URL: https://doi.org/10.1007/11944836%5C_25.

Bauer, Andreas, Martin Leucker, and Christian Schallhart (2007). "The Good, the Bad, and the Ugly, But How Ugly Is Ugly?" In: *Runtime Verification, 7th International Workshop, RV 2007, Vancouver, Canada, March 13, 2007, Revised Selected Papers*. Ed. by Oleg Sokolsky and Serdar Tasiran. Vol. 4839. Lecture Notes in Computer Science. Springer, pp. 126–138. ISBN: 978-3-540-77394-8. DOI: 10.1007/978-3-540-77395-5_11. URL: https://doi.org/10.1007/978-3-540-77395-5%5C_11.

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

Bodden, Eric, Laurie J. Hendren, and Ondrej Lhoták (2007). "A Staged Static Program Analysis to Improve the Performance of Runtime Monitoring." In: *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*. Ed. by Erik Ernst. Vol. 4609. Lecture Notes in Computer Science. Springer, pp. 525–549. ISBN: 978-3-540-73588-5. DOI: 10.1007/978-3-540-73589-2_25. URL: https://doi.org/10.1007/978-3-540-73589-2%5C_25.

Bollig, Benedikt, Normann Decker, and Martin Leucker (2012). "Frequency Linear-time Temporal Logic." In: *Sixth International Symposium on Theoretical Aspects of Software Engineering, TASE 2012, 4-6 July 2012, Beijing, China*. Ed. by Tiziana Margaria, Zongyan Qiu, and Hongli Yang. IEEE Computer Society, pp. 85–92. ISBN: 978-0-7695-4751-0. DOI: 10.1109/TASE.2012.43. URL: https://doi.org/10.1109/TASE.2012.43.

Brookes, Stephen D., C. A. R. Hoare, and A. W. Roscoe (1984). "A Theory of Communicating Sequential Processes." In: *J. ACM* 31.3, pp. 560–599. DOI: 10.1145/828.833. URL: https://doi.org/10.1145/828.833.

Brörkens, Mark and Michael Möller (2002). "Dynamic Event Generation for Runtime Checking using the JDI." In: *Electr. Notes Theor. Comput. Sci.* 70.4, pp. 21–35. DOI: 10.1016/S1571-0661(04)80575-9. URL: https://doi.org/10.1016/S1571-0661(04)80575-9.

Cau, Antonio and Hussein Zedan (1997). "Refining Interval Temporal Logic Specifications." In: *Transformation-Based Reactive Systems Development, 4th International AMAST Workshop on Real-Time Systems and Concurrent and Distributed Software, ARTS'97, Palma, Mallorca, Spain, May 21-23, 1997, Proceedings*. Ed. by Miquel Bertran and Teodor Rus. Vol. 1231. Lecture Notes in Computer Science. Springer, pp. 79–94. ISBN: 3-540-63010-4. DOI: 10.1007/3-540-63010-4_6. URL: https://doi.org/10.1007/3-540-63010-4%5C_6.

Chen, Feng and Grigore Rosu (2005). "Java-MOP: A Monitoring Oriented Programming Environment for Java." In: *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*. Ed. by Nicolas Halbwachs and Lenore D. Zuck. Vol. 3440. Lecture Notes in Computer Science. Springer, pp. 546–550. ISBN: 3-540-25333-5. DOI: 10.1007/978-3-540-31980-1_36. URL: https://doi.org/10.1007/978-3-540-31980-1%5C_36.

Chen, Feng and Grigore Rosu (2007). "Mop: an efficient and generic runtime verification framework." In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. Ed. by Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr. ACM, pp. 569–588. ISBN: 978-1-59593-786-5. DOI: 10.1145/1297027.1297069. URL: https://doi.org/10.1145/1297027.1297069.

Chen, Feng and Grigore Rosu (2009). "Parametric Trace Slicing and Monitoring." In: *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*. Ed. by Stefan Kowalewski and Anna Philippou. Vol. 5505. Lecture Notes in Computer Science. Springer, pp. 246–261. ISBN: 978-3-642-00767-5. DOI: 10.1007/978-3-642-00768-2_23. URL: https://doi.org/10.1007/978-3-642-00768-2%5C_23.

Chimento, Jesús Mauricio, Wolfgang Ahrendt, Gordon J. Pace, and Gerardo Schneider (2015). "StaRVOOrS: A Tool for Combined Static and Runtime Verification of Java." In: *Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings*. Ed. by Ezio Bartocci and Rupak Majumdar. Vol. 9333. Lecture Notes in Computer Science. Springer, pp. 297–305. ISBN: 978-3-319-23819-7. DOI: 10.1007/978-3-319-23820-3_21. URL: https://doi.org/10.1007/978-3-319-23820-3%5C_21.

Cok, David R. (2011). "OpenJML: JML for Java 7 by Extending Open-JDK." In: *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*. Ed. by Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi. Vol. 6617. Lecture Notes in Computer Science. Springer, pp. 472–479. ISBN: 978-3-642-20397-8. DOI: 10.1007/978-3-642-20398-5_35. URL: https://doi.org/10.1007/978-3-642-20398-5%5C_35.

Colombo, Christian, Gordon J. Pace, and Gerardo Schneider (2008). "Dynamic Event-Based Runtime Monitoring of Real-Time and Contextual Properties." In: *Formal Methods for Industrial Critical Systems, 13th International Workshop, FMICS 2008, L'Aquila, Italy, September 15-16, 2008, Revised Selected Papers*. Ed. by Darren D. Cofer and Alessandro Fantechi. Vol. 5596. Lecture Notes in Computer Science. Springer, pp. 135–149. ISBN: 978-3-642-03239-4. DOI: 10.1007/978-3-642-03240-0_13. URL: https://doi.org/10.1007/978-3-642-03240-0%5C_13.

Colombo, Christian, Gordon J. Pace, and Gerardo Schneider (2009). "LARVA — Safer Monitoring of Real-Time Java Programs (Tool Paper)." In: *Seventh IEEE International Conference on Software Engineering and Formal Methods, SEFM 2009, Hanoi, Vietnam, 23-27 November 2009*. Ed. by Dang Van Hung and Padmanabhan Krishnan. IEEE Computer Society, pp. 33–37. ISBN: 978-0-7695-3870-9. DOI: 10.1109/SEFM.2009.13. URL: https://doi.org/10.1109/SEFM.2009.13.

Convent, Lukas, Sebastian Hungerecker, Martin Leucker, Torben Scheffel, Malte Schmitz, and Daniel Thoma (2018). "TeSSLa: Temporal Stream-based Specification Language." In: *CoRR* abs/1808.10717. arXiv: 1808.10717. URL: http://arxiv.org/abs/1808.10717.

Courcelle, Bruno (1983). "Fundamental Properties of Infinite Trees." In: *Theor. Comput. Sci.* 25, pp. 95–169. DOI: 10.1016/0304-3975(83)90059-2. URL: https://doi.org/10.1016/0304-3975(83)90059-2.

D'Angelo, Ben, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, and Zohar Manna (2005). "LOLA: Runtime Monitoring of Synchronous Systems." In: *12th International Symposium on Temporal Representation and Reasoning (TIME 2005), 23-25 June 2005, Burlington, Vermont, USA*. IEEE Computer Society, pp. 166–174. ISBN: 0-7695-2370-6. DOI: 10.1109/TIME.2005.26. URL: https://doi.org/10.1109/TIME.2005.26.

De Boer, Frank and Stijn De Gouw (2014). "Combining Monitoring with Run-Time Assertion Checking." In: *Formal Methods for Executable Software Models - 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2014, Bertinoro, Italy, June 16-20, 2014, Advanced Lectures*. Ed. by Marco Bernardo, Ferruccio Damiani, Reiner Hähnle, Einar Broch Johnsen, and Ina Schaefer. Vol. 8483. Lecture Notes in Computer Science. Springer, pp. 217–

262. ISBN: 978-3-319-07316-3. DOI: 10.1007/978-3-319-07317-0_6. URL: https://doi.org/10.1007/978-3-319-07317-0%5C_6.

Delgado, Nelly, Ann Q. Gates, and Steve Roach (2004). "A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools." In: *IEEE Trans. Software Eng.* 30.12, pp. 859–872. DOI: 10.1109/TSE.2004.91. URL: https://doi.org/10.1109/TSE.2004.91.

Engberg, Uffe and Mogens Nielsen (May 1986). "A Calculus of Communicating Systems with Label Passing." In: *DAIMI Report Series* 15.208. DOI: 10.7146/dpb.v15i208.7559. URL: https://tidsskrift.dk/daimipb/article/view/7559.

Falcone, Yliès, Klaus Havelund, and Giles Reger (2013). "A Tutorial on Runtime Verification." In: *Engineering Dependable Software Systems*. Ed. by Manfred Broy, Doron A. Peled, and Georg Kalus. Vol. 34. NATO Science for Peace and Security Series, D: Information and Communication Security. IOS Press, pp. 141–175. ISBN: 978-1-61499-206-6. DOI: 10.3233/978-1-61499-207-3-141. URL: https://doi.org/10.3233/978-1-61499-207-3-141.

Falcone, Yliès, Srdan Krstic, Giles Reger, and Dmitriy Traytel (2018). "A Taxonomy for Classifying Runtime Verification Tools." In: *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings*. Ed. by Christian Colombo and Martin Leucker. Vol. 11237. Lecture Notes in Computer Science. Springer, pp. 241–262. ISBN: 978-3-030-03768-0. DOI: 10.1007/978-3-030-03769-7_14. URL: https://doi.org/10.1007/978-3-030-03769-7%5C_14.

Faymonville, Peter, Bernd Finkbeiner, Sebastian Schirmer, and Hazem Torfah (2016). "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, pp. 152–168. ISBN: 978-3-319-46981-2. DOI: 10.1007/978-3-319-46982-9_10. URL: https://doi.org/10.1007/978-3-319-46982-9%5C_10.

Forgy, Charles (1982). "Rete: A Fast Algorithm for the Many Patterns/Many Objects Match Problem." In: *Artif. Intell.* 19.1, pp. 17–37. DOI: 10.1016/0004-3702(82)90020-0. URL: https://doi.org/10.1016/0004-3702(82)90020-0.

Francalanza, Adrian (2016). "A Theory of Monitors - (Extended Abstract)." In: *Foundations of Software Science and Computation Structures - 19th International Conference, FOSSACS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Ed. by Bart Jacobs and Christof Löding. Vol. 9634. Lecture Notes in Computer Science. Springer, pp. 145–161. ISBN: 978-3-662-49629-9. DOI: 10.1007/978-3-662-49630-5_9. URL: https://doi.org/10.1007/978-3-662-49630-5%5C_9.

Francalanza, Adrian (2017). "Consistently-Detecting Monitors." In: *28th International Conference on Concurrency Theory, CONCUR 2017, September 5-8, 2017, Berlin, Germany*. Ed. by Roland Meyer and Uwe Nestmann. Vol. 85. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 8:1–8:19. ISBN: 978-3-95977-048-4. DOI: `10.4230/LIPIcs.CONCUR.2017.8`. URL: `https://doi.org/10.4230/LIPIcs.CONCUR.2017.8`.

Francalanza, Adrian, Luca Aceto, and Anna Ingólfsdóttir (2017). "Monitorability for the Hennessy-Milner logic with recursion." In: *Formal Methods in System Design* 51.1, pp. 87–116. DOI: `10.1007/s10703-017-0273-z`. URL: `https://doi.org/10.1007/s10703-017-0273-z`.

Francalanza, Adrian, Jorge A. Pérez, and César Sánchez (2018). "Runtime Verification for Decentralised and Distributed Systems." In: *Lectures on Runtime Verification: Introductory and Advanced Topics*. Ed. by Ezio Bartocci and Yliès Falcone. Cham: Springer International Publishing, pp. 176–210. ISBN: 978-3-319-75632-5. DOI: `10.1007/978-3-319-75632-5_6`. URL: `https://doi.org/10.1007/978-3-319-75632-5_6`.

Franceschini, Luca (2019). "RML: runtime monitoring language: a system-agnostic DSL for runtime verification." In: *Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming, Genova, Italy, April 1-4, 2019*. ACM, 28:1–28:3. ISBN: 978-1-4503-6257-3. DOI: `10.1145/3328433.3328462`. URL: `https://doi.org/10.1145/3328433.3328462`.

Frisch, A., G. Castagna, and V. Benzaken (2008). "Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types." In: *J. ACM* 55.4.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0201633612.

Halbwachs, Nicolas, Fabienne Lagnier, and Christophe Ratel (1992). "Programming and Verifying Real-Time Systems by Means of the Synchronous Data-Flow Language LUSTRE." In: *IEEE Trans. Software Eng.* 18.9, pp. 785–793. DOI: `10.1109/32.159839`. URL: `https://doi.org/10.1109/32.159839`.

Havelund, Klaus (2015). "Rule-based runtime verification revisited." In: *STTT* 17.2, pp. 143–170. DOI: `10.1007/s10009-014-0309-2`. URL: `https://doi.org/10.1007/s10009-014-0309-2`.

Havelund, Klaus and Allen Goldberg (2005). "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, pp. 374–383. ISBN: 978-3-540-69147-1. DOI: `10.1007/978-3-540-69149-5\_40`. URL: `https://doi.org/10.1007/978-3-540-69149-5%5C_40`.

Hoare, C. A. R. (1969). "An Axiomatic Basis for Computer Programming." In: *Commun. ACM* 12.10, pp. 576–580. DOI: 10.1145/363235.363259. URL: https://doi.org/10.1145/363235.363259.

Hoenicke, Jochen (2006). "Combination of processes, data, and time." PhD thesis. Carl von Ossietzky University of Oldenburg. URL: http://d-nb.info/981576087.

Hopcroft, John E., Rajeev Motwani, and Jeffrey D. Ullman (2007). *Introduction to automata theory, languages, and computation, 3rd Edition*. Pearson international edition. Addison-Wesley. ISBN: 978-0-321-47617-3.

Johnson, Ralph E. and Richard P. Gabriel, eds. (2005). *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*. ACM. ISBN: 1-59593-031-0.

Kahlon, Vineet, Franjo Ivancic, and Aarti Gupta (2005). "Reasoning About Threads Communicating via Locks." In: *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*. Ed. by Kousha Etessami and Sriram K. Rajamani. Vol. 3576. Lecture Notes in Computer Science. Springer, pp. 505–518. ISBN: 3-540-27231-3. DOI: 10.1007/11513988_49. URL: https://doi.org/10.1007/11513988%5C_49.

Kiczales, Gregor, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold (2001). "An Overview of AspectJ." In: *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings*. Ed. by Jørgen Lindskov Knudsen. Vol. 2072. Lecture Notes in Computer Science. Springer, pp. 327–353. ISBN: 3-540-42206-4. DOI: 10.1007/3-540-45337-7_18. URL: https://doi.org/10.1007/3-540-45337-7%5C_18.

Kiczales, Gregor, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin (1997). "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, pp. 220–242. ISBN: 3-540-63089-9. DOI: 10.1007/BFb0053381. URL: https://doi.org/10.1007/BFb0053381.

Klint, Paul, Tijs van der Storm, and Jurgen J. Vinju (2009). "RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation." In: *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, Edmonton, Alberta, Canada, September 20-21, 2009*. IEEE Computer Society, pp. 168–177. ISBN: 978-0-7695-3793-1. DOI: 10.1109/SCAM.2009.28. URL: https://doi.org/10.1109/SCAM.2009.28.

Kozen, Dexter (1983). "Results on the Propositional mu-Calculus." In: *Theor. Comput. Sci.* 27, pp. 333–354. DOI: 10.1016/0304-3975(82)90125-6. URL: https://doi.org/10.1016/0304-3975(82)90125-6.

Larsen, Kim Guldstrand (1990). "Proof Systems for Satisfiability in Hennessy-Milner Logic with Recursion." In: *Theor. Comput. Sci.* 72.2&3, pp. 265–288. DOI: 10.1016/0304-3975(90)90038-J. URL: https://doi.org/10.1016/0304-3975(90)90038-J.

Leavens, Gary T., Albert L. Baker, and Clyde Ruby (2006). "Preliminary design of JML: a behavioral interface specification language for java." In: *ACM SIGSOFT Software Engineering Notes* 31.3, pp. 1–38. DOI: 10.1145/1127878.1127884. URL: https://doi.org/10.1145/1127878.1127884.

Leotta, Maurizio, Davide Ancona, Luca Franceschini, Dario Olianas, Marina Ribaudo, and Filippo Ricca (2018). "Towards a Runtime Verification Approach for Internet of Things Systems." In: *Current Trends in Web Engineering - ICWE 2018 International Workshops, MATWEP, EnWot, KD-WEB, WEOD, TourismKG, Cáceres, Spain, June 5, 2018, Revised Selected Papers*. Ed. by Cesare Pautasso, Fernando Sánchez-Figueroa, Kari Systä, and Juan Manuel Murillo Rodriguez. Vol. 11153. Lecture Notes in Computer Science. Springer, pp. 83–96. ISBN: 978-3-030-03055-1. DOI: 10.1007/978-3-030-03056-8_8. URL: https://doi.org/10.1007/978-3-030-03056-8%5C_8.

Leotta, Maurizio, Diego Clerissi, Luca Franceschini, Dario Olianas, Davide Ancona, Filippo Ricca, and Marina Ribaudo (2019). "Comparing Testing and Runtime Verification of IoT Systems: A Preliminary Evaluation based on a Case Study." In: *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2019, Heraklion, Crete, Greece, May 4-5, 2019*. Ed. by Ernesto Damiani, George Spanoudakis, and Leszek A. Maciaszek. SciTePress, pp. 434–441. ISBN: 978-989-758-375-9. DOI: 10.5220/0007745604340441. URL: https://doi.org/10.5220/0007745604340441.

Leucker, Martin, César Sánchez, Torben Scheffel, Malte Schmitz, and Alexander Schramm (2018). "TeSSLa: runtime verification of non-synchronized real-time streams." In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018*. Ed. by Hisham M. Haddad, Roger L. Wainwright, and Richard Chbeir. ACM, pp. 1925–1933. DOI: 10.1145/3167132.3167338. URL: https://doi.org/10.1145/3167132.3167338.

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

Lokam, Satya V. and R. Ramanujam, eds. (2018). *37th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2017, December 11-15, 2017, Kanpur, India*. Vol. 93. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. ISBN: 978-3-95977-055-2. URL: http://www.dagstuhl.de/dagpub/978-3-95977-055-2.

Martin, Michael C., V. Benjamin Livshits, and Monica S. Lam (2005). "Finding application errors and security flaws using PQL: a program query language." In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*. Ed. by Ralph E. Johnson and Richard P. Gabriel. ACM, pp. 365–383. ISBN: 1-59593-031-0. DOI: 10.1145/1094811.1094840. URL: https://doi.org/10.1145/1094811.1094840.

Meyer, Bertrand (1992). "Applying "Design by Contract"." In: *IEEE Computer* 25.10, pp. 40–51. DOI: 10.1109/2.161279. URL: https://doi.org/10.1109/2.161279.

Milner, Robin (1980). *A Calculus of Communicating Systems*. Vol. 92. Lecture Notes in Computer Science. Springer. ISBN: 3-540-10235-3. DOI: 10.1007/3-540-10235-3. URL: https://doi.org/10.1007/3-540-10235-3.

Pandya, Paritosh K. (2000). *Specifying and Deciding Quantified Discrete-time Duration Calculus Formulae using DCVALID*. Tech. rep.

Parr, Terence John and Russell W. Quong (1995). "ANTLR: A Predicated-$LL(k)$ Parser Generator." In: *Softw., Pract. Exper.* 25.7, pp. 789–810. DOI: 10.1002/spe.4380250705. URL: https://doi.org/10.1002/spe.4380250705.

Pnueli, Amir (1977). "The Temporal Logic of Programs." In: *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, pp. 46–57. DOI: 10.1109/SFCS.1977.32. URL: https://doi.org/10.1109/SFCS.1977.32.

Pnueli, Amir and Aleksandr Zaks (2006). "PSL Model Checking and Run-Time Verification Via Testers." In: *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*. Ed. by Jayadev Misra, Tobias Nipkow, and Emil Sekerinski. Vol. 4085. Lecture Notes in Computer Science. Springer, pp. 573–586. ISBN: 3-540-37215-6. DOI: 10.1007/11813040_38. URL: https://doi.org/10.1007/11813040%5C_38.

Pnueli, Amir and Lenore D. Zuck (1993). "In and Out of Temporal Logic." In: *Proceedings of the Eighth Annual Symposium on Logic in Computer Science (LICS '93), Montreal, Canada, June 19-23, 1993*. IEEE Computer Society, pp. 124–135. ISBN: 0-8186-3140-6. DOI: 10.1109/LICS.1993.287594. URL: https://doi.org/10.1109/LICS.1993.287594.

Prior, Arthur Norman (1957). *Time and Modality*. John Locke Lecture.

Reger, Giles, Helena Cuenca Cruz, and David E. Rydeheard (2015). "MarQ: Monitoring at Runtime with QEA." In: *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Ed. by Christel Baier and Cesare Tinelli. Vol. 9035. Lecture

Notes in Computer Science. Springer, pp. 596–610. ISBN: 978-3-662-46680-3. DOI: 10.1007/978-3-662-46681-0_55. URL: https://doi.org/10.1007/978-3-662-46681-0%5C_55.

Schiavio, Filippo, Haiyang Sun, Daniele Bonetta, Andrea Rosà, and Walter Binder (2019). "NodeMOP: runtime verification for Node.js applications." In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, SAC 2019, Limassol, Cyprus, April 8-12, 2019*. Ed. by Chih-Cheng Hung and George A. Papadopoulos. ACM, pp. 1794–1801. ISBN: 978-1-4503-5933-7. DOI: 10.1145/3297280.3297456. URL: https://doi.org/10.1145/3297280.3297456.

Sen, Koushik, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs (2013). "Jalangi: a selective record-replay and dynamic analysis framework for JavaScript." In: *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*. Ed. by Bertrand Meyer, Luciano Baresi, and Mira Mezini. ACM, pp. 488–498. ISBN: 978-1-4503-2237-9. DOI: 10.1145/2491411.2491447. URL: https://doi.org/10.1145/2491411.2491447.

Simon, Luke, Ajay Mallya, Ajay Bansal, and Gopal Gupta (2006). "Coinductive Logic Programming." In: *Logic Programming, 22nd International Conference, ICLP 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*. Ed. by Sandro Etalle and Miroslaw Truszczynski. Vol. 4079. Lecture Notes in Computer Science. Springer, pp. 330–345. ISBN: 3-540-36635-0. DOI: 10.1007/11799573_25. URL: https://doi.org/10.1007/11799573%5C_25.

Sokolsky, Oleg, Klaus Havelund, and Insup Lee (2012). "Introduction to the special section on runtime verification." In: *STTT* 14.3, pp. 243–247. DOI: 10.1007/s10009-011-0218-6. URL: https://doi.org/10.1007/s10009-011-0218-6.

Thati, Prasanna and Grigore Rosu (2005). "Monitoring Algorithms for Metric Temporal Logic Specifications." In: *Electr. Notes Theor. Comput. Sci.* 113, pp. 145–162. DOI: 10.1016/j.entcs.2004.01.029. URL: https://doi.org/10.1016/j.entcs.2004.01.029.

DECLARATION

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements.

Genova, Italy, March 2020

Luca Franceschini