Runtime Verification Logics
A Language Design Perspective

Klaus Havelund[1⋆] and Giles Reger[2⋆⋆]

[1] Jet Propulsion Laboratory, California Inst. of Technology, USA
[2] University of Manchester, Manchester, UK

Abstract. Runtime Verification is a light-weight approach to systems verification, where actual executions of a system are processed and analyzed using rigorous techniques. In this paper we shall narrow the term's definition to represent the commonly studied variant consisting of verifying that a single system execution conforms to a specification written in a formal specification language. Runtime verification (in this sense) can be used for writing test oracles during testing when the system is too complex for full formal verification, or it can be used during deployment of the system as part of a fault protection strategy, where corrective actions may be taken in case the specification is violated. Specification languages for runtime verification appear to differ from for example temporal logics applied in model checking, in part due to the focus on monitoring of events that carry data, and specifically due to the desire to relate data values existing at different time points, resulting in new challenges in both the complexity of the monitoring approach and the expressiveness of languages. Over the recent years, numerous runtime verification specification languages have emerged, each with its different features and levels of expressiveness and usability. This paper presents an overview and a discussion of this design space.

1 Introduction

Runtime Verification (RV) [31, 48] is narrowly viewed[3] as the process of monitoring and checking the runtime behavior of a system, from here on referred to as the System Being Monitored (SBM), against a formal specification. RV can be applied for safety, security, and comprehension purposes. The SBM must emit an event stream (via instrumentation or otherwise), the execution trace, which is then consumed by a monitor, which as a secondary input takes a formal specification. RV can be applied in online mode, where the monitor executes at the same time as the SBM, tracking its moves step

⋆ The research performed by this author was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

⋆⋆ The work of this author is related to COST Action ARVI IC1402, supported by COST (European Cooperation in Science and Technology).

[3] RV more broadly includes such topics as checking traces with algorithms, learning specifications including statistical information from traces, trace visualization, program instrumentation, and fault protection.

by step, or it can be applied in offline mode to a log produced by the SBM. Orthogonally, RV can be applied before deployment of the software, for example as part of the testing process, or after deployment, for example as part of a fault protection strategy, where the monitor can influence the behavior of the SBM. In this case, the monitor will usually be run in online mode. The monitor in the simplest case will produce a true/-false verdict, but can be more informative, and produce richer information about the trace seen so far.

To be effective, a runtime verification method requires an *expressive specification language* to capture properties of interest, an *elegant specification language* allowing specifications to be succinct and easy to write and read, and an *efficient monitoring algorithm* to ensure that monitoring does not impede the running of the monitored system. In this paper we shall focus on the former two (although efficiency will be discussed as it does indeed influence the design space), and investigate what we consider the most common variations of specification languages for RV. Numerous such specification languages have been developed in recent time. These are usually based on well known concepts such as e.g. state machines, regular expressions, temporal logics (past time as well as future time), timed logics, context free grammars, variations of the μ-calculus, rule-based systems, stream processing, and process algebras.

A big emphasis over the last decade has been on data parameterized logics, suited for monitoring sequences of events carrying data parameters (named records). Temporal logics applied in model checking (MC) [39] have to some extent allowed data as well. However, state of the art RV logics tend to support relating data across time points in a manner not as commonly supported in temporal logics for MC (although instances exist, e.g. [1]). To illustrate this, assume that we analyze finite traces (logs), and that we operate with a variant of LTL [53] with a finite trace semantics. Consider the following classical MC formula: $\Box(p \rightarrow \Diamond q)$, meaning if p is true in a position in the trace, then q must be true at a later point in that finite trace. In a system such as SPIN, it is possible to associate expressions over the state to the propositions p and q, for example $p = x \geq 0$ and $q = y \geq 0$. Expanding these names in the formula, we get the formula: $\Box(x \geq 0 \rightarrow \Diamond y \geq 0)$. This formula refers to data. However, to monitor such a formula (on a finite trace) requires a memory of only 1 bit, raised iff. $x \geq 0$ has been observed true and $y \geq 0$ has not yet been observed true when analyzing the trace from left to right. Consider now a different formula, expressing that whenever $x \geq 0$ and has a value k then y should eventually obtain that value: $\forall k \, \Box((x \geq 0 \wedge x = k) \rightarrow \Diamond y = k)$. This property can be very costly to monitor since the monitor from any point where $x \geq 0$ will have to remember the value k of x until y catches up.

Due to the nature of RV where only a single trace is examined, it is considered possible to allow very expressive specification languages, in contrast to static analysis, where expressiveness of the specification language normally is considered in conflict with degree of automation achievable. It is this perceived freedom to explore richer logics that have caused RV logics to incorporate data on a larger scale. It is, however, not the case that RV logics need to be so different from for example MC logics. RV logics can fundamentally focus on finite traces (safety properties), whereas MC logics must handle infinite traces (safety and liveness properties). But beyond this point, the two classes of logics could in principle have a very large intersection. Runtime verifica-

tion can be seen as exploring new branches of logics also potentially useful for model checking.

This paper presents a discussion of some of the design space for state-of-the-art RV logics, that we have found of general interest. The presentation is split into a discussion of core temporal constructs without considering data (although data occur), followed by considerations of how to deal with data. The discussion is in part based on property examples and their specifications produced by participants of two recent runtime verification competitions, CRV (Competition on Runtime Verification) 2014 [8] and CRV 2015 [32]. Participants used their favorite specification language to specify a set of shared properties proposed by the participants. In this study we inspected properties submitted by the developers of MARQ [55], LOGFIRE [37], Larva [22], JAVAMOP [51], JUnitRV [24], Monpoly [10], and Solist [17]. The paper focuses due to lack of space specifically on state machines, regular expressions and temporal logics, since these are the most commonly seen. This leaves out RV systems for such formalisms as context free grammars [51], variations of the μ-calculus [4], rule-based systems [6, 37], stream processing [23], and process algebras [7], all of which are quite interesting alternatives. As the focus of this paper is on the usability of specification languages, we will often make use of ASCII representations of specifications in different formalisms. However, where the focus is not on usability, but on some other feature of the language, we will use more convenient formalisms such as graphical automata and mathematical formula.

The paper is organized as follows. We begin with a brief summary of the main elements considered of importance in runtime verification (Section 2). We then present parts of the design space for propositional logics ignoring data (Section 3) and then with data (Section 4). We conclude with a summary of our findings (Section 5).

2 Fundamentals of Runtime Verification

To set the scene we briefly recall what we mean by runtime verification (RV) in this paper and, therefore, what a specification language for RV involves. In runtime verification we *abstract* an executing system being monitored (SBM) as a sequence of discrete observations, also referred to as a *trace*. We call these observations *events*. Commonly events are either propositional names or named data records i.e. a pair of a name and a list of data values. Events can be produced directly by the system, or extracted by code instrumentation: special code pieces inserted in the executing code, either manually or using some form of automated code instrumentation software, for example aspect-oriented programming technology [41]. In the case of offline monitoring, the trace will be finite. In the case of online monitoring, the executing system may be theoretically non-terminating. However, even in this case any monitor will at any time have to rely only on a finite set of observations - a finite *prefix* of the theoretically infinite execution.

We shall refer to a desired behavior of a system to be monitored as a *property*. Let Γ denote the set of all possible traces. A property is abstractly seen as a subset $\mathcal{P} \subseteq \Gamma$ of traces, namely the traces that we say satisfy (belong to) the property. We shall usually describe properties in informal English, and then formalize them in a specification language. A specification language allows us to formally define $\mathcal{P}$ via a textual speci-

fication φ. The property (set of traces) represented by a specification φ is denoted by $\mathcal{P}(\varphi)$. The monitoring problem is then to check whether a particular trace τ belongs to this set i.e. to check $\tau \in \mathcal{P}(\varphi)$. This is often referred to as *matching* the trace against the property. Note that specifications can be provided in negative form as discussed below. For this reason we need to distinguish between the language $\mathcal{L}(\varphi)$ denoted by a specification, which is a very straight forward definition, and the property $\mathcal{P}(\varphi)$ denoted by the specification, defined in terms of $\mathcal{L}(\varphi)$. This will be clarified in the following. A number of concerns must be addressed in any RV system, as discussed below.

Polarity A specification φ may specify the *good* (desired) behavior or the *bad* (undesired) behavior. In the positive case $\mathcal{P}(\varphi) = \mathcal{L}(\varphi)$. In the negative case $\mathcal{P}(\varphi) = \Gamma \setminus \mathcal{L}(\varphi)$. In the former case matching represents *validation* and in the latter it represents *violation* of the property. This choice can have an impact on the readability of a specification. For example, consider the following *UnsafeMapIterator* property about JAVA collection objects.

Property 1 (UnsafeMapIterator). Given a map object m, collection object c, and iterator object i, if c is created from m (c is for example the set of m's key values), and i is created from c, and later m is updated, then i should not be used any further. We use the event $\texttt{create}(x, y)$ to indicate that object y is created from object x, $\texttt{use}(i)$ to indicate that iterator i is used for iteration, and $\texttt{update}(m)$ to indicate that map m is updated.

A positive formulation of this property using a data parameterized regular expression[4] (note that the main emphasis is not on data here) could be:

$$\Lambda m, c, i : \texttt{create}(m, c).\texttt{update}(m)^*.\texttt{create}(c, i).\texttt{use}(i)^*.\texttt{update}(m)^* \quad (1)$$

This property states the sequence of events that are allowed (for a map m, collection c and iterator i). Most notably, this sequence disallows an use event occurring after a map update. The positive formulation needs to capture all acceptable behaviors. A negative formulation could be:

$$\exists m, c, i : \texttt{create}(m, c).\texttt{create}(c, i).\texttt{update}(m).\texttt{use}(i) \quad (2)$$

Where we take a non-standard *skip* semantics (see Section 3.1) that skips any event that does not match the next expected event. In the negative formulation it suffices to describe the sequence of events required to lead to failure, which in some cases can be simpler. Therefore, there is an argument for allowing for both positive and negative formulations even if the underlying language is closed under negation.

Where to Match In the previous example we matched the total trace against the formulas, that is checking $\tau \in \mathcal{L}(\varphi)$ in the positive case and $\tau \in \Gamma \setminus \mathcal{L}(\varphi)$ in the negative case. This is referred to as *total* matching, and is the most common approach. An alternative is to perform *suffix* matching, first proposed in [2], where a trace belongs to

[4] Here $\Lambda m, c, i$ is related to trace-slicing (see Section 4.4) and has the meaning that the property should hold *for all* subtraces projected on possible values for m, c, i.

the property denoted by the specification if a suffix of the trace belongs to the language of the specification. That is: $\mathcal{P}(\varphi) = \{\sigma.\tau \mid \tau \in \mathcal{L}(\varphi), \sigma \in \Gamma\}$. To see how this can improve readability consider the following property, also about JAVA objects.

Property 2 (HasNextIterator). For every iterator object i, a call to `next` must be preceded by a call to `hasNext` returning `true`, without any other `next` calls occurring in between.

The following two specifications of this property are negative (the undesired case). One (left) uses total-matching on the whole trace, and the other (right) uses suffix-matching.

$$\Lambda i : (\texttt{hasNext}(i, \texttt{true})^+.\texttt{next}(i))^*.\texttt{next}(i) \qquad\qquad \Lambda i : (\epsilon \mid \texttt{next}(i)).\texttt{next}(i)$$

Where ϵ denotes the start of the trace. Suffix-matching also allows us to write the slightly more concise $\Lambda i : \texttt{next}(i).\texttt{next}(i)$, which is not quite equivalent as it misses the case where the trace begins with $\texttt{next}(i)$. Suffix matching is typically combined with negatively formulated regular expressions.

Finite versus Infinite Traces When monitoring a trace produced by a system, at any point in time the trace observed so far will be finite. This means that the semantics of runtime verification logics should deal with finite traces. Finite state machines and regular expressions are typically defined over *finite* traces. However, traditionally, temporal logics applied in, for example, model checking, are defined over infinite traces, and such temporal logics must be adapted to the finite trace scenario, re-defining their semantics, when applied in a runtime verification context. One such approach, which for example is applicable to off-line log file analysis, is to handle obligations such as $\Diamond p$ as false on a finite trace where p never occurs. Hence the result of evaluating a temporal formula on a trace is either true or false, as in the case of finite state machines and regular expressions (language membership). A different approach, applicable to online monitoring, consists of viewing a finite trace as a prefix of some infinite trace. At each time point the current verdict depends on whether the finite trace observed so far potentially can be extended to a satisfying (finite or infinite) trace. This naturally leads one to go beyond the true and false verdicts, and introduce additional verdicts, such as "so far true" and "so far false", for cases where there are both satisfying and violating extensions. The reader is invited to consult [14, 31] for further discussions.

Safety and Co-Safety Properties A *safety* property intuitively captures the notion that nothing bad happens [44]. The languages of such properties are prefix-closed since if a trace is safe then all of its prefixes must be safe. A consequence of this is that safety properties can be falsified by a finite prefix of a trace i.e. there can be a point before the end of the trace where it is known that the property has been falsified. Conversely, *co-safety* properties capture the notion that something good happens, are extension-closed[5], and can be *validated* by a finite prefix. Properties may be neither safety or co-safety properties but may share qualities with both classes. A *response* property of

[5] A language is extension closed if whenever τ is in the language then so is $\tau.\sigma$ for any σ.

the form *"whenever A happens B should eventually happen"* is an example of a property that is neither, and can not be decided by a prefix of a trace. One may therefore deem such a property non-monitorable. However, in the case of offline monitoring, where one checks a finite trace that is not extended, it is possible to give a very precise true/false semantics to such a formula. Classes of monitorable properties are discussed further in [30, 15].

Beyond Language Inclusion So far we have mostly discussed logics for checking Boolean satisfaction in the form of: $\tau \in \mathcal{P}(\varphi)$. We have, however, briefly mentioned extending the Boolean verdict domain $\{true, false\}$ with values such as "so far true" and "so far false" [14, 15], in some work unified into a "so far unknown" ?-result [51]. The full generalization of the Boolean result domain is any data domain D considered useful. For example, a logic could be designed for computing statistical information as to how well the trace satisfies a property, or even producing user-defined computations over the trace. Collecting statistical information as a query is described in [33]. The LOLA system [23] produces streams of data. Statistical model checking [46, 47] is an approach where executions of the systems are monitored until an algorithm from statistics can produce an estimate for the system to satisfy a given property.

3 The Choice of Base Language

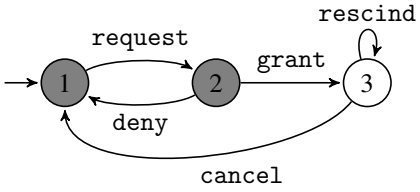
The first, and most important, choice when designing an RV logic is that of the base language. Here we consider the most classical choices of state machines and regular expressions [58], as well as temporal logic [49].

3.1 State Machines

One of the most fundamental formalisms for specifying orderings of events is state machines. We begin by introducing a property well-suited to state machines, concerned with the allocation of resources to tasks.

Property 3 (Resource Lifecycle). For every task t and resource r there is a life-cycle of allowed actions. Initially the task does not own the resource and from this state it can request the resource. This request can be denied or granted. If denied it returns to the unowned state, if granted it moves to an owned state. In an owned state the task can be asked to rescind the resource (hand it back), in which case it stays in this state, or the task can cancel its ownership, in which case it returns to the unowned state. A granted resource must eventually be canceled. No other action orderings are allowed.

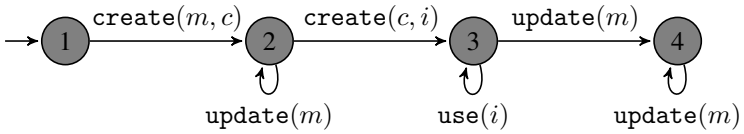
If we ignore the data part (task and resource identities), this property can be specified as a state machine as follows, where only states 1 and 2 are acceptance states, meaning that a granted resource must be eventually canceled. We give both a graphical and textual representation of the state machine.



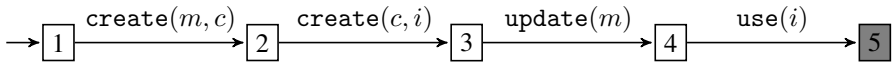

```
accept state 1 {
  request -> 2
}
accept state 2 {
  deny  -> 1
  grant -> 3
}
state 3 {
  rescind -> 3
  cancel  -> 1
}
```

The Semantics of Missing Transitions In the above state machine the transition relation is not complete (closed) i.e. we do not have a next target state for each combination of source state and event. For example, there is no transition with the label `deny` leaving state 1. The implicit understanding is that the transition relation is closed to an implicit *failure* state: all missing transitions lead to the *failure* state. We will call this approach a *next semantics* as it requires each next event to cause a transition. The alternative is a *skip semantics* where observed events may be skipped if there is no transition for them. Note that the standard interpretation of finite state machines (in theoretical computer science) is a next semantics. In contrast, UML statecharts [52, 27] are often given a skip semantics. See [3, 5] for RV systems allowing a mix of next and skip semantics.

To illustrate the difference, let us return to Property 1 (UnsafeMapIterator) where we gave a positive (formula (1) page 4) and negative (formula (2) page 4) formulation of the property as regular expressions. We can turn these regular expressions into state machines[6] as follows. Graphically we represent states with a next and skip semantics as circles and squares respectively[7]. The positive state machine formulation is:



The negative state machine formulation of this property is:



Traditionally regular expressions are translated to finite state machines with next-states. We observe that the positive formulation uses next-states whilst the negative uses skip-states. It is quite common to use a next semantics with positive formulations and a skip

[6] As before, the focus is not on the data part. Here we use the same operators as before, which are like universal and existential quantification for the positive and negative formulations respectively. The way we add parameters to state machines is covered extensively in Section 4.

[7] We note that this graphical presentation has been reversed compared to some previous work [3, 55]. We have chosen this presentation here as a next semantics is more typical for state machines as is a circle being used to represent a state, and states in state charts, which usually have skip semantics, normally are drawn as boxes, although typically with rounded corners.

semantics with negative formulations. We would, however, want to allow a mixture of such states within one specification, allowing a fine-grained control over the closure. As a simple example demonstrating this desire, consider a property over an alphabet of events $\{e_1, \ldots, e_n, \mathsf{quiet}, \mathsf{loud}\}$ stating that no other events should occur between quiet and loud. Below we demonstrate three different state machines capturing this property. The first uses implicit next states, the second uses implicit skip states, and the third uses a mixture. Using a mixture of states allows us to specify the property more concisely. Whilst this is a simple example, the general idea extends to more complex properties.

```
accept state 1 {
  e1 -> 1
  ...
  en -> 1
  quiet -> 2
}
accept state 2 {
  loud -> 2
}
```

```
accept state 1 {
  quiet -> 2
}
accept state 2 {
  e1 -> error
  ...
  en -> error
  loud -> 2
}
```

```
accept skip state 1 {
  quiet -> 2
}
accept next state 2 {
  loud -> 2
}
```

Alphabets In the case where next-states are used, as in the first positive formulation above, where each observed incoming event must match a transition, it is crucial that only *events of concern* are matched against the transitions. Otherwise any trace with additional events might easily fail to conform. To avoid this problem, such a specification must be associated with an alphabet: the events of concern. A trace that contains events not in the alphabet must first be projected to remove such. Often the alphabet is the set of events mentioned in the specification, but that is not always the case.

Fine-Grained Acceptance An advantage of state machines is that they allow for a fine-grained notion of acceptance. This is demonstrated in the above state machine for Property 3 (Resource Lifecycle) where state 3 is non-final whilst states 1 and 2 are final. This is key for any language wanting to capture properties which are not purely safety properties. An extension of this fine-grained acceptance is the ability to attach different kinds of failures to different states. This has particular use in runtime monitoring where different correction actions may be required for different forms of failure, as i.e. supported in [51]. Indeed, it would allow a specification to separate *soft* failures that only require reporting and *hard* failures that require immediate termination or intervention.

Anonymous States One reason that temporal logics and regular expressions often yield more succinct specifications than state machines is that all intermediate states need to be explicitly named in the state machine. A simple syntactic layer of syntax on top of state machines can, however, allow *anonymous states* [5], making state machines more succinct. Below left we merge states 1 and 2 of the Resource Lifecycle property by turning state 2 into an anonymous acceptance state (state 3, which is not shown, is the same as before). Ignoring the rescind event, below right is shown how an event with a single outgoing transition can be treated even more concisely (here states are accepting by default and there is a next-semantics).

```
accept state 1 {
  request -> accept {
    deny  -> 1
    grant -> 3
  }
}
```

```
state 1 {
  request -> {
    deny  -> 1
    grant -> cancel -> 1
  }
}
```

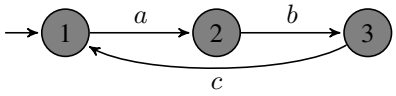
Generally, one has to capture these assumptions (acceptance state or not, next-state or skip-state) with an additional annotations in the non-default cases.

3.2 Regular Expressions

Anonymous states are carried to the extreme in regular expressions i.e. there are no named states. State machines and regular expressions have the same expressive power in the propositional case. Regular expressions are more succinct than their corresponding state machines since intermediate states are not mentioned by name, only transitions are mentioned.

Standard Operators We have already seen several examples of regular expressions and how they related to state machines, including how next-states and skip-states can be used to model their semantics. The basic form of a regular expression is a letter, such as for example: a, representing the language $\{a\}$. The operators apply semantically to languages and produce new languages. Given two regular expressions E_1 and E_2, the basic operators are union: $E_1|E_2$ (the union of the languages denoted by E_1 and E_2, sometimes written, although not here, as $E_1 + E_2$); concatenation: E_1E_2 (the set of words, each of the form l_1l_2 where l_i is a word in the language denoted by E_i); and finally closure: E^* (set of words each obtained by concatenating any number of words denoted by E). Additional operators are usually defined for convenience, but provide no additional expressive power. These include the dot: '.' (representing any letter, the union of all letters in the alphabet); plus: E^+ (meaning one or more, equivalent to EE^*); optional: $E?$ (meaning $E|\epsilon$ where ϵ accepts the empty string); and repetition: E^n (for some number n, meaning n copies of E, and variants of this operator indicating minimum and maximum number of occurrences). Negation is also commonly seen but most typically on letters. A common approach is to write unions of many letters: $a_1|a_2|\ldots|a_n$, as a list: $[a_1, a_2, \ldots, a_n]$, and negation of all these letters is then written as $[^\wedge a_1, a_2, \ldots, a_n]$. Negation of entire regular expressions, as in $\neg E$, is also semantically possible, but usually avoided due to complexity in generating the corresponding state machine.

Safety Properties The standard interpretation of the regular expression concatenation operator (that E_1E_2 denotes the set of words l_1l_2 where l_i is in the language denoted by E_i) makes it inconvenient to express certain safety properties. Consider for example the language denoted by the following state machine:



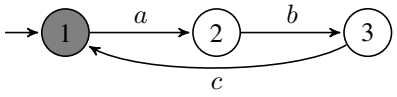
The main observation here is that all states are acceptance states, hence the language includes strings such as a, ab, abc, $abca$, etc. Representing this language as a regular expression with the standard semantics, however, becomes slightly inconvenient and error prone to write:

$$(a\,b\,c)^*(\epsilon \mid a \mid a\,b)$$

It would instead be desirable just to write:

$$(a\,b\,c)^*$$

However, this formula denotes the following automaton with the standard interpretation of regular expressions:



If we want the former interpretation, but the latter formulation of the regular expression, we need to provide a closure operation that closes a language to include all its prefixes. That is, given a regular expression E with the standard interpretation one can form the closure $closure(E)$ of this to include the language denoted by E as well as all its prefixes. Our property would then become $closure((a\,b\,c)^*)$.

Limitations of Regular Expressions While regular expressions generally are very succinct and useful, in some cases the regular expression formulation of e.g. a state machine can become so convoluted that an ordinary user will be challenged in creating it, as well as in reading it. To illustrate this let us revisit Property 3 (ResourceLifecycle). Recall that the state machine was pretty straightforward to create. A regular expression version of this property is the following:

$$((request\ deny)^*\ request\ grant\ rescind^*\ cancel)^*\ request?$$

This regular expression is not completely obvious to create, in part due to the fact that some of the states in the state machine are acceptance states and some are not. In fact, we got this regular expression wrong in the first attempt. It is easy to see that if the state machine gets much more complicated, the regular expression becomes overly complex to write and even read. Furthermore, updatable data variables, which are straightforward to support in state machines, are not straightforward to introduce in regular expressions, this will be discussed in the subsequent section. Consequently, one may want to pursue a formalism that supports state machines (or a similar concept such as rule systems or variants of the μ-calculus [43], which in common have that states can be named and used in loops) in addition to a logic such as regular expressions or/and temporal logic.

3.3 Temporal Logic

In Section 3.1 it was discussed how states in a state machine can be anonymous, mixing anonymous states and named states in one notation. Regular expressions go to

the extreme and eliminate the notion of named state all together. Likewise, temporal logic eliminates the notion of named states. It was generally clear from the competition benchmarks that temporal logic provided the most elegant formulation of many properties. The difference was rather remarkable in several cases.

Standard Operators Introducing temporal logic in its many variations is beyond the scope of this paper. However, we will discuss the standard operators of future time Linear Temporal Logic (LTL) [53], the most common temporal logic used in runtime verification[8], and their past time counterparts. Future time LTL can be described as propositional logic plus the temporal operators $\bigcirc$ (*next*) and $\mathcal{U}$ (*until*). Their semantics are that $\bigcirc\varphi$ holds if φ holds at the next time point, and $\varphi_1 \mathcal{U} \varphi_2$ holds if φ_2 holds at some future time point and φ_1 holds at all time points from the current until and including the one before that future time point. The operators $\Diamond$ (*eventually*), $\Box$ (*always*), and $\mathcal{W}$ (*weak until*) can then be defined as follows: $\Diamond\varphi = true\ \mathcal{U}\ \varphi$, $\Box\varphi = \neg\Diamond\neg\varphi$, and $\varphi_1 \mathcal{W} \varphi_2 = \Box\varphi_1 \vee (\varphi_1 \mathcal{U} \varphi_2)$. Similarly, past time operators include $\bullet$ (*previous*, the dual of $\bigcirc$) and $\mathcal{S}$ (*since*, the dual of $\mathcal{U}$). Their semantics are that $\bullet\varphi$ holds if φ holds at the previous time point, and $\varphi_1 \mathcal{S} \varphi_2$ holds if φ_2 holds at some past time point and φ_1 holds at all time points since then to the current. The operators $\blacklozenge$ (*sometime in the past*) and $\blacksquare$ (*always in the past*) can then be defined as follows: $\blacklozenge\varphi = true\ \mathcal{S}\ \varphi$, $\blacksquare\varphi = \neg\blacklozenge\neg\varphi$. A convenient logic is likely one that includes past time as well as future time operators.

The symbols just introduced look mathematically elegant, but they are not in ASCII format. Therefore it is typical to replace these logical symbols by text. For example $\Box$ may be written as the word `always`, or as the symbol `[]`. We will use a mix of the logical and textual word presentations here.

Illustrating Strength of Temporal Logic With the following property we shall illustrate the advantage of a temporal logic over a state machine.

Property 4 (ResourceConflictManagement). This property represents the management of conflicts between resources as managed by a planetary rover's internal resource management system - or any resource management system in general. It is assumed that conflicts between resources are declared at the beginning of operation. After this point resources that are in conflict with each other cannot be granted at the same time. A conflict between resources r_1 and r_2 is captured by the event `conflict`(r_1,r_2) and a conflict is symmetrical. Resources are granted and canceled using `grant`(r) and `cancel`(r) respectively.

The specification of this property as a state machine in textual format becomes somewhat verbose (note that here we write properties in ASCII format for better illustrating how they would be written down in practice):

[8] CTL (Computation Tree Logic) [21] is a logic on execution path trees, and has therefore not been popular in runtime verification. However, one can imagine a CTL-like logic being used for analyzing a set of traces, merged into a tree.

```
For all r1,r2
accept skip state start {
  conflict(r1,r2) -> free
  conflict(r2,r1) -> free
}
accept skip state free {
  grant(r1) -> granted
}
accept skip state granted {
  cancel(r1) -> free
  grant(r2)  -> failure
}
```

Alternatively, this property can be stated as a more concise temporal logic formula, for example as the following future time temporal logic formula:

```
forall r1,r2
  always ((conflict(r1,r2) or conflict(r2,r1)) =>
    (always (grant(r1) =>
      ((not grant(r2)) weakuntil cancel(r1)))))
```

That is, it is always the case that if a conflict is declared between two resources r_1 and r_2, then it is always the case that if r_1 is granted then r_2 is not thereafter granted unless r_1 is canceled first. In both formulations, to capture the symmetric conflict event, we need to match against either $\texttt{conflict}(r_1, r_2)$ or $\texttt{conflict}(r_2, r_1)$.

We can express this as a negative (a match is an error) regular expression as follows:

```
forall r1,r2
  (conflict(r1,r2)|conflict(r2,r1)) .* grant(r1) (!cancel(r1))* grant(r2)
```

When working in temporal logic one usually formulates properties positively: what is desired to hold, whereas when formulating the same properties as regular expressions they appear easier to write in negative form. When formalizing requirements, however, it may appear somewhat inconvenient to have to negate the properties. Another example is the property $\Box(a \Rightarrow \Diamond b)$, which as a regular expression may be stated as a suffix matching negative expression $a.(\neg b)^*$. A positive regular expression formulation gets rather convoluted: $((\neg a)^*(a.^*b)?)^*$. Hence if a positive formulation of requirements is desired, as e.g. in project requirement documents, temporal logic may in some scenarios be more attractive than regular expressions.

The Convenience of Past Time Operators The same property can also be stated as a past time formula, as follows.

$$(\forall r_1, r_2)\Box\left(\left(\texttt{grant}(r_1) \wedge \blacklozenge\begin{pmatrix}\texttt{conflict}(r_1, r_2)\\ \vee\\ \texttt{conflict}(r_2, r_1)\end{pmatrix}\right) \Rightarrow \neg\bullet\begin{pmatrix}\neg\texttt{cancel}(r_2)\\ \mathcal{S}\\ \texttt{grant}(r_2)\end{pmatrix}\right)$$

However, this past time logic formula is not convincingly easier to read than the future time version. Especially as there are multiple references to different points in the past. There are, however, cases where past time is more convenient, as also pointed out in

[45]. Consider the hasNextIterator property 2 again. The property states that every call of next on an iterator should be preceded by a call of hasNext (which returns true). If we should state this property as a future time property, it would become:

$$(\forall i) \begin{pmatrix} (\neg \texttt{next}(i) \; \mathcal{W} \; \texttt{hasNext}(i, \mathit{true})) \wedge \\ \Box(\texttt{next}(i) \Rightarrow \bigcirc(\neg \texttt{next}(i) \; \mathcal{W} \; \texttt{hasNext}(i, \mathit{true}))) \end{pmatrix}$$

This property seems overly complicated. This is caused by the necessity to separate two scenarios: (i) the first occurring `next` in the trace, and (ii) subsequent `next` events, appearing after previous `next` events. The property becomes slightly more concise, and thus more readable, when formulated in past time logic:

$$(\forall i) \, \Box(\texttt{next}(i) \rightarrow \bullet(\neg \texttt{next}(i) \; \mathcal{S} \; \texttt{hasNext}(i, \mathit{true})))$$

Adding Convenient Operators Temporal logic is often attributed being difficult to use, and it is occasionally claimed that even state machines are easier to use by practitioners. The specification of the competition exercises, however, shows to us that temporal logic makes specification substantially easier in quite many cases. A logic like LTL, however, appears to have some flaws from a usability point of view, including: binary operators that are tricky to remember the semantics of (such as *weak until* versus *until*, *since*, etc.), formulas tend to get nested, requiring use of parentheses for grouping sub-expressions for even the simplest formulas, and cumbersome handing of sequencing. We briefly recall how some of these problems can be alleviated with convenient alternative syntax.

Consider the previous past time formulation of the HasNextIterator property, that contains the subterm: $\neg \texttt{next}(i) \; \mathcal{S} \; \texttt{hasNext}(i, \mathit{true})$, meaning: $\texttt{hasNext}(i, \mathit{true})$ has occurred in the past and since then no $\texttt{next}(i)$ has occurred. This is not a very readable formulation of this property. An example of a more convenient operator is the temporal operator $[P, Q)$ from MaC [42], meaning P has been true in the past and since then Q has not. Using this operator the sub-term becomes: $[\texttt{hasNext}(i, \mathit{true}), \texttt{next}(i))$, which visually better illustrates the temporal order of events. The property now becomes:

$$(\forall i) \, \Box(\texttt{next}(i) \rightarrow \bullet[\texttt{hasNext}(i, \mathit{true}), \texttt{next}(i)))$$

Consider further that in such implications usually the right-hand temporal expression is meant to be true in the previous state (hence the use of the $\bullet$-operator). One could fold the $\rightarrow$-operator and $\bullet$-operator into one operator $\overset{\bullet}{\rightarrow}$, assume all variables quantified, and a $\Box$ in front of all properties, and write the property as follows:

$$\texttt{next}(i) \overset{\bullet}{\rightarrow} [\texttt{hasNext}(i, \mathit{true}), \texttt{next}(i))$$

Similarly one can imagine a $P \overset{\bigcirc}{\rightarrow} Q = P \rightarrow \bigcirc Q$ operator for future time logic. Another classical convenient operator is $\mathit{never}\, P$ being equivalent to $\Box \neg P$.

Limitations of Temporal Logic As shown in [61], LTL cannot express all regular properties (it is only star-free regular), for example it cannot express the property: *"p holds at every other moment"*, which can easily be expressed as a state machine or a regular expression as follows: $(.\,p)^*$. LTL is furthermore also at times inconvenient as a notation. We shall consider two examples here, a state machine, and a temporal formula conditioned on a sequence of events. First the *state machine*. The temporal logic formulation of Property 3 (ResourceLifecycle), ignoring the data element, can be given as follows:

$$\texttt{stop} \vee (\texttt{request} \wedge \Box \begin{pmatrix} \texttt{request} \rightarrow \bigcirc(\texttt{deny} \vee \texttt{grant} \vee \texttt{stop}) \\ \texttt{deny} \rightarrow \bigcirc(\texttt{request} \vee \texttt{stop}) \\ \texttt{grant} \rightarrow \bigcirc(\texttt{rescind}\, \mathcal{U}\, \texttt{cancel}) \\ \texttt{cancel} \rightarrow \bigcirc(\texttt{request} \vee \texttt{stop}) \end{pmatrix}$$

where $\texttt{stop} = \Box\neg(\texttt{request} \vee \texttt{deny} \vee \texttt{grant} \vee \texttt{rescind} \vee \texttt{cancel})$, and is used to indicate that no further events are required. These rules exactly mirror the state transitions of the state machine. In this case, temporal logic, specifically LTL, is arguably less elegant than state machines. Note that without introducing the name `stop` the formula would become even more complicated.

As our second example, let us consider *a temporal formula conditioned on a sequence of events*. To do this we will use Property 1 (UnsafeMapIterator). Suppose we wanted to express this property in temporal logic. A possible formulation would be the following rather unreadable formula:

$$\Lambda m, c, i : \Box\neg \left((\texttt{create}(m, c) \wedge \begin{pmatrix} \neg\texttt{create}(c, i)\, \mathcal{U}\, (\texttt{create}(c, i) \wedge \\ (\neg\texttt{update}(m)\, \mathcal{U}\, (\texttt{update}(m) \wedge \Diamond\texttt{use}(i)))) \end{pmatrix} \right)$$

A more readable temporal logic formula is the following, which, however, does not say quite the same thing (since the second and third $\Box$-operator occurrences each quantify over all future events), although it seems in this case to be usable.

$$\Lambda m, c, i : \Box(\texttt{create}(m, c) \Rightarrow \Box(\texttt{create}(c, i) \Rightarrow \Box(\texttt{update}(m) \Rightarrow \Box\neg\texttt{use}(i))))$$

To obtain a more readable formula, we could instead combine regular expressions and temporal logic and write it as follows, using a regular expression on the left-hand side of the implication and an LTL formula on the right-hand side:

$$\Lambda m, c, i : \texttt{create}(m, c).\texttt{create}(c, i).\texttt{update}(m) \Rightarrow \Box\neg\texttt{use}(i)$$

The temporal logic PSL [28] adds an operator to LTL named *suffix implication*, and denoted $r \mapsto \psi$, for a regular expression r and a temporal logic formula ψ, which holds on a word w if for every prefix of w recognized by r, the suffix of w starting at the letter on which that prefix ends, satisfies ψ. This addition to LTL results in a logic with an expressive power corresponding to ω-regular languages (PSL is a logic intended for model checking, where infinite words are considered). Similar ideas are also seen in dynamic logic, see for example [34]. PSL generally contains several operators, which

make modeling easier. These include beyond the suffix implication also: repetition $r * n$ (repeat a regular expression n times); intersection $r_1 \cap r_2$; a past time operator $ended(r)$ that turns a regular expression r to hold on the past trace; strong $r!$ and weak regular expressions r, where strong is the normal interpretation of a regular expression, and a weak regular expression denotes the language of the strong regular expression augmented with all prefixes (what on page 10 was referred to as the closure of a regular expression and denoted by $closure(r)$).

4 Handling Data

The previous section ignored the details of how each of the languages could be extended to deal with data. Here we review the main approaches. We shall first outline what we mean by data. Subsequently, the handling of data is discussed in the contexts of state machines, regular expressions and temporal logics. However, the discussion of data for one base language usually carries over to other base languages.

4.1 Where do Data Occur?

Data can come from three sources.

Variables in the SBM. The monitor may be able to directly observe the internal state of the executing program. For example, if the monitor code is embedded (as code snippets) into the SBM. Program assertions, as supported by most programming languages, form an example of this. Alternatively, a transition of a state machine may be guarded by $x > 4$ where x is a program variable. This introduces a tight coupling between the specification and system being monitored. This form of data is not discussed here.

Event Parameters. Events transmitted from the SBM to the monitor can carry data as parameters. That is, an event consist of a name and a list of data values. An example is the event $login(u, t)$ representing the logging in by user u at time t. Within the runtime verification community such events are often called *parametric events*, as the data values are seen as parameters, and traces of these events are called *parametric traces*. Parametric events are the main source of data in this presentation.

Variables in the Monitor. The monitor itself can declare, update and read variables local to the monitor. This is seen in solutions where monitors are given as state machines or written in a programming language. This approach will also be discussed below.

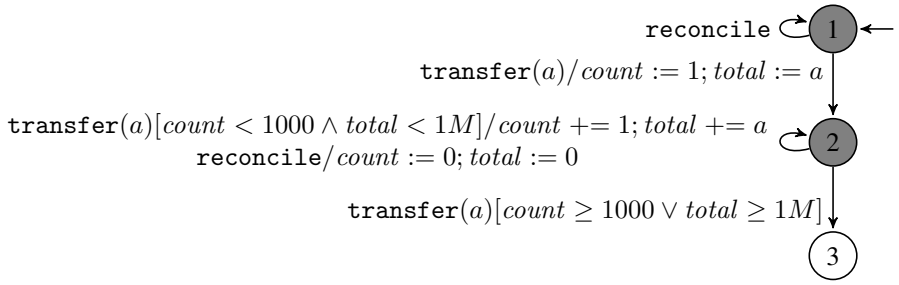
4.2 Extended Finite State Machines

Conventional finite state machines have a finite number of control states and transitions are labelled with atomic letters over a finite alphabet. Extended finite state machines (EFSM) [19, 40] extend finite state machines by allowing the declaration of a set of mutable variables, which can be read in transition guards, and updated in transition actions, where an action is a sequence of assignment statements assigning values to

the variables. The standard transition relation is lifted to configurations, i.e. pairs of (control) states and variable valuations. Turing machines and pushdown automata (with the expressive power of context free languages) [58] are examples of EFSMs, so this is a powerful model. However, as we shall see, EFSMs are not convenient for our purposes in their original form. We use the following property to illustrate EFSMS.

Property 5 (Reconciling Account). The administrator must reconcile an account every 1000 attempted external money transfers or an aggregate total of one million dollars. The `reconcile` event is recorded when accounts are reconciled and the event `transfer`(a) records a transfer of a dollars.

Note that the `transfer`(a) event in this property carries data (the amount a transferred). EFSMs traditionally do not operate on such parameterized events but on atomic events. We shall, however, make this extension here in our first example, moving from evaluating input over a finite alphabet to input over an infinite alphabet. The EFSM for this property is shown in the following.



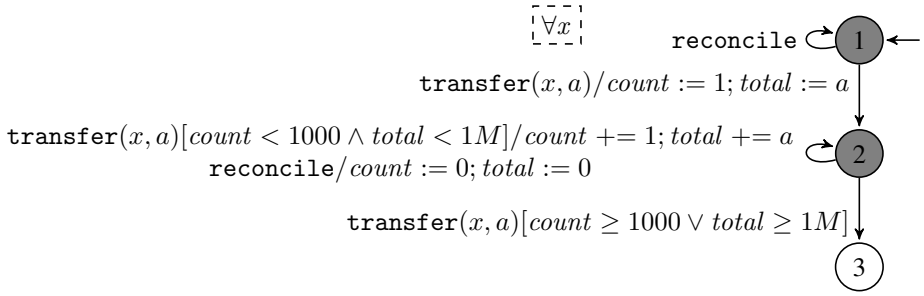
Two variables are introduced. The variables *count* and $total$ hold the number of transfers respectively the sum of amounts transferred since the last reconciliation. Transitions are now written as `event`$[guard]/assignment$. Event `transfer`(a) on a transition is used to *match* a concrete event in the trace and bind the value in that event to the variable a. Each new transfer in the trace causes a to be bound to a new value, it is essentially just a variable just like *count* and $total$. Typically variables are used to hold data of primitive types (integers, reals, Booleans, etc), but they could be used to hold more complex data structures. It is worth observing that UML state charts effectively are a form of extended state machines, with added concepts such as hierarchical states.

4.3 Typestates

Extended state machines with parameterized events as described above, where parameters are mutable variables, however, are still not convenient for specification purposes. We need an event parameter concept where parameters once bound stay constant, and we have a copy of the state machine for each parameter value. In other words, we need to quantify over parameters. To demonstrate a case where this may be required let us consider a further extension of the account reconciliation example.

Property 6 (Reconciling Accounts). The administrator must reconcile every account x every 1000 attempted external money transfers or an aggregate total of one million dollars. The `reconcile` event is recorded when accounts are reconciled and the event `transfer`(x, a) records a transfer of a dollars for account x.

This can be specified in the same way as before but this time adding a quantification over account x.



There will be an instance spawned of this state machine for each account x being monitored. This notion of quantification corresponds to what is also referred to as a *typestate* property [60], a programming language concept for static typing, which extends the notion of type with a (safety) state machine over the methods of that type. This state machine is quantified over all objects of the given type. Consider the account as a type with the methods `transfer`(a) and `reconcile`$()$. A typestate is a refinement of such a type where a constraint is defined on the order in which these methods can be called. A typestate monitor can very simply be implemented by adding the EFSM monitor to each object of the type (state). This approach can also be used for Property 2 (HasNextIterator).

As noted, there are now two kinds of variables in the state machine, those that are quantified (x) and constant once bound, and those that are continuously mutable (a, $count$, $total$), referred to as *free* variables. In this formulation we want each instance of the state machine to have its own copies of the free variables. Therefore, we can view these variables as having *local scope* i.e. they are local to each particular instantiation of quantified variables. In extended finite state machines variables are often thought of us *global*, but it is clear that in this case we want locality. There is, however, a case to be made for variables with *global scope* where all instances of the extended finite state machine read from and write to such. This would be needed i.e. if reconciliation frequency depended on the total number of transfers for all accounts.

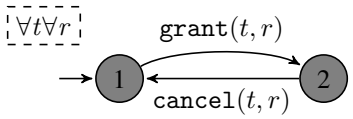
4.4 Parametric Trace Slicing

Typestates, implemented as extended state machines, although pleasantly simple, are not sufficiently convenient for commonly occurring monitoring scenarios due to the restriction of only quantifying over one variable. Consider for example Property 3 (Resource Lifecycle). We here need to deal with tasks as well as resource objects. We

want to specify the appropriate behavior *for each pair* of tasks and resources. Similarly for Property 1 (UnsafeMapIterator). A generalization of the typestate approach is the concept of *parametric trace slicing*, first introduced in Tracematches [2] to work for regular expressions, and then generalized in JavaMOP [18, 51] for adding parametric trace slicing to any propositional temporal language, that can be defined as a "plugin". Quantified Event Automata (QEA) [3, 54] are a further generalization adding existential quantification and free variables (see below). Consider for example the following property (not taken from the competitions).

Property 7 (Simple ResourceManagement). A resource can only be granted once to a task until the task cancels the resource (granting and canceling a resource wrt. a particular task must alternate). A resource r is granted to a task t using the event $\texttt{grant}(t, r)$ and canceled using $\texttt{cancel}(t, r)$.

This property can be formalized as follows using a trace slicing approach.



The parametric trace slicing approach considers *collections* of instantiations of quantified variables. In the original formulation of this idea, each collection is associated with a propositional monitor for that combination of parameter values. In this case the propositional monitor is a standard state machine. The trace is then *projected* into slices: one for each instantiation (combination of parameter values), so that only events relevant to the instantiation are included in the monitoring wrt. these particular values. This view of quantification requires domains for the quantified variables (the sets of values they quantify over). A typical choice is to let these domains consist of the values occurring in the trace. Given the above state machine, the following (satisfying) trace results in the domain for the variable t to be $\{A\}$ and the domain for the variable r to be $\{R_1, R_2\}$:

$$\texttt{grant}(A, R_1).\texttt{cancel}(A, R_1).\texttt{grant}(A, R_2).\texttt{cancel}(A, R_2)$$

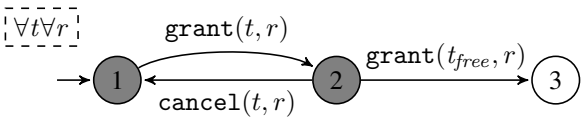
4.5 Parametric Trace Slicing with Free Variables

In parametric trace slicing each instantiation of quantified variables encountered in the trace is associated with a propositional monitor, and only events concerned with that instantiation are mapped to the monitor. Things, however, become more complex when free variables are introduced, which is necessary to increase expressiveness (see [3] for how parametric trace slicing is extended with free variables). To illustrate this, we introduce a more complex version of Property 7 (SimpleResourceManagement).

Property 8 (ResourceManagement). In addition to Property 7 (i.e. granting and canceling a resource wrt. a particular task must alternate), a resource can only be held by at most one task at a time. Recall that a resource r is granted to a task t using the event $\texttt{grant}(t, r)$ and canceled using $\texttt{cancel}(t, r)$.

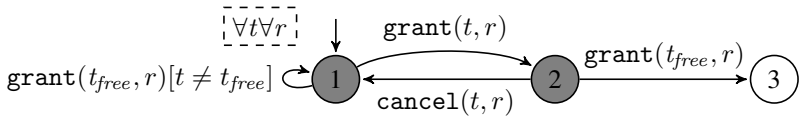
The previous specification for Property 7 does not capture this property. Consider the trace $\texttt{grant}(A, R).\texttt{grant}(B, R)$, violating the property. This generates two variable valuations: $[t = A, r = R]$ and $[t = B, r = R]$. The event $\texttt{grant}(B, R)$ is not relevant for the instance $[t = A, r = R]$, it is only relevant for the instance $[t = B, r = R]$, and vice versa. The trace is unfortunately consequently sliced into two independent subtraces: $\texttt{grant}(A, R)$ and $\texttt{grant}(B, R)$, with no connection between them. Therefore this formulation will not detect a violation in this trace.

To detect a violation we need to detect the existence of a violating task, i.e. the one that tries to take the resource r whilst it is held by task t. We could attempt to do this using a free variable t_{free} to capture the violating task as follows.



It turns out that we need to re-define the notion of projection to handle such free variables. Assume again the quantification instance $[t = A, r = R]$. Clearly, events that only mention A and R are relevant as before. But also grant events mentioning R and some other task are also relevant as they could match $\texttt{grant}(t_{free}, r)$. This would mean that $\texttt{grant}(B, R)$ would be relevant to the instance $[t = A, r = R]$. But now we can see a further issue. Consider the safe trace $\texttt{grant}(B, R).\texttt{cancel}(B, R).\texttt{grant}(A, R)$. The projection to $[t = A, r = R]$ will be $\texttt{grant}(B, R).\texttt{grant}(A, R)$, since $\texttt{grant}(B, R)$ is considered relevant for the reason mentioned above, and since $\texttt{cancel}(B, R)$ is not since $A \neq B$. This trace will therefore be rejected as the monitor cannot make a transition on the first event $\texttt{grant}(B, R)$.

This highlights the subtleties that occur when dealing with free variables and projection. Once an event with a free variable has been added to the alphabet the user should consider how this affects the events that could be relevant at other states. This leads to a final correct, but less attractive, formulation:



This extends the state machine with a looping transition to skip grants of the resource to other tasks when the current task t does not hold the resource. An alternative solution would have been to existentially quantify t_{free}. However, this has implications related to the efficiency of the associated monitoring algorithm, which we do not discuss here [54].

4.6 Quantification in Temporal Logics

First-Order Quantification The standard way of dealing with data in logic via *quantification* is relevant to temporal logic, and is based on the notion of evaluating a formula

with respect to a first-order temporal structure. The standard approach is to add a formula expression such as $\forall x.\varphi$ to the syntax of the logic and include a case similar to the following in the trace semantics [10]

$$\mathcal{T}, i, \sigma \models \forall x.\varphi \;\; \textit{iff for every} \;\; d \in \mathcal{D}(x) \;\; \textit{we have} \;\; \mathcal{T}, i, \sigma[x \mapsto d] \models \varphi$$

where the temporal structure $\mathcal{T}$ captures the trace, i.e. it is a finite sequence of structures where each structure describes which events are present in that time point. This usually follows the standard logical approach of modeling events as *predicates* and defining an *interpretation* evaluating events occurring at that time point to true and all other events to false. The temporal structure also defines the domain function $\mathcal{D}$ but the way in which it does this differs between approaches, as described below.

The notion of first-order LTL introduced by Emerson [29] follows this approach, although assumes predicates have global interpretation, which is not suitable here but the extension is straightforward. However, for pragmatic reasons, languages for runtime verification have been designed to consider alternative first-order extensions of LTL. The rest of this section discusses these alternative design decisions.

Different Notions of Quantification There are two main approaches to defining the domain function $\mathcal{D}$ above. Simply, either $\mathcal{D}$ is local to the current time point (as in [13, 36]), or it is constant throughout the temporal structure (as in [10] and the original work of Emerson). If the domain function is local then it is typically derived from the events occurring in that time point. If the domain function is constant then it typically consists of (a superset of) values appearing in the trace. The idea of the first approach is to restrict quantification so that it is only used to create future obligations about events occurring at the current point in time. For this to work it is necessary to syntactically restrict the occurrence of quantification so that the values it is quantifying over occur at the current time point e.g. the first of the following two (normally) equivalent formulas would break this rule.

$$(\forall f)\Box(\texttt{open}(f) \rightarrow \Diamond\texttt{close}(f)) \qquad \Box(\forall f)(\texttt{open}(f) \rightarrow \Diamond\texttt{close}(f))$$

The second formula has the same interpretation in the two models of quantification (where we assume that the domain in the current time point is a subset of the constant domain) as the right-side of the implication will only be true for values in the current time point. This demonstrates that, under certain syntactic restrictions, the two models of quantification coincide. This can be advantageous as monitoring algorithms dealing with the first model of quantification will generally be more straightforward as decisions about quantification can be local. To see that the two models are different, note that the formula

$$\Box(\exists x)(\neg p(x))$$

is unsatisfiable in general for the first model of quantification as the domain of x is given exactly by the values such that $p(x)$ is true at the current time point. But in the second model of quantification this becomes a reasonable statement.

Note that the first model is dependent on *where* quantification happens and as a result cannot express some formulas. For example, this formula cannot be expressed

in this model as the values being quantified over cannot be present in the current time point:

$$(\forall f)\Box(\texttt{open}(f) \rightarrow \exists u : (\Diamond\texttt{read}(f,u) \vee \Diamond\texttt{write}(f,u)))$$

In either setting it is possible for domains to be *infinite* (in theory) as long as formulas are *domain independent*, which is a semantic notion ensuring that only a finite subset of the domain is required for trace-checking (see [20, 11]). As an example, $(\forall x)\Diamond p(x)$ is not domain independent, but $(\forall x)\Box p(x) \rightarrow \Diamond q(x)$ is domain independent as checking this formula only requires checking the finite subset of values v such that $p(v)$ appears in the trace. Note that determining whether a formula is domain independent is undecidable and practically this is checked via conservative syntactic restrictions.

The Difference with Parametric Trace Slicing It is at this point appropriate to point out, that the standard logic interpretation of quantification presented above is different from the parametric slicing approach, resulting in subtly different interpretations of formulas. Consider Property 3 (Resource Lifecycle). Previously (page 14) we gave a propositional temporal logic formulation (no data) that included the sub-formula:

$$\Box(\texttt{request} \rightarrow \bigcirc(\texttt{deny} \vee \texttt{grant} \vee \texttt{stop})$$

When we add first-order quantification this becomes:

$$(\forall r)\Box(\texttt{request}(r) \rightarrow \bigcirc(\texttt{deny}(r) \vee \texttt{grant}(r) \vee \texttt{stop}(r))$$

This formula, however, no longer means what we want it to. Consider e.g. the trace:

$$\texttt{request}(A).\texttt{request}(B).\texttt{deny}(A).\texttt{deny}(B)$$

This is expected to be a correct trace, but it does not satisfy the quantified formula since in the first state $\texttt{request}(A)$ is true but none of $\texttt{deny}(A)$, $\texttt{grant}(A)$, nor $\texttt{stop}(A)$ are true in the next (second) state. However, with parametric trace slicing, the formula (note the use of Λ instead of $\forall$).

$$\Lambda r.\ \Box(\texttt{request}(r) \rightarrow \bigcirc(\texttt{deny}(r) \vee \texttt{grant}(r) \vee \texttt{stop}(r))$$

has the intended interpretation (accepting the trace), as parametric trace slicing *projects* the trace to a slice only including events relevant for r. As outlined in [57], the main difference is the treatment of the notion of *next*.

Everything is Quantification We have previously seen the need for introducing free variables in state machines in combination with parametric trace slicing. Free variables are not needed in temporal logic, where quantification is sufficient. Consider for example Property 8 (ResourceManagement) and the corresponding state machine on page 19, which uses a free task variable t_{free} in addition to the quantified t. The property can alternatively be formulated in temporal logic only using quantification as follows:

$$(\forall t, r)\Box(\texttt{grant}(t,r) \Rightarrow \bigcirc((\neg(\exists t')\ \texttt{grant}(t',r))\ \mathcal{W}\ \texttt{cancel}(t,r)))$$

The approach is to allow quantification inside the formula, and not only at the outermost level. In first-order temporal logic it is generally possible to write the quantifiers at arbitrary points of the specification. It is important to understand that quantification is to be evaluated at the point in the trace that it is met. For example,

$$(\forall x)\Diamond \mathtt{f}(x) \not\equiv \Diamond(\forall x)\mathtt{f}(x)$$

as the first property says that for every x the event $\mathtt{f}(x)$ eventually happens, but for different values for x this could happen at different points, whereas the second property requires that these all happen at the same point. However, in some cases quantification inside a temporal operator can be lifted outside, the standard identities are the following:

$$(\forall x)\Box\varphi \equiv \Box(\forall x)\varphi \qquad (\exists x)\Diamond\varphi \equiv \Diamond(\exists x)\varphi$$

The main reason why quantification is enough in temporal logic is due to the fact that temporal logic has a notion of sub-formula, and a quantification is over some sub-formula. In state machines we usually do not operate with a notion of sub-machine (except in state charts), and it therefore becomes difficult to define the scope of a quantifier. One can imagine embedded quantifiers in regular expressions, however, just as one can imagine free variables in regular expressions.

Section 4.3 discussed the concept of free variables with global scope. A similar notion in temporal logic could correspond to the usage of so-called *counting quantifiers*. Suppose for example that we wanted to state that each task t can at most hold N resources at a time. This can be captured as the property:

$$\Box(\forall t)(\exists^{\leq N} r)(\neg \mathtt{cancel}(t,r)\ \mathcal{S}\ \mathtt{grant}(t,r))$$

This can be read as: *it is always the case that for all tasks, there exist at most N resources r, that have not been canceled by t since they were granted to t.* Counting quantifiers typically preserve the expressiveness of the language, assuming that the language includes predicates.

Some properties of interest to runtime verification go beyond the expressiveness of first-order temporal logic. A simple extension is the usage of so-called *percentage counting quantifiers* [50] of the form $\mathbf{A}_{\geq P}x : p(x) \Rightarrow \phi$ which capture the property that for at least $P\%$ of the values d in the domain of x such that $p(d)$ holds, the statement ϕ holds. This allows for the expression of properties such as

$$\mathbf{A}_{\geq 0.95}s : \mathtt{socket}(s) \Rightarrow (\Box\mathtt{receive}(s) \Rightarrow \Diamond\mathtt{respond}(s))$$

stating that at least 95% of open sockets are eventually closed.

Another property of interest corresponding to *second-order* quantification is that of *deadlock avoidance*. As described in [16], if a graph is constructed where directed edges between locks indicate a lock ordering, then a cycle in this graph indicates a potential deadlock. Cycle detection is a reachability property, which is inherently second-order i.e. it relates to the second-order temporal property

$$\neg\exists\{l_1,\ldots l_n\}\left(\mathtt{lock}(l_{n+1},l_1) \wedge \bigwedge_{i=0}^{i=n-1} \mathtt{lock}(l_i,l_{i+1})\right)$$

i.e. there does not exist a set of locks containing a cycle.

The Past is not Simpler Concerning monitoring algorithms, future time logic with data lends itself to a very simple syntax-oriented tableaux-like procedure, as in [4, 12, 5, 36]. Past time logics interestingly require a different more elaborate approach, i.e. dynamic programming, as described in [38, 10].

Events and the Signature Typically in first-order logic one has *predicates* and *functions*. As mentioned previously, it is normal for first-order extensions of LTL for runtime verification to model events as predicates interpreted as either true or false in the current time point. This easily supports the notion of multiple events occurring in a single time point. Indeed, if one were to restrict this to a single event then this notion becomes an implicit axiom of the logic, changing the semantics i.e. the formula $(\forall x, y)\Diamond(\mathtt{f}(x) \wedge \mathtt{g}(y))$ becomes unsatisfiable.

Some extensions also allow other non-event predicates and functions to appear in the signature. In this case the temporal structure should provide an interpretation for these symbols. This can support the calling of external functions. It would be usual for these interpretations to be constant throughout the trace. A specific case of this is when those predicates and functions are taken from a particular *theory* as described next.

Modulo Theories There has been a lot of recent interest in automated reasoning in first-order logic, also referred to as *Satisfiability Modulo Theories* (SMT). The general idea is to extend first order logic with theories for particular sub-domains (i.e. arithmetic, arrays, datatypes, etc.), and build decision procedures specific to those domains. The same concept can be extended to reasoning in first-order temporal logic. SMT can specifically be applied in runtime verification, as described in [25], which presents an approach for *monitoring modulo theories*, which relies on an SMT solver to discharge the data related obligations in each state.

4.7 Register Automata and Freeze Quantification

Register automata [40] and freeze quantifiers in temporal logic [26] are systems based on the notion of *registers*, in which data observed in the trace can be stored, and later read and compared with other data in the trace. A register automaton [40] has in addition to the traditional control states also a finite set of registers. Data observed in the trace can be stored in registers when encountered, and can later be read for comparison with other data. Register automata form a subclass of extended finite state machines where the registers play the role of the variables. Similarly, *freeze quantifiers* [26] are used in temporal logic to capture "storing" of values. The formula $\downarrow_r \varphi$ stores the data value at the current position in the trace in register r (actually it stores an equivalence class denoting all those positions having an equivalent value), and evaluates the formula with that register assignment. The unfreeze formula $\uparrow_r$ checks whether the data value at the current position is equivalent to that in register r. As an example, the following temporal property using quantification:

$$(\forall f)\Box(\mathtt{open}(f) \rightarrow \Diamond\mathtt{close}(f))$$

can instead be formulated as follows using a freeze quantifier:

$$\Box(\texttt{open} \rightarrow \downarrow_r \Diamond(\texttt{close} \wedge \uparrow_r))$$

Such registers here can be seen as the equivalent of the pattern matching solutions found in systems such as JLO [59] and TraceContract [5], and correspond to quantification over the current time point (see page 20). Register-based systems are typically studied for their theoretical properties, and are usually somewhat limited. For example is it usually only possible to compare data for equality. Register automata have been used within runtime verification [35].

5 Conclusion

Our discussion has centered around state machines, regular expressions, and temporal logics, and how data can be integrated in such. Important systems have due to lack of space been left out of this discussion, including context free grammars, variations of the μ-calculus, rule-based systems, stream processing, and process algebras. We have, however, hopefully succeeded in illustrating important parts of the design space for runtime verification logics. It has been pointed out that formulas in a logic can be used to identify good traces (positive formulations) or bad traces (negative formulations), and that the succinctness of specifications can depend on this choice. Furthermore, such formulas can be matched against the entire trace, or just against a suffix of the trace. Negative formulations usually go with suffix matching. The useful distinction between next-states and skip-states in state machines has also been pointed out. For writing formalized requirements for a project, the positive formulation over total traces is probably to be preferred, whereas negative formulations over suffixes can be more succinct in some cases. It has been illustrated how different base logics appear advantageous for particular examples, a fact that is not too surprising. How to (whether to) handle data is a crucial problem in the design of a runtime verification logic, and alternative approaches have been promoted in the literature. Parametric trace slicing has so far shown the most efficient [56, 9], although initially causing limited expressiveness.

If we allow ourselves to dangerously imagine an ideal runtime verification logic, it would be a combination of regular expressions (allowing to conveniently express sequencing) and future and past time temporal logic (often resulting in succinct specifications). However, the notion of states, as found in state machines and rule systems is important as well. The ability to distinguish between next-states and skip-states seems useful, in state machines as well as in regular expressions. It is interesting that state machines with anonymous states (where intermediate states are not named) is a formalism very related to future time temporal logic. It would be useful if convenient shorthands for formulas and aggregation operators could be user-defined. A logic should support time, scopes, and should allow for modularizing specifications. The Eagle logic [4], based on a linear μ-calculus with future and past time operators as well as a sequencing operator, was an attempt to support many of these ideas. Eagle allowed for user defined temporal operators, including the standard Linear Temporal Logic operators.

Working with engineers has shown that current practice to write trace checkers consists of programming in high-level scripting/programming languages, such as for ex-

ample Python. Observing the kind of checks performed on such traces suggests that a monitoring logic needs to be rather expressive, and probably Turing complete for practical purposes, allowing for example advanced string processing features and arithmetic computations. Some specification logics have been developed as APIs in programming languages, in the realization of these observations. The distinction between formal specification in a domain-specific logic on the one hand, and programming in a general purpose programming language on the other hand, might get blurred in the field of runtime verification due to the practical needs of monitoring systems. We also expect to see more systems that compute data from traces rather than just produce Boolean-like verdicts.

References

1. *XTL Manual*. http://cadp.inria.fr/man/xtl.html.
2. C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. *SIGPLAN Not.*, 40:345–364, October 2005.
3. H. Barringer, Y. Falcone, K. Havelund, G. Reger, and D. E. Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In *FM*, pages 68–84, 2012.
4. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *VMCAI*, pages 44–57, 2004.
5. H. Barringer and K. Havelund. TraceContract: A Scala DSL for trace analysis. In *Proc. of the 17th International Symposium on Formal Methods (FM'11)*, volume 6664 of *LNCS*, pages 57–72, 2011.
6. H. Barringer, D. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: from EAGLE to RuleR. *J Logic Computation*, 20(3):675–706, June 2010.
7. M. M. Bartetzko D., Fischer C. and W. H. Jass - Java with assertions. In *Proc. of the 1st Int. Workshop on Runtime Verification (RV'01), Paris, France*, volume 55(2) of *ENTCS*, pages 103–117. Elsevier, July 2001.
8. E. Bartocci, B. Bonakdarpour, and Y. Falcone. First international competition on software for runtime verification. In *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, pages 1–9, 2014.
9. E. Bartocci, Y. Falcone, B. Bonakdarpour, C. Colombo, N. Decker, K. Havelund, Y. Joshi, F. Klaedtke, R. Milewicz, G. Reger, G. Rosu, J. Signoles, D. Thoma, E. Zalinescu, and Y. Zhang. First international competition on runtime verification: rules, benchmarks, tools, and final results of crv 2014. *International Journal on Software Tools for Technology Transfer*, pages 1–40, 2017.
10. D. Basin, F. Klaedtke, S. Marinovic, and E. Zălinescu. Monitoring of temporal first-order properties with aggregations. *Formal Methods in System Design*, 2015.
11. D. Basin, F. Klaedtke, S. Müller, and B. Pfitzmann. Runtime monitoring of metric first-order temporal properties. In *Proceedings of the 28th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 2 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 49–60. Schloss Dagstuhl - Leibniz Center for Informatics, 2008.
12. A. Bauer, R. Goré, and A. Tiu. A first-order policy language for history-based transaction monitoring. In *Proceedings of the 6th International Colloquium on Theoretical Aspects of Computing (ICTAC)*, volume 5684 of *Lect. Notes Comput. Sci.*, pages 96–111. Springer, 2009.

13. A. Bauer, J. Küster, and G. Vegliach. The ins and outs of first-order runtime verification. *Form. Method. Syst. Des.*, 46(3):286–316, 2015.
14. A. Bauer, M. Leucker, and C. Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In *Proceedings of the 7th international conference on Runtime verification*, RV'07, pages 126–138, Berlin, Heidelberg, 2007. Springer-Verlag.
15. A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for ltl and tltl. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, Sept. 2011.
16. S. Bensalem and K. Havelund. Dynamic deadlock analysis of multi-threaded programs. In *Haifa Verification Conference, Haifa, Israel, November 13-16, 2005*, volume 3875 of *Lect. Notes Comput. Sci.*, pages 208–223. Springer, 2006.
17. D. Bianculli, C. Ghezzi, and P. San Pietro. *The Tale of SOLOIST: A Specification Language for Service Compositions Interactions*, pages 55–72. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
18. F. Chen and G. Roşu. MOP: An Efficient and Generic Runtime Verification Framework. In *Object-Oriented Programming, Systems, Languages and Applications(OOPSLA'07)*, pages 569–588. ACM press, 2007.
19. K. T. Cheng and A. S. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *Proceedings of the 30th International Design Automation Conference*, DAC '93, pages 86–91, New York, NY, USA, 1993. ACM.
20. J. Chomicki, D. Toman, and M. H. Böhlen. Querying ATSQL databases with temporal logic. *ACM Trans. Database Syst.*, 26(2):145–178, 2001.
21. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Proceedings of the Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, Yorktown Heights, New York, May 1981. Springer.
22. C. Colombo, G. J. Pace, and G. Schneider. Larva — safer monitoring of real-time java programs (tool paper). In *Proceedings of the 2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*, SEFM '09, pages 33–37, Washington, DC, USA, 2009. IEEE Computer Society.
23. B. D'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna. LOLA: Runtime monitoring of synchronous systems. In *Proceedings of the 12th International Symposium on Temporal Representation and Reasoning*, pages 166–174. IEEE Computer Society, 2005.
24. N. Decker, M. Leucker, and D. Thoma. jUnit^{RV}—adding runtime verification to jUnit. In G. Brat, N. Rungta, and A. Venet, editors, *NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings*, volume 7871 of *Lecture Notes in Computer Science*, pages 459–464. Springer, 2013.
25. N. Decker, M. Leucker, and D. Thoma. Monitoring modulo theories. *International Journal on Software Tools for Technology Transfer*, pages 1–21, 2015.
26. S. Demri and R. Lazić. LTL with the freeze quantifier and register automata. *ACM Trans. Comput. Logic*, 10(3):16:1–16:30, Apr. 2009.
27. D. Drusinsky. *Modeling and Verification using UML Statecharts*. Elsevier, 2006. ISBN-13: 978-0-7506-7949-7, 400 pages.
28. C. Eisner and D. Fisman. Temporal logic made practical. 2014.
29. E. A. Emerson. Handbook of theoretical computer science (vol. b). chapter Temporal and Modal Logic, pages 995–1072. MIT Press, Cambridge, MA, USA, 1990.
30. Y. Falcone, J.-C. Fernandez, and L. Mounier. Runtime verification. chapter Runtime Verification of Safety-Progress Properties, pages 40–59. Springer-Verlag, Berlin, Heidelberg, 2009.
31. Y. Falcone, K. Havelund, and G. Reger. A tutorial on runtime verification. In *Engineering Dependable Software Systems*, pages 141–175. 2013.

32. Y. Falcone, D. Nickovic, G. Reger, and D. Thoma. Second international competition on runtime verification - CRV 15. In *Runtime Verification - 15th International Conference, RV 2015, Vienna, Austria, 2015. Proceedings*, volume 9333, pages 365–382, 2015.
33. B. Finkbeiner, S. Sankaranarayanan, and H. Sipma. Collecting statistics over runtime executions. *Formal Methods in System Design*, 27(3):253–274, 2005.
34. M. J. Fischer and R. E. Ladner. Propositional dynamic logic of regular programs. *J. Comput. Syst. Sci.*, 18:194–211, 1979.
35. R. Grigore, D. Distefano, R. L. Petersen, and N. Tzevelekos. Runtime verification based on register automata. In *Tools and Algorithms for the Construction and Analysis of Systems: 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 260–276. Springer Berlin Heidelberg, 2013.
36. S. Hallé and R. Villemaire. Runtime enforcement of web service message contracts with data. *IEEE Trans. Services Computing*, 5(2):192–206, 2012.
37. K. Havelund. Rule-based runtime verification revisited. *International Journal on Software Tools for Technology Transfer*, 17(2):143–170, 2015.
38. K. Havelund and G. Roşu. Efficient monitoring of safety properties. *International Journal on Software Tools for Technology Transfer*, 6(2):158–173, 2004.
39. G. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2004.
40. M. Kaminski and N. Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, Nov. 1994.
41. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proc. of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, volume 2072 of *Lect. Notes Comput. Sci.*, pages 327–353, 2001.
42. M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: A run-time assurance approach for Java programs. *Formal Methods in System Design*, 24(2):129–155, 2004.
43. D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(3):333–354, 1983.
44. O. Kupferman and M. Y. Vardi. Model checking of safety properties. *Form. Methods Syst. Des.*, 19(3):291–314, Oct. 2001.
45. F. Laroussinie, N. Markey, and P. Schnoebelen. Temporal logic with forgettable past. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS'02, pages 383–392, Washington, DC, USA, 2002. IEEE Computer Society.
46. K. G. Larsen and A. Legay. *Statistical Model Checking: Past, Present, and Future*, pages 3–15. Springer International Publishing, 2016.
47. A. Legay, B. Delahaye, and S. Bensalem. Statistical model checking: An overview. In *1st Int. Conference on Runtime Verification (RV'10)*, volume 6418 of *LNCS*. Springer, 2010.
48. M. Leucker and C. Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009.
49. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag New York, Inc., New York, NY, USA, 1995.
50. R. Medhat, B. Bonakdarpour, S. Fischmeister, and Y. Joshi. Accelerated runtime verification of LTL specifications with counting semantics. In *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings*, pages 251–267, 2016.
51. P. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An overview of the MOP runtime verification framework. *J Software Tools for Technology Transfer*, pages 1–41, 2011.
52. OMG. OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1, August 2011.

53. A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.
54. G. Reger. *Automata Based Monitoring and Mining of Execution Traces*. PhD thesis, University of Manchester, 2014.
55. G. Reger, H. C. Cruz, and D. Rydeheard. MarQ: monitoring at runtime with QEA. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15)*, 2015.
56. G. Reger, S. Hallé, and Y. Falcone. Third international competition on runtime verification CRV 2016. In *Runtime Verification - 16th International Conference, RV 2016. Proceedings*, 2016.
57. G. Reger and D. Rydeheard. From first-order temporal logic to parametric trace slicing. In E. Bartocci and R. Majumdar, editors, *Runtime Verification: 6th International Conference, RV 2015, Vienna, Austria, September 22-25, 2015. Proceedings*, pages 216–232. Springer International Publishing, 2015.
58. M. Sipser. *Introduction to the Theory of Computation*. Cengage Learning, Boston, Massachusetts, 3rd edition, 2013.
59. V. Stolz and E. Bodden. Temporal assertions using AspectJ. In *Proc. of the 5th Int. Workshop on Runtime Verification (RV'05)*, volume 144(4) of *ENTCS*, pages 109–124. Elsevier, 2006.
60. R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, Jan. 1986.
61. M. Y. Vardi. *From Church and Prior to PSL*, pages 150–171. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.