Monitoring Events that Carry Data

Klaus Havelund[1]*, Giles Reger[2], Daniel Thoma[3], and Eugen Zălinescu[4]

[1] Jet Propulsion Laboratory, California Inst. of Technology, USA
[2] University of Manchester, UK
[3] Universität zu Lübeck, Germany,
[4] Technische Universität München, Germany

Abstract. Very early runtime verification systems focused on monitoring what we can refer to as propositional events: just names of events. For this, finite state machines, standard regular expressions, or propositional temporal logics were sufficient formalisms for expressing properties. However, in practice there is a need for monitoring events that in addition carry data arguments. This adds complexity to both the property specification languages, and monitoring algorithms, which is reflected in the many alternative such approaches suggested in the literature. This chapter presents five different formalisms and monitoring approaches that support specifications with data, in order to illustrate the challenges and various solutions.

Keywords: Runtime verification, data rich events, temporal logic, state machines, rule systems, stream processing.

1 Introduction

Runtime verification (RV) as a field is broadly defined as focusing on processing execution traces (output of an observed system) for verification and validation purposes, ignoring how the traces are generated, in contrast to testing, where test case (input to observed system) generation is in focus. Of particular interest is the problem of verifying that a sequence of events, a trace, satisfies a temporal property, formulated e.g. as a state machine or temporal logic formula. Applications cover such domains as security monitoring and safety monitoring.

We shall distinguish between two variants of this problem: *propositional* and *parameterised* runtime verification, according to the format of events. In the propositional case, events are atomic without structure, for example simple identifiers, such as *openGate* and *closeGate*. Here we assume a finite (and usually small) alphabet Σ of atomic identifiers. This case resembles the classic finite trace language membership of language theory [52], where properties are stated for example as finite state machines or regular expressions with atomic letters, as in the following regular expression: $(\mathit{openGate}; \mathit{closeGate})^*$. Similarly, the propositional verification problem has

*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.

been studied in model checking [51], where properties for example are stated in Linear Temporal Logic (LTL), and where models are infinite traces of atomic propositions. Very early RV systems supported only this propositional case. Within recent years, however, emphasis within the research community has been on parameterised runtime verification, where events carry data. Here events are drawn from an alphabet $\Sigma \times D^*$ for some possibly infinite value domain D, which includes values occurring in monitored events, for example reals, strings, objects, etc. This chapter reviews five alternative approaches to parameterised runtime verification, covering extensions of temporal logic and automata with quantification, as well as rule-based and stream processing systems.

As an example consider the following (well studied) data parameterised property, which we shall name *UnsafeMapIterator*, and which will be formalised in the different approaches. The property concerns the use of Java collections, which are part of the Java library API. The property requires that if a collection is created from a java.util.Map object (i.e the key set of the map), and then a java.util.Iterator object is created from that collection, and the original map thereafter is updated, then thereafter the next() method cannot be called on that iterator. Four events are relevant: $\mathsf{create}(m,c)$ records the creation of collection c from map m; $\mathsf{iterator}(c,i)$ records the creation of iterator i from collection c; $\mathsf{update}(m)$ records the update of m; and $\mathsf{next}(i)$ records the call of the next() method on iterator i. More complicated properties can easily be imagined, requiring computations to be performed, such as counting, etc. Due to lack of space we shall, however, limit ourselves to this property as a running example.

The chapter presents five formalisms and monitoring approaches, chosen to represent a broad view of the solution space wrt. logics and algorithms. FOTL [16, 17] is a first-order temporal logic, with a monitoring algorithm that has roots in approaches for checking temporal integrity constraints of databases [26]. MMT (Monitoring Modulo Theories) [34] is a generic framework that allows lifting monitor synthesis procedures for propositional temporal logics to a temporal logic over structures within some first-order theory using SMT solving. These first two approaches represent variations of first-order linear temporal logic, a very important class of candidate logics for runtime verification. The two systems also represent different interesting monitoring algorithms for this case. QEA (Quantified Event Automata) [10] are automata supporting quantification over data. The corresponding approach generalises the concept of trace slicing as found in earlier influential RV systems such as TRACEMATCHES [5] and MOP [25, 62]. Trace slicing likely provides the most efficient monitors among state-of-the-art systems. LOGFIRE [47] is a rule-based framework interpreting rules working on a collection of facts. It is implemented using an adaption of the RETE algorithm known from artificial intelligence. It is furthermore implemented as an internal DSL (an API in the Scala programming language). LOLA [29] is a stream-based specification language inspired by Lustre and Esterel. The corresponding approach incrementally constructs output streams from input streams. This is a rather new approach to monitoring.

The chapter is organised as follows. Section 2 introduces preliminary notation. Sections 3 to 7 introduce the five different formalisms and monitoring approaches. Section 8 further discusses and compares the five approaches. Section 9 presents related work, while Section 10 concludes the chapter.

2 Preliminaries

Primitive Types By $\mathbb{B}$ we denote the set of Boolean values $\{\mathsf{true}, \mathsf{false}\}$ together with the usual operators such as $\neg, \wedge, \vee, \rightarrow$. By $\mathbb{N}$ we denote the set of natural numbers $\{0, 1, 2, \ldots\}$ and by $\mathbb{R}$ the set of real numbers. We assume a set of event names $\mathscr{N}$, a set of variable names $\mathscr{V}$, and an infinite domain $\mathbb{D}$ of values (data occurring in events).

Non-Primitive Types A power set type is denoted by $\wp(T)$, denoting the set of all subsets of the type T. Tuple types are denoted by $T_1 \times T_2 \times \ldots \times T_n$, containing elements of the form $(v_1, ..., v_n)$ for $v_i \in T_i$.

By $S \rightarrow T$ we denote the set of total functions from S to T. By $S \nrightarrow T$ we denote the set of partial functions from S to T with a finite domain, also referred to as maps. A map can be explicitly constructed with the notation: $[x_1 \mapsto v_1, ..., x_n \mapsto v_n]$, with $[\,]$ denoting the empty map. A map is applied with the same notation as function application: $m(x_i)$ yielding v_i. The values for which a map m is defined is denoted by $\mathsf{dom}(m)$, resulting in the set $\{x_1, ..., x_n\}$. One map m_1 is overridden by another map m_2 with the notation $m_1 \dagger m_2$. That is, if $m = m_1 \dagger m_2$ then $m(x) = m_2(x)$ if $x \in \mathsf{dom}(m_2)$ else $m(x) = m_1(x)$. Maps are expected to be applied to values in their domain.

By T^* we denote the set of finite sequences over T where each sequence element is of type T. A sequence σ of length N is a function of type $\{n \in \mathbb{N} \mid n < N\} \rightarrow T$. The length of a sequence σ is denoted by $|\sigma|$. The element at position $i \in \mathbb{N}$ in sequence σ is denoted $\sigma(i)$ or σ_i. A sequence can be explicitly constructed using the notation: $\langle v_1, \ldots, v_n \rangle$, with $\langle\ \rangle$ denoting the empty sequence. A non-empty sequence $s = \langle v_1, v_2, \ldots, v_n \rangle$ of type T^* can be deconstructed with the functions $\mathsf{head} : T^* \rightarrow T$ and $\mathsf{tail} : T^* \rightarrow T^*$ as follows: $\mathsf{head}(s) = v_1$ and $\mathsf{tail}(s) = \langle v_2, \ldots, v_n \rangle$. We occasionally write $\bar{v}$ to represent a sequence $\langle v_1, \ldots, v_n \rangle$. Sequences are also referred to as lists.

First-order Logic A *signature* $S = (C, P, \mathsf{ar})$ consists of finite disjoint sets C and P of constant and respectively relation (or predicate) symbols, and a *arity* function $\mathsf{ar} : P \rightarrow \mathbb{N}$. A *term* is a constant $c \in C$ or a variable $x \in \mathscr{V}$.

First-order *formulas* over the signature $S = (C, P, \mathsf{ar})$ are given by the grammar

$$\varphi ::= p(t_1, \ldots, t_{\mathsf{ar}(p)}) \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists x.\, \varphi,$$

where p ranges over P, the t_is range over terms, and x ranges over $\mathscr{V}$. As syntactic sugar, we use standard Boolean constants and connectives such as *true*, *false*, $\wedge$, $\rightarrow$, and the universal quantifier $\forall x$. The set of *free variables* of a formula φ, that is, those that are not in the scope of some quantifier in φ, is denoted by $\mathit{fv}(\varphi)$. A *sentence* is a formula without free variables.

A *structure* $\mathscr{S}$ over the signature S consists of a (finite or infinite) *domain* $D \neq \emptyset$ and *interpretations* $c^{\mathscr{S}} \in D$ and $p^{\mathscr{S}} \subseteq D^{\mathsf{ar}(r)}$, for each $c \in C$ and $p \in P$. Given a structure with domain D, a *valuation* is a mapping $\theta : \mathscr{V} \rightarrow D$. For a valuation θ, $\bar{x} = (x_1, \ldots, x_n) \in \mathscr{V}^n$, and $\bar{d} = (d_1, \ldots, d_n) \in D^n$, we write $\theta[\bar{x} \mapsto \bar{d}]$ for the valuation that maps x_i to d_i, for $1 \leq i \leq n$, and leaves the other variables' valuation unaltered. We abuse notation and apply a valuation θ also to constants, with $\theta(c) = c^{\mathscr{S}}$, for all $c \in C$. The semantics of first-order formulas is defined as usual. We write $(\mathscr{S}, \theta) \models \varphi$ if a formula φ is satisfied for some structure $\mathscr{S}$ and valuation θ.

Events and Traces An *event* is a tuple $(id, \langle v_1, \ldots, v_n \rangle)$ consisting of a name $id \in \mathscr{N}$ and a sequence of values $v_i \in \mathbb{D}$. An event is typically written as $id(v_1, \ldots, v_n)$. The type of events is denoted by $\mathbb{E} = \mathscr{N} \times \mathbb{D}^*$. The type of (event) *traces* is denoted by $\mathbb{E}^*$.

The Monitoring Problem We will focus on the following problem: given some specification language $\mathscr{L}$, find a procedure $M : \mathscr{L} \to (\mathbb{E}^* \to \mathit{Verdict})$, that for any specification $\varphi \in \mathscr{L}$ and any trace $\tau \in \mathbb{E}^*$, computes a verdict $M(\varphi)(\tau)$ indicating whether the trace τ satisfies the specification φ or not. Note, however, that *Verdict* generally can be any data domain, including the traditional case of Booleans or some extension of Booleans. Such a procedure normally processes the trace iteratively, event by event, keeping state between iterations. A verdict is consequently issued for each new event, and not just at the end of the trace. Thus, *Verdict* typically includes a special verdict with the meaning "unknown verdict" or "no definitive verdict (yet)." We refer to such a procedure as a *monitoring algorithm*.

3 Monitoring First-order Temporal Properties

3.1 Overview

First-order temporal logics are natural specification languages for formalising requirements of hardware and software systems. In particular, the first-order aspect is well-suited to capture relations between data and quantify over data. While first-order temporal logics are not widely used in verification because of decidability issues [50], they do admit efficient monitoring.

In this section we present a monitoring approach for the past-only fragment of first-order temporal logic (FOTL). The presentation is a stripped-down version, due to limited space, of the approaches in [16, 17], given for richer logics, which additionally include future temporal operators, quantitative temporal constraints to express deadlines, interpreted functions like arithmetic operators, rigid predicates like inequality, and SQL-like aggregation operators. In a nutshell, the monitoring algorithm is based on a translation of formulas in a fragment of FOTL into relational algebra expressions. The algorithm is implemented in the MONPOLY tool [15].

To get a glimpse of the specification language, we formalise next the *UnsafeMapIterator* property. To each event we associate a corresponding predicate symbol. Then the following FOTL formula represents a possible formalisation.

$$\Box \forall i. \Big(\mathsf{next}(i) \to \exists m, c. \left(\neg \mathsf{update}(m) \,\mathsf{S}\, \left(\mathsf{iterator}(c, i) \land \blacklozenge \mathsf{create}(m, c) \right) \right) \Big)$$

The formula requires that always,[5] for any iterator i, if this iterator is used, then there are a map m and a collection c such that (a) at some previous time point the iterator i was created from collection c, (b) before that, the collection c was created from the map m, and (c) since the iterator's creation, the map m has not been updated.

[5] Since we restrict ourselves to the past-only fragment of FOTL, the outermost temporal operator $\Box$ ("always") is not part of our definition of the logic given in Section 3.2. However, we include it in the formalisation to emphasise that the property must be fulfilled at all time points.

3.2 Syntax and Semantics

FOTL *formulas* over the signature $S = (C, P, \mathsf{ar})$ are given by the grammar

$$\varphi ::= p(t_1, \dots, t_{\mathsf{ar}(p)}) \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists x.\, \varphi \mid \CIRCLE\, \varphi \mid \varphi \,\mathsf{S}\, \varphi$$

where p ranges over P, the t_is range over $C \cup \mathcal{V}$, and x ranges over $\mathcal{V}$. The symbols $\CIRCLE$ and S denote the "previous" and the "since" temporal operators. Intuitively, the formula $\CIRCLE\, \varphi$ states that φ holds at the previous time point, while the formula $\varphi \,\mathsf{S}\, \psi$ states that there is a time point in the past where ψ holds and from the next time point and onwards the formula φ continuously holds. As syntactic sugar, besides the one for first-order logic, we use the temporal operator $\blacklozenge$ ("once") with $\blacklozenge\, \varphi := \mathit{true} \,\mathsf{S}\, \varphi$.

A *temporal structure* over the signature S is a sequence $\overline{\mathcal{T}} = (\mathcal{T}_0, \mathcal{T}_1, \dots)$ of structures over S such that

(1) all structures $\mathcal{T}_i$, with $i \geq 0$, have the same domain, denoted D, and
(2) constant symbols have rigid interpretation: $c^{\mathcal{T}_i} = c^{\mathcal{T}_0}$, for all $c \in C$ and $i > 0$.

We call the indices of the elements in the sequence $\overline{\mathcal{T}}$ *time points*. In a temporal structure, predicates may have different interpretations at different time points. As detailed later, predicates and their interpretations are used to represent events. Recall that there are no function symbols (beside constants) to be interpreted.

Definition 1. *Let $\overline{\mathcal{T}}$ be a temporal structure over the signature S, with $\overline{\mathcal{T}} = (\mathcal{T}_0, \mathcal{T}_1, \dots)$, φ a formula over S, θ a valuation, and $i \in \mathbb{N}$. We define the satisfaction relation $(\overline{\mathcal{T}}, \theta, i) \models \varphi$ inductively as follows:*

$$\begin{array}{lll}
(\overline{\mathcal{T}}, \theta, i) \models p(t_1, \dots, t_{\mathsf{ar}(p)}) & \mathit{iff} & \big(\theta(t_1), \dots, \theta(t_{\mathsf{ar}(p)})\big) \in p^{\mathcal{T}_i}, \\
(\overline{\mathcal{T}}, \theta, i) \models \neg\psi & \mathit{iff} & (\overline{\mathcal{T}}, \theta, i) \not\models \psi, \\
(\overline{\mathcal{T}}, \theta, i) \models \psi \vee \psi' & \mathit{iff} & (\overline{\mathcal{T}}, \theta, i) \models \psi \text{ or } (\overline{\mathcal{T}}, \theta, i) \models \psi', \\
(\overline{\mathcal{T}}, \theta, i) \models \exists x.\, \psi & \mathit{iff} & (\overline{\mathcal{T}}, \theta[x \mapsto d], i) \models \psi, \text{ for some } d \in D, \\
(\overline{\mathcal{T}}, \theta, i) \models \CIRCLE\, \psi & \mathit{iff} & i > 0 \text{ and } (\overline{\mathcal{T}}, \theta, i-1) \models \psi, \\
(\overline{\mathcal{T}}, \theta, i) \models \psi \,\mathsf{S}\, \psi' & \mathit{iff} & \text{for some } j \leq i,\ (\overline{\mathcal{T}}, \theta, j) \models \psi', \text{and} \\
 & & \quad (\overline{\mathcal{T}}, \theta, k) \models \psi, \text{ for all } k \text{ with } j < k \leq i.
\end{array}$$

For a temporal structure $\overline{\mathcal{T}}$, a time point $i \in \mathbb{N}$, and a formula φ with the vector $\bar{x}$ of free variables, let $[\![\varphi]\!]^{(\overline{\mathcal{T}}, i)} := \big\{\bar{d} \in D^{|fv(\varphi)|} \mid (\overline{\mathcal{T}}, \theta[\bar{x} \mapsto \bar{d}], i) \models \varphi\big\}$. The set $[\![\varphi]\!]^{(\overline{\mathcal{T}}, i)}$ consists of the satisfying elements of φ at time point i in $\overline{\mathcal{T}}$. Instead of $[\![\varphi]\!]^{(\overline{\mathcal{T}}, i)}$ we write $[\![\varphi]\!]^i$ when $\overline{\mathcal{T}}$ is clear from the context.

3.3 Monitoring Algorithm

Setup We assume that property formalisations are of the form $\Box \forall \bar{x}.\, \varphi$, where φ is an FOTL formula and $\bar{x}$ is the sequence of φ's free variables. The property requires that $\forall \bar{x}.\, \varphi$ holds at every time point in the temporal structure $\overline{\mathcal{T}}$ representing the monitored system's behaviour. Moreover, we assume that $\overline{\mathcal{T}}$ has domain $\mathbb{D}$ and it is a *temporal database*, i.e. the relation $p^{\mathcal{T}_i}$ is finite, for any $p \in P$ and $i \in \mathbb{N}$.

The inputs of the monitoring algorithm are a formula ψ, which is logically equivalent to $\neg\varphi$, and a temporal database $\overline{\mathscr{T}}$, which is processed iteratively. That is, at each iteration $i \geq 0$, the monitor processes the structure $\mathscr{T}_i$. The algorithm outputs, again iteratively, the relation $[\![\psi]\!]^i$, for each $i \geq 0$. As ψ and $\neg\varphi$ are equivalent, the tuples in $[\![\psi]\!]^i$ represent the property violations at time point i. Note that we drop the topmost universal quantifier, since an instantiation of the free variables $\bar{x}$ that satisfies ψ provides additional information about the violation. Note that the property is satisfied if and only if the output at each iteration is the empty set.

Remark 1. Given an event trace τ, we build a temporal database as follows, assuming that all events with the same name have the same number of arguments. We also assume a signature (C, P, ar) with $\mathscr{N} \subseteq P$, and arities of predicate symbols matching those of the corresponding events names. The temporal database $\overline{\mathscr{T}}$ is built as follows: if at position i, with $0 \leq i < |\tau|$ the event $e(d_1, \ldots, d_n)$ occurs then $e^{\mathscr{T}_i} = \{(d_1, \ldots, d_n)\}$ and $p^{\mathscr{T}_i} = \emptyset$, for any $p \in P$ with $p \neq e$. For all $i \geq |\tau|$ we take $p^{\mathscr{T}_i} = \emptyset$, for all $p \in P$.

Note that, since we are considering here the past-only fragment of FOTL, structures at time points $j > i$ are irrelevant for the evaluation at time point i. Thus, when monitoring a trace τ, the algorithm is stopped after iteration $|\tau| - 1^6$.

Example 1. We illustrate this setup on the *UnsafeMapIterator* property. Consider the following event sequence:

$$\mathsf{create}(\mathsf{m}, \mathsf{c}_1).\mathsf{create}(\mathsf{m}, \mathsf{c}_2).\mathsf{iterator}(\mathsf{c}_1, \mathsf{i}_1).\mathsf{update}(\mathsf{m}).\mathsf{iterator}(\mathsf{c}_2, \mathsf{i}_2).\mathsf{next}(\mathsf{i}_1)$$

The corresponding temporal database contains the interpretations $\mathsf{create}^{\mathscr{T}_0} = \{(\mathsf{m}, \mathsf{c}_1)\}$, $\mathsf{create}^{\mathscr{T}_1} = \{(\mathsf{m}, \mathsf{c}_2)\}$, and $\mathsf{create}^{\mathscr{T}_i} = \emptyset$, for $i \in \{2, 3, \ldots\}$, etc.

Let φ be the formula from Section 3.1 (page 4) formalising the *UnsafeMapIterator* property with the $\Box$ operator and the $\forall$ quantifier stripped off. Furthermore let $\gamma(i)$ be the consequent of the implication in φ and let $\psi(i) := \mathsf{next}(i) \wedge \neg\gamma(i)$. We thus have $\psi(i) \equiv \neg\varphi(i)$. One can check that $[\![\psi]\!]^i = \emptyset$, for $i \in \{0, \ldots, 4\}$, and that $[\![\psi]\!]^5 = \{(\mathsf{i}_1)\}$, meaning that there are no violations at time points 0 to 4, and there is a violation at time point 5, for iterator i_1.

Remark 2. Note that when encoding event traces as temporal databases, the interpretation of predicate symbols are always either empty or singleton relations. This need not be the case in arbitrary temporal databases. For instance, the relations at a time point could contain the tuples involved in a database transaction.

Monitorable Fragment The computation of $[\![\psi]\!]^i$ is by recursion over ψ's formula structure, using relational algebra operators to compute the evaluation of a formula from

[6] When considering specifications with a future dimension, see [17], we require that future operators are bounded: they only look boundedly far into the future; this corresponds to hard-time specifications, and can be specified with metric temporal constraints; that is in Metric FOTL [53]. Note that the approach thus handles a safety fragment of (Metric) FOTL. Then, to handle a finite trace, since it is assumed that time is observed by the monitoring algorithm only through event timestamps, a new dummy event with a sufficiently large timestamp is added at the end of the trace, and the algorithm is stopped after observing this last event.

the evaluation of its direct subformulas, possibly from previous time points. Not all formulas in FOTL are effectively monitorable, since unrestricted use of logic operators may require infinite relations to be built during evaluation. Thus the algorithm is only able to deal with formulas from the following *monitorable fragment* of FOTL, which consists of the formulas ψ that satisfy the following conditions:

1. $fv(\alpha) = fv(\beta)$, for any subformula of ψ of the form $\alpha \vee \beta$;
2. $fv(\alpha) \subseteq fv(\beta)$, for any subformula of ψ of the form $\beta \wedge \neg\alpha$,[7] $\alpha \,\mathsf{S}\, \beta$, and $\neg\alpha \,\mathsf{S}\, \beta$;
3. a subformula of the form $\neg\alpha$ can only appear as part of a subformula of the form $\beta \wedge \neg\alpha$ or $\neg\alpha \,\mathsf{S}_I\, \beta$.

This set of syntactic restrictions on ψ ensure in particular that $[\![\psi]\!]^i$ is finite, for any $i \in \mathbb{N}$. Consider for instance the non-monitorable formula $\psi = p(x) \vee q(y)$. Given that $\mathbb{D}$ is infinite, there are infinitely many tuples $(a,b) \in \mathbb{D}^2$ that satisfy ψ, at any time point i, namely all tuples in $(p^{\mathcal{T}_i} \times \mathbb{D}) \cup (\mathbb{D} \times q^{\mathcal{T}_i})$. For example, if $p(a)$ holds at i (i.e. $a \in p^{\mathcal{T}_i}$), then, *for any* $b \in \mathbb{D}$, the formula $p(a) \vee q(b)$ holds at i, i.e. $(a,b) \in [\![\psi]\!]^i$.

The MonPoly tool implements a set of heuristics to rewrite non-monitorable formulas into monitorable formulas. While these heuristics have proved to be effective in practice, they are often not necessary as it is usually easy to directly express a domain-independent formula $\neg\varphi$ as an equivalent monitorable formula ψ. For instance, for $\varphi = p(x,y) \rightarrow \blacklozenge(q(x) \vee r(y)))$, the non-monitorable formula $\psi = p(x,y) \wedge \neg \blacklozenge(q(x) \vee r(y))$ can be rewritten to the monitorable formula $(p(x,y) \wedge \neg \blacklozenge q(x)) \vee (p(x,y) \wedge \neg \blacklozenge r(y))$.

Algorithm We start with some definitions. A *table* is a tuple $(R,\bar{x})$, written $R_{\bar{x}}$, where $R \subseteq \mathbb{D}^k$ is a relation and $\bar{x}$ is a sequence of k variables, for some $k \in \mathbb{N}$. Given tables A and B and variable sequence $\bar{x}$, we denote by $\sigma_C(A)$, $\pi_{\bar{x}}(A)$, $A \bowtie B$, $A \rhd B$, and $A \cup B$, the relational algebra operators *selection*, *projection*, *(natural) join*, *antijoin*, and respectively *union* applied to tables A and B, where C is a set of constraints of the form $t = t'$, for $t,t' \in C \cup \mathcal{V}$. We refer to textbooks on databases, e.g. [4], for their definitions.

Example 2. Let $A_{\langle x,y\rangle}$, $B_{\langle y,z\rangle}$, $C_{\langle y\rangle}$ be tables with $A = \{(1,2),(1,4),(3,4)\}$, $B = \{(2,5),(2,6)\}$, and $C = \{4\}$. We have $A_{\langle x,y\rangle} \bowtie B_{\langle y,z\rangle} = \{(1,2,5),(1,2,6)\}_{\langle x,y,z\rangle}$, $A_{\langle x,y\rangle} \rhd C_{\langle y\rangle} = \{(1,2)\}_{\langle x,y\rangle}$, $\sigma_{x=3}(A_{\langle x,y\rangle}) = \{(3,4)\}_{\langle x,y\rangle}$, and $\pi_{\langle y\rangle}(A_{\langle x,y\rangle}) = \{2,4\}_{\langle y\rangle}$.

Next, the free variables of a formula α are used as attributes of the relation $[\![\alpha]\!]^i$. We write $[\![\alpha]\!]^i_{\bar{x}}$ for the table $([\![\alpha]\!]^i)_{\bar{x}}$, where $\bar{x}$ is the vector of free variables of α. The following equalities express in our notation the standard correspondence, known as Codd's theorem, between first-order logic and relational algebra.

$$\begin{aligned} [\![\alpha \wedge \beta]\!]^i_{\bar{z}} &= [\![\alpha]\!]^i_{\bar{x}} \bowtie [\![\beta]\!]^i_{\bar{y}} & \qquad [\![\alpha \vee \beta]\!]^i_{\bar{x}} &= [\![\alpha]\!]^i_{\bar{x}} \cup [\![\beta]\!]^i_{\bar{x}} \\ [\![\alpha \wedge \neg\beta]\!]^i_{\bar{z}} &= [\![\alpha]\!]^i_{\bar{x}} \rhd [\![\beta]\!]^i_{\bar{y}} & \qquad [\![\exists y'.\, \alpha]\!]^i_{\bar{x}'} &= \pi_{\bar{x}'}[\![\alpha]\!]^i_{\bar{x}} \end{aligned}$$

where α and β are monitorable formulas with free variables $\bar{x}$ and respectively $\bar{y}$, $\bar{z}$ is the sequence $\bar{x}$ concatenated with the subsequence of $\bar{y}$ of variables not in $\bar{x}$, and $\bar{x}'$ is the subsequence of $\bar{x}$ without the variable y'. For instance, if $\alpha = p(x_1,x_2)$ and

[7] Note that here we treat the operator $\wedge$ as a primitive.

$\beta = q(x_2,x_3)$, then $\bar{x} = \langle x_1,x_2\rangle$, $\bar{y} = \langle x_2,x_3\rangle$, and $\bar{z} = \langle x_1,x_2,x_3\rangle$. We have omitted the equation for predicates $p(\bar{t})$, which is straightforward but tedious, and uses the selection and projection operators. E.g., if x is a variable and a is a constant, then $\llbracket p(x,a)\rrbracket^i_{\langle x\rangle} = \pi_{\langle x\rangle}(\sigma_{\{y=a\}}(p^{\mathcal{T}_i}_{\langle x,y\rangle}))$. Note also that when $\bar{x} = \bar{y}$, then the join (i.e. $\bowtie$) and antijoin (i.e. $\rhd$) operations are identical to the set intersection (i.e. $\cap$) and respectively set difference (i.e. $\setminus$) operations.

We now consider the evaluation of formulas ψ that have temporal operators as their main connective. In contrast to the first-order connectives, their evaluation at a time point depends on the evaluation of their subformulas at previous time points. The evaluation of $\llbracket\psi\rrbracket^i$ for $i > 0$ is based on the following equalities:

$$\llbracket \bullet\,\alpha\rrbracket^i_{\bar{x}} = \llbracket\alpha\rrbracket^{i-1}_{\bar{x}} \qquad \llbracket \alpha\,\mathsf{S}\,\beta\rrbracket^i_{\bar{y}} = \llbracket\beta\rrbracket^i_{\bar{y}} \cup \big(\llbracket\alpha\,\mathsf{S}\,\beta\rrbracket^{i-1}_{\bar{y}} \bowtie \llbracket\alpha\rrbracket^i_{\bar{x}}\big)$$

where α, β, $\bar{x}$, and $\bar{y}$ are as in the previous set of equations. For $i = 0$, we have $\llbracket\bullet\,\alpha\rrbracket^i_{\bar{x}} = \emptyset_{\bar{x}}$ and $\llbracket\alpha\,\mathsf{S}\,\beta\rrbracket^i_{\bar{y}} = \llbracket\beta\rrbracket^i_{\bar{y}}$. A similar equality is used for formulas of the form $\neg\alpha\,\mathsf{S}\,\beta$, replacing the join with the antijoin. To accelerate the computation of $\llbracket\psi\rrbracket^i$, the monitoring algorithm maintains state for each temporal subformula, storing previously computed intermediate results. Namely, the algorithm stores between the iterations $i-1$ and i, when $i > 0$, the relation $\llbracket\alpha\rrbracket^{i-1}$ and respectively $\llbracket\alpha\,\mathsf{S}\,\beta\rrbracket^{i-1}$. By storing these relations, the subformulas α and β need not be evaluated again at time points $j < i$ during the evaluation of ψ at time point i.

It is straightforward to translate the previous equalities into a bottom-up evaluation procedure of $\llbracket\varphi\rrbracket^i$, for $i \in \mathbb{N}$. We note that relational algebra operators have standard, efficient implementations [42], which can be used to evaluate the right-hand side relational algebra expressions.

Example 3. We present next a partial run of the algorithm for the property UnsafeMapIterator on the event sequence from Example 1. The formulas ψ and γ are as in Example 1. We also let the subformulas β, β', and γ' be as follows:

$$\psi(i) = \mathsf{next}(i) \land \neg\exists m,c.\, \big(\underbrace{\neg\mathsf{update}(m)\,\mathsf{S}\,\underbrace{\big(\mathsf{iterator}(c,i) \land \underbrace{\blacklozenge\,\mathsf{create}(m,c)}_{\beta'(m,c)}\big)}_{\beta(c,i,m)}}_{\gamma'(m,c,i)}\big)$$

That is, we have

$$\begin{aligned} \beta'(m,c) &:= \blacklozenge\,\mathsf{create}(m,c), & \gamma(i) &= \exists m,c.\,\gamma'(m,c,i),\\ \beta(c,i,m) &:= \mathsf{iterator}(c,i) \land \beta'(m,c), & \psi(i) &= \mathsf{next}(i) \land \neg\gamma(i).\\ \gamma'(m,c,i) &:= \neg\mathsf{update}(m)\,\mathsf{S}\,\beta(c,i,m), & & \end{aligned}$$

The algorithm computes the set $\llbracket\psi\rrbracket^j$ of violations, for $j \in \mathbb{N}$, based on the following equalities:

$$\llbracket\beta'\rrbracket^j_{\langle m,c\rangle} = \begin{cases} \mathsf{create}^{\mathcal{T}_j}_{\langle m,c\rangle} \cup \llbracket\beta'\rrbracket^{j-1}_{\langle m,c\rangle} & \text{if } j > 0\\ \mathsf{create}^{\mathcal{T}_j}_{\langle m,c\rangle} & \text{if } j = 0 \end{cases} \qquad \begin{aligned} \llbracket\gamma\rrbracket^j_{\langle i\rangle} &= \pi_{\langle i\rangle}(\llbracket\gamma'\rrbracket^j_{\langle c,i,m\rangle})\\ \llbracket\psi\rrbracket^j_{\langle i\rangle} &= \mathsf{next}^{\mathcal{T}_j}_{\langle i\rangle} \rhd \llbracket\gamma\rrbracket^j_{\langle i\rangle} \end{aligned}$$

j	$\mathsf{next}^{\mathscr{T}_j}$	$\mathsf{update}^{\mathscr{T}_j}$	$\mathsf{iterator}^{\mathscr{T}_j}$	$\mathsf{create}^{\mathscr{T}_j}$	$[\![\beta']\!]^j$	$[\![\beta]\!]^j$	$[\![\gamma']\!]^j$	$[\![\gamma]\!]^j$	$[\![\psi]\!]^j$
0	$\emptyset$	$\emptyset$	$\emptyset$	$\{(\mathsf{m},\mathsf{c}_1)\}$	$\{(\mathsf{m},\mathsf{c}_1)\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1	$\emptyset$	$\emptyset$	$\emptyset$	$\{(\mathsf{m},\mathsf{c}_2)\}$	B	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
2	$\emptyset$	$\emptyset$	$\{(\mathsf{c}_1,\mathsf{i}_1)\}$	$\emptyset$	B	$\{(\mathsf{c}_1,\mathsf{i}_1,\mathsf{m})\}$	$\{(\mathsf{c}_1,\mathsf{i}_1,\mathsf{m})\}$	$\{\mathsf{i}_1\}$	$\emptyset$
3	$\emptyset$	$\{\mathsf{m}\}$	$\emptyset$	$\emptyset$	B	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
4	$\emptyset$	$\emptyset$	$\{(\mathsf{c}_2,\mathsf{i}_2)\}$	$\emptyset$	B	$\{(\mathsf{c}_2,\mathsf{i}_2,\mathsf{m})\}$	$\emptyset$	$\emptyset$	$\emptyset$
5	$\{\mathsf{i}_1\}$	$\emptyset$	$\emptyset$	$\emptyset$	B	$\emptyset$	$\emptyset$	$\emptyset$	$\{\mathsf{i}_1\}$

Table 1. Relations computed by the monitoring algorithm for a sample trace.

$$[\![\beta]\!]^j_{\langle c,i,m\rangle} = \mathsf{iterator}^{\mathscr{T}_j}_{\langle c,i\rangle} \bowtie [\![\beta']\!]^j_{\langle m,c\rangle}$$

$$[\![\gamma']\!]^j_{\langle c,i,m\rangle} = \begin{cases} [\![\beta]\!]^j_{\langle c,i,m\rangle} \cup \left([\![\gamma']\!]^{j-1}_{\langle c,i,m\rangle} \triangleright \mathsf{update}^{\mathscr{T}_j}_{\langle c,i\rangle}\right) & \text{if } j > 0 \\ [\![\beta]\!]^j_{\langle c,i,m\rangle} & \text{if } j = 0 \end{cases}$$

Concretely, the algorithm computes the relations $[\![\alpha]\!]^j$, shown in Table 1, at iterations $j \in \{0,\ldots,5\}$, for $\alpha \in \{\beta',\beta,\gamma',\gamma,\psi\}$, where $B = \{(\mathsf{m},\mathsf{c}_1),(\mathsf{m},\mathsf{c}_2)\}$. We recall that the relations $[\![\beta']\!]^j$ and $[\![\gamma']\!]^j$ are stored by the algorithm between iterations, while all other relations are discarded.

4 Monitoring Modulo Theories

4.1 Overview

For propositional temporal logics such as LTL or CaRet [6] monitoring has been studied extensively and appropriate semantics and monitor synthesis procedures have been developed [20, 22, 35, 31, 21]. Monitoring Modulo Theories (MMT) is a general framework for lifting any of these logics, their semantics and the corresponding synthesis algorithms from the propositional setting to the setting of data values and data constraints. To achieve this, it introduces a general notion of *temporal logic*, capturing many well known propositional temporal logics such as LTL, RLTL [54] or CaRet [6], and the notion of *data logic* based on first-order theories to express constraints over data without any temporal aspects. Next, it combines the two logics into one, *temporal data logic*, whose semantics clearly separates the time and data aspects. This separation gives rise to a monitoring procedure that combines classical monitoring of propositional temporal properties with SMT solving. In this section we present a simplified version of the framework, instantiated for a particular data logic, namely the logic of equality constraints. We refer to [33, 34] for the general framework and for more details. We note that the restriction to equality constraints is a significant restriction and also means that a full SMT solver is not needed. The approach is implemented in the jUnit$^{\text{RV}}$ tool [32].

For a brief illustration, consider the *UnsafeMapIterator* property. Its temporal aspect can be modelled naturally using LTL. Its data aspect can be modelled easily using equality constraints. Combining LTL as a temporal logic and the logic of equality constraints as a data logic results in a formalism that is well suited to model our example property, using for instance the following formula.

$$\forall c,m,i.\, \Box(\mathsf{create}(m,c) \to \bigcirc\Box(\mathsf{iterator}(c,i) \to \bigcirc\Box(\mathsf{update}(m) \to \bigcirc\Box\neg\mathsf{next}(i))))$$

Due to the simplicity of the data aspect in this example the formula does not contain any explicit first-order constraint. However, the first-order subformulas $\mathsf{create}(m,c)$, $\mathsf{iterator}(c,i)$, $\mathsf{update}(m)$, and $\mathsf{next}(i)$ are seen as atomic propositions for the temporal logic, and they give raise to equality constraints at run time. For instance, if the event $\mathsf{next}(\mathsf{i}_1)$ occurs, then the constraint $i = \mathsf{i}_1$ is generated.

4.2 Syntax and Semantics

We will define a *temporal data logic* (TDL) as the extension of a *temporal logic* (TL) from the propositional setting to the first-order setting. There are two main differences between TDL and FOTL from the previous section: first, TDL is parameterised by the propositional temporal logic, and second, it has a finite not an infinite trace semantics.

We assume that the temporal logic is given over some finite non-empty set AP of atomic propositions, its models are finite traces over $\Sigma := \wp(AP)$, and its truth values are elements of a complete lattice $(\mathbb{V}, \sqsubseteq)$ (that is, it does not necessarily have a Boolean semantics).[8] Thus, the semantics of a TL formula ψ is given as a function $[\![\psi]\!] : \Sigma^* \to \mathbb{V}$.

To define TDL in terms of TL, we fix a first-order signature S (see page 3), a finite set $X \subseteq \mathscr{V}$ of variables, and a finite set $F := \{\chi_1, \ldots, \chi_n\}$ of first-order formulas over S with free variables from X. The set F constitutes TL's set AP of "atomic propositions." That is, TDL and TL view the same set differently: TDL considers its elements as formulas and TL views them as propositions i.e. it is agnostic to their structure.[9]

A TDL formula φ consists of a TL *core formula* ψ over AP and a sequence of preceding *global first-order quantifiers* binding free variables in ψ. Formally, the syntax of TDL formulas φ is defined according to the grammar

$$\varphi ::= \exists x.\varphi \mid \forall x.\varphi \mid \psi$$

where $x \in \mathscr{V}$ is a variable and ψ is a TL formula over F.

Example 4. We illustrate how the formula given in Section 4.1 can be seen as a TDL formula. We take $S = (\emptyset, \{\mathsf{next}, \mathsf{update}, \mathsf{iterator}, \mathsf{create}, =\}, \mathsf{ar})$, with ar as expected, $X = \{m, c, i\}$, and $F = \{\mathsf{cr}, \mathsf{it}, \mathsf{u}, \mathsf{n}\}$, where $\mathsf{cr} := \mathsf{create}(m,c)$, $\mathsf{it} := \mathsf{iterator}(c,i)$, $\mathsf{u} := \mathsf{update}(m)$, and $\mathsf{n} := \mathsf{next}(i)$. Then $\varphi = \forall c, m, i.\, \psi$, with

$$\psi = \Box(\mathsf{cr} \to \bigcirc\Box(\mathsf{it} \to \bigcirc\Box(\mathsf{u} \to \bigcirc\Box\neg\mathsf{n})))$$

TDL formulas are interpreted over finite sequences of first-order structures[10] over the signature S, with the same domain $\mathbb{D}$, and with finite intepretations of predicate symbols. We also assume that S contains the equality predicate symbol $=$, interpreted rigidly, that is, all structures interpret equality in the same way, as expected. The original approach from [33] generalises this assumption and presents a setting where data

[8] A complete lattice is a partial order $(M, \sqsubseteq)$ where every subset $N \subseteq M$ has a least upper bound $\sqcup N$ and a greatest lower bound $\sqcap N$.

[9] A one-to-one mapping from F to AP can be defined, but we refrain to do so, for simplicity.

[10] These relate directly to the notion of event, as in Section 3 (see Remark 1). E.g., the event $\mathsf{create}(\mathsf{m}_1, \mathsf{c}_1)$ would be represented as the structure interpreting create as the set $\{(\mathsf{m}_1, \mathsf{c}_1)\}$.

constraints can be expressed in a so-called *data logic*, not only through equality, but through richer first-order theories; see [34] for details. We let $\mathit{\Gamma}$ denote the set of all such first-order structures, which we call *observations*.

Finally, to define the TDL semantics, we also need a way to project a sequence of observations from $\mathit{\Gamma}$ into a sequence of letters from $\mathit{\Sigma}$. To this end, we define next, for a valuation $\theta : \mathscr{V} \to \mathbb{D}$, the projection function $\pi_\theta : \mathit{\Gamma} \to \mathit{\Sigma}$ as follows.

$$\pi_\theta(\gamma) := \{\chi \in F \mid (\gamma, \theta) \models \chi\}$$

That is, the projection of a first-order structure γ is the set of formulas in F that are true in that structure for θ. Recall that such formulas can be viewed as propositional symbols in the temporal logic as there is a direct mapping between F and AP.

We define the semantics of a TDL formula φ as a mapping $[\![\varphi]\!]_\theta : \mathit{\Gamma}^* \to \mathbb{V}$, with respect to a valuation $\theta : \mathscr{V} \to \mathbb{D}$, as follows:

$$[\![\exists x.\varphi']\!]_\theta(\bar{\gamma}) := \bigsqcup_{d\in\mathbb{D}} [\![\varphi']\!]_{\theta[x\mapsto d]}(\bar{\gamma}), \qquad [\![\forall x.\varphi']\!]_\theta(\bar{\gamma}) := \bigsqcap_{d\in\mathbb{D}} [\![\varphi']\!]_{\theta[x\mapsto d]}(\bar{\gamma}),$$

$$[\![\psi]\!]_\theta(\bar{\gamma}) := [\![\psi]\!](\pi_\theta(\bar{\gamma}))$$

where ψ is a TL formula, $\sqcap$ and $\sqcup$ denote the meet and respectively the join of the lattice $(\mathbb{V}, \sqsubseteq)$, and π_θ is extended to sequences as expected: $\pi_\theta(\gamma_1 \ldots \gamma_n) = \pi_\theta(\gamma_1)\ldots\pi_\theta(\gamma_n)$. If φ is a *sentence*, that is, it does not contain any free variable, we omit to annotate a specific valuation θ and write $[\![\varphi]\!]$ for its semantics. This is well-defined since valuations of variables that do not occur freely in φ do not affect its semantics. Note also that the $[\![\cdot]\!]$ notation is overloaded; however, its meaning will be clear from the context.

In examples, we use LTL$_3$ [22] as a concrete TL. We recall it briefly: $[\![\psi]\!](\overline{\sigma}) = \mathsf{true}$, if $\overline{\sigma}\bar{\tau} \models \psi$ for any $\bar{\tau} \in \mathit{\Sigma}^\omega$, $[\![\psi]\!](\overline{\sigma}) = \mathsf{false}$, if $\overline{\sigma}\bar{\tau} \not\models \psi$ for any $\bar{\tau} \in \mathit{\Sigma}^\omega$, and $[\![\psi]\!](\overline{\sigma}) = ?$ otherwise, where $\bar{\tau} \models \psi$ denoted the standard, infinite trace LTL semantics and $\mathbb{V} = (\{\mathsf{false}, \mathsf{true}, ?\}, \sqsubseteq)$ with $\mathsf{false} \sqsubseteq ? \sqsubseteq \mathsf{true}$. Other examples of TLs are RLTL [54], CaRet [6], and versions of LTL with finite trace semantics, see e.g. [37].

Example 5. We illustrate the TDL semantics on the formula from Example 4. To this end, we recall the trace from Example 1:

$$\mathsf{create}(\mathsf{m}_1,\mathsf{c}_1).\mathsf{create}(\mathsf{m}_1,\mathsf{c}_2).\mathsf{iterator}(\mathsf{c}_1,\mathsf{i}_1).\mathsf{update}(\mathsf{m}_1).\mathsf{iterator}(\mathsf{c}_2,\mathsf{i}_2).\mathsf{next}(\mathsf{i}_1)$$

The sequence $\bar{\gamma}$ of observations modelling this trace is obtained as in Section 3 (see Example 1). Table 2 presents the projections $\overline{\sigma}$ of $\bar{\gamma}$ obtained for some valuations θ, and the corresponding verdicts for ψ on $\overline{\sigma}$, where m', c', and i' denote arbitrary values from $\mathbb{D}$ different from m_1, from c_1 and c_2, and from i_1 and i_2, respectively. As expected, we have $[\![\varphi]\!](\bar{\gamma}) = \mathsf{false}$, by taking the meet of all the values $[\![\psi]\!](\overline{\sigma})$.

4.3 Monitoring Algorithm

Preliminaries A *symbolic monitor* $\mathscr{M} = (Q, \mathit{\Sigma}, \delta, q_0, \lambda, \mathbb{V})$ is a state machine with output, where Q is a finite set of states, $\mathit{\Sigma}$ and $\mathbb{V}$ are as defined in Section 4.2, $\delta : Q \times \mathit{\Sigma} \to Q$ is a transition function, $q_0 \in Q$ is the initial state, and $\lambda : Q \to \mathbb{V}$ is a labelling

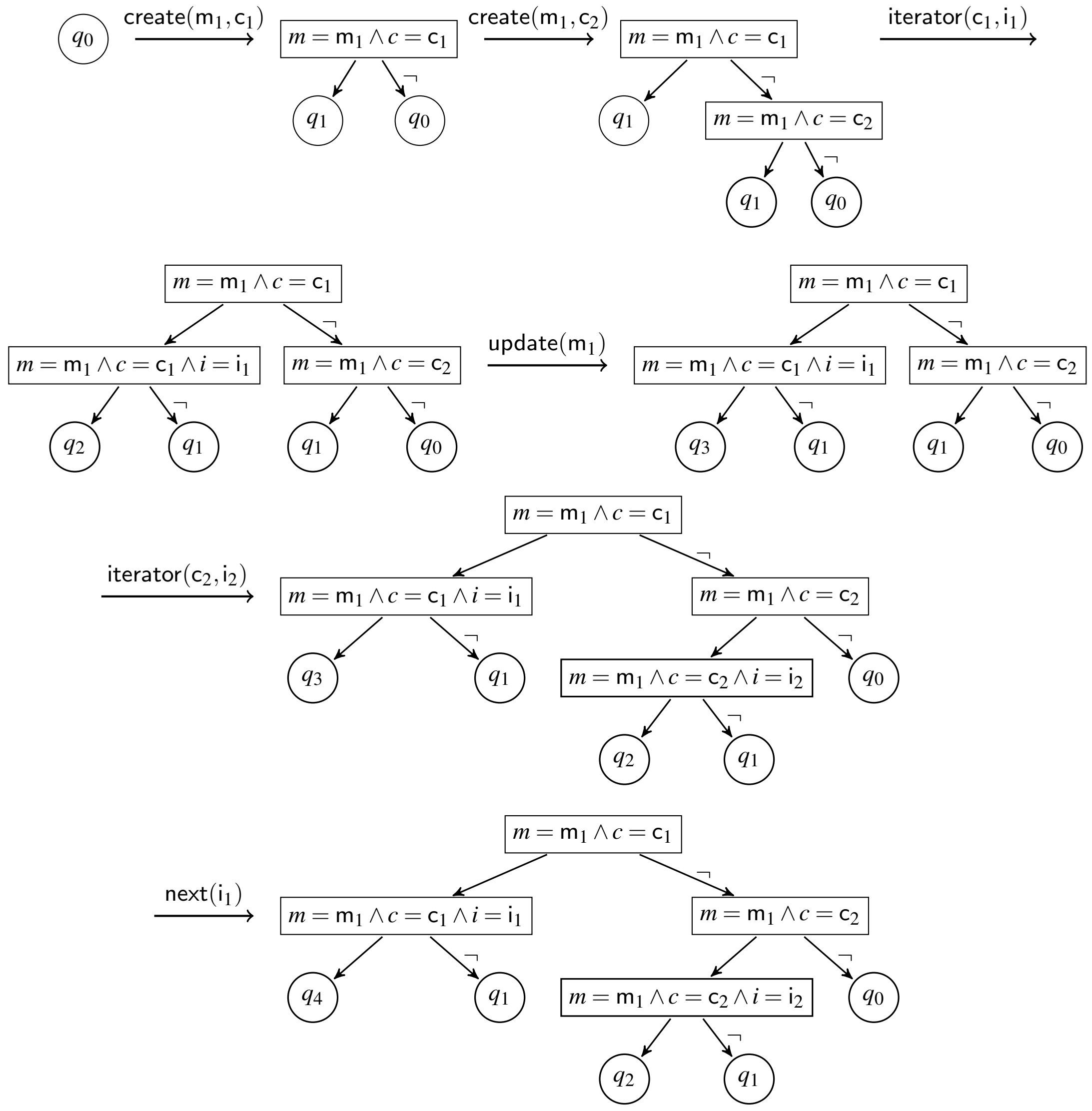


Fig. 1. Constraint trees built by the monitoring algorithm for a sample trace.

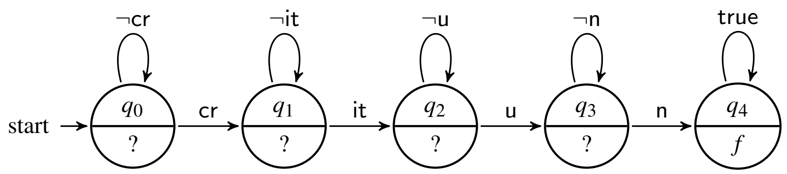


Fig. 2. Symbolic monitor for the formula $\square(\mathsf{cr} \to \bigcirc\square(\mathsf{it} \to \bigcirc\square(\mathsf{u} \to \bigcirc\square\neg\mathsf{n})))$.

θ	$\overline{\sigma} := \pi_\theta(\bar{\gamma})$	$[\![\psi]\!](\overline{\sigma})$
$[m \mapsto \mathsf{m}_1, c \mapsto \mathsf{c}_1, i \mapsto \mathsf{i}_1]$	$\{\mathsf{cr}\}.\emptyset.\{\mathsf{it}\}.\{\mathsf{u}\}.\emptyset.\{\mathsf{n}\}$	false
$[m \mapsto \mathsf{m}_1, c \mapsto \mathsf{c}_1, i \mapsto \mathsf{i}_2]$	$\{\mathsf{cr}\}.\emptyset.\emptyset.\{\mathsf{u}\}.\emptyset.\{\mathsf{n}\}$	?
$[m \mapsto \mathsf{m}_1, c \mapsto \mathsf{c}_1, i \mapsto \mathsf{i}']$	$\{\mathsf{cr}\}.\emptyset.\emptyset.\emptyset.\emptyset.\emptyset.$	?
...		
$[m \mapsto \mathsf{m}_1', c \mapsto \mathsf{c}', i \mapsto \mathsf{i}']$	$\emptyset.\emptyset.\emptyset.\emptyset.\emptyset.\emptyset$	?

Table 2. Evaluation of a TL core formula over a trace for various valuations.

function mapping states to verdicts from $\mathbb{V}$. It is assumed that for a given TL, there is a monitor generation procedure which, for any TL formula ψ builds a monitor $\mathscr{M}$ such that $\lambda(\delta(q_0, \overline{\sigma})) = [\![\psi]\!](\overline{\sigma})$, for any $\overline{\sigma} \in \Sigma^*$.

A *constraint* is a quantifier-free first-order formula over a signature that contains no predicate symbol except equality.[11] For instance, $x = a \land y \neq b$ is a constraint. Note that such a constraint ρ describes the set Θ_ρ of valuations θ such that $\rho\theta$ holds in the theory of equality. It is easy to see that for any observation (i.e. first-order structure) γ and a first-order formula χ, there exists a constraint, denoted $\hat{\gamma}(\chi)$, such that $(\gamma, \theta) \models \chi$ iff $\hat{\gamma}(\chi)\theta$ holds, for any valuation θ. For instance, if γ is the observation corresponding to the event $\mathsf{iterator}(\mathsf{c}_1, \mathsf{i}_1)$, then $\hat{\gamma}(\chi)$ is $c = \mathsf{c}_1 \land i = \mathsf{i}_1$ for $\chi = \mathsf{iterator}(c, i)$, while $\hat{\gamma}(\chi)$ is false for $\chi = \mathsf{next}(i)$. Note that $\Theta_{\hat{\gamma}(\chi)} = \{\theta \mid (\gamma, \theta) \models \chi\}$ and thus $\hat{\gamma}(\chi)$ can be used to represent the set $\{\theta \mid (\gamma, \theta) \models \chi\}$. This property that will be used in the monitoring algorithm. We also assume a procedure SAT that takes as input a constraint and outputs whether the constraint is satisfiable or not. We recall that the general framework [34] considers arbitrary theories, not only that of equality, as presented here. In general, the SAT procedure is implemented by invoking an SMT solver for the considered theory.

A *constraint tree t* is a finite, non-empty, full binary tree having constraints as inner nodes and monitor states as leafs. A tree is denoted either by $\mathsf{Inner}(\rho, t_1, t_2)$ where ρ is a constraint, and t_1 and t_2 are the root's left and right subtrees respectively, or by $\mathsf{Leaf}(q)$ where $q \in Q$. Each node v in a tree t induces a constraint $\rho(v)$ defined as the conjunction of the constraints on the path from the root to the node v.

Algorithm The monitoring algorithm incrementally processes a sequence $\bar{\gamma}$ of observations in order to compute the semantics of some given TDL formula φ. Let ψ be φ's TL core formula. The algorithm uses the symbolic monitor $\mathscr{M}$ for ψ. Intuitively, the algorithm executes one instance of $\mathscr{M}$ for each projection $\pi_\theta(\bar{\gamma})$, with $\theta : \mathscr{V} \to \mathbb{D}$ some valuation. As there are finitely many projections, they partition the set of valuations into a finite number of equivalence classes. The algorithm maintains a mapping from representatives θ of these classes to states q of $\mathscr{M}$, where $q = \delta(q_0, \pi_\theta(\bar{\gamma}))$. The property that $[\![\psi]\!](\pi_\theta(\bar{\gamma})) = \lambda(q)$ allows the algorithm to compute the verdict associated with the current sequence $\bar{\gamma}$ of observations, by iterating through the verdicts $\lambda(q)$, for q in the image of the mentioned mapping. Indeed, in case all global quantifiers in φ are universal, the verdict is the meet over all verdicts $\lambda(q)$. In general, when existential quantifiers are also present, the computation of the verdict is more involved; see [34].

[11] In the more general framework constraints must contain interpreted predicates only.

Algorithm 1 The monitoring algorithm.

```
proc step($t$, $\gamma$) = traverse($\gamma$, $t$, true)

proc traverse($\gamma$, $t$, $\rho$)
  case $t$ = Inner($\rho'$, $t_1$, $t_2$): Inner($\rho'$, traverse($\gamma$, $t_1$, $\rho \wedge \rho'$), traverse($\gamma$, $t_2$, $\rho \wedge \neg\rho'$))
  case $t$ = Leaf($q$): split($\gamma$, $q$, $F$, $\rho$, $\emptyset$)

proc split($\gamma$, $q$, $P$, $\rho$, $a$)
  if $P = \emptyset$ then Leaf($\delta(q,a)$)
  else
    $\chi$, $P'$ := choose($P$)
    $t_1$ := if SAT($\rho \wedge \hat{\gamma}(\chi)$) then split($\gamma$, $q$, $P'$, $\rho \wedge \hat{\gamma}(\chi)$, $a \cup \{\chi\}$)) else Empty
    $t_2$ := if SAT($\rho \wedge \neg\hat{\gamma}(\chi)$) then split($\gamma$, $q$, $P'$, $\rho \wedge \neg\hat{\gamma}(\chi)$, $a$)) else Empty
    if $t_1$ = Empty then $t_2$ else if $t_2$ = Empty then $t_1$ else Inner($\hat{\gamma}(\chi)$, $t_1$, $t_2$)
```

The mapping from equivalence classes of valuations to states q is represented algorithmically by a constraint tree. Namely, for each leaf node v with state q, the constraint $\rho = \rho(v)$ describes the set Θ_ρ of valuations. By construction, these sets of valuations are the equivalence classes mentioned previously. We briefly describe how the algorithm maintains the constraint tree. The initial constraint tree is $\mathsf{Leaf}(q_0)$. For a new observation $\gamma \in \Gamma$ and a monitor instance at v, if all valuations in Θ_ρ project γ to the same letter $a \in \Sigma$, then the monitor instance changes its state from q to $\delta(q,a)$. Otherwise, if γ is mapped to different letters for different valuations in Θ_ρ, then Θ_ρ is split and new monitor instances are created. More precisely, if for some $\chi \in F$ both $\rho \wedge \hat{\gamma}(\chi)$ and $\rho \wedge \neg\hat{\gamma}(\chi)$ are satisfiable, then there are two valuations $\theta, \theta' \in \Theta_\rho$ such that $(\gamma, \theta) \models \chi$ and $(\gamma, \theta') \not\models \chi$. It follows that $\chi \in \pi_\theta(\gamma)$ while $\chi \notin \pi_{\theta'}(\gamma)$, and thus the projections are different. In this case, a new inner node with constraint $\hat{\gamma}(\chi)$ is created. This procedure is performed for each $\chi \in F$. For each new path to a leaf, the state at the leaf node is updated to $\delta(q,a)$, where a is the projection corresponding to that path. The pseudo-code of the procedure that updates the constraint tree is given in Listing 1.

Example 6. Figure 1 shows a run of the algorithm for the *UnsafeMapIterator* property on the event sequence from Example 5, using the symbolic monitor given in Figure 2.

5 Parametric Trace Slicing

5.1 Overview

Parametric trace slicing was introduced as an attempt to efficiently monitor properties with a notion of behaviour being specified *for each* set of values. The initial focus was on efficient algorithms and the formal descriptions and relation to quantification came later. The fundamental idea is to separate monitoring into two parts. The first part *slices* a trace with respect to the data values it contains by associating each set of values with the subtrace of events containing only those values. The second part separately checks each slice against the same property.

The idea was first introduced by TraceMatches [5] where regular expressions are matched against suffixes of a trace slice. The suffix-matching semantics allowed a simple monitoring algorithm as each potential matching suffix could be monitored separately. This was later generalised to total matching and called *parametric trace slicing* [25, 62] and implemented in the JavaMOP [56] tool. There are various languages based on parametric trace slicing and we have chosen to present the concepts using quantified event automata (QEA) [10, 58]. Work on QEA extend the earlier work on parametric trace slicing by giving a more general quantifier-based notion of acceptance as well as using local free (unquantified) variables to allow the per-slice property to reason about values in the slice (see the *AuctionBidding* example below).

A QEA for the *UnsafeMapIterator* property is given below (left). This specifies three universally quantified variables m, c, and i standing for the map, collection and iterator in the example. What follows is a state machine describing the required behaviour for a single *slice* of the trace with respect to those variables. In this case the state machine describes the set of transitions needed to reach a failure state, i.e. it captures the bad behaviour (this relies on the **skip** state modifier explained below). A slice is a projection of a trace obtained by keeping only events that are relevant to a given instantiation of the quantified variables. Acceptance is defined in terms of which slice are accepted by the state machine i.e. for universal quantification this is all slices.

The *UnsafeMapIterator* property is suited (and often used) for parametric trace slicing as, once the slicing has occurred, the underlying property can be treated propositionally. To demonstrate a case where the slice is not treated propositionally consider an *AuctionBidding* property of an auction bidding site where items are listed with a reserve price and bids are strictly increasing. This is captured by the second QEA below where transitions are extended with optional *guards* and *assignments*. Notice how variables r and a are not quantified, these are used to store the reserve price and current bid amount respectively. As these values are used to evaluate the property they must be preserved in the trace so this can no longer be treated propositionally. Lastly, this second QEA does not use an explicit **failure** state but relies on *state closure* (see later), i.e. it captures good behaviour.

```
qea(UnsafeMapIterator) {
  forall(m,c,i)
  accept skip(start){
    create(m,c) → createdC}
  accept skip(createdC){
    iterator(c,i) → createdI}
  accept skip(createdI){
    update(m) → updated}
  accept skip(updated){
    next(i) → failure}
}
```

```
qea(AuctionBidding) {
  forall(i)
  accept next(start){
    list(i,r) do c:=0 → listed}
  accept next(listed){
    bid(i,a) if a>c do c:=a → listed
    sell(i) if c>r → sold
  }
  accept next(sold){}
}
```

5.2 Syntax and Semantics

In Roşu and Chen's parametric trace slicing theory [25, 62] there is a strong separation between the notion of quantification (although they did not call it quantification) which defines what the slices are, and per-slice acceptance (by so-called *plugin* languages) which decides whether a slice is accepted. We repeat this presentation here with event automata describing the plugin language.

Basic Definitions An *event pattern* is an ordered pair of an event name and a list of parameters where a parameter is a variable in $\mathcal{V}$ or value in $\mathbb{D}$. That is, an event pattern is a tuple $(id, \bar{p}) \in \mathcal{N} \times (\mathcal{V} \cup \mathbb{D})^*$, written $id(\bar{p})$. When an event pattern only contains values in $\mathbb{D}$ it coincides with the notion of an event as defined in Section 2. We write $\mathcal{N}(X)$ for any *event alphabet* that contains event patterns using names in $\mathcal{N}$, variables in $X \subseteq \mathcal{V}$, and values in $\mathbb{D}$. Note that $\mathcal{N}(X)$ does not contain all such event patterns but the ones relevant to the monitored property. For instance, for the *UnsafeMapIterator* property we have $\mathcal{N}(\{m,c,i\}) = \{\mathsf{create}(m,c), \mathsf{iterator}(c,i), \mathsf{update}(m), \mathsf{next}(i)\}$.

A *valuation* $\theta \in Env = \mathcal{V} \nrightarrow \mathbb{D}$ is a map from variables to values. By abusing valuations to treat them as total functions (by implicit extension with the identify function) we can apply valuations to event patterns as follows: given an event pattern $id(p_1, \dots, p_n)$ let $\theta(id(p_1, \dots, p_n)) = id(\theta(p_1), \dots, \theta(p_n))$. An event e and an event pattern ep match if there exists a valuation θ such that $\theta(ep) = e$. Let $\mathsf{matches}(e, ep)$ hold iff e and ep match and let $\mathsf{match}(e, ep)$ be the smallest (wrt $\sqsubseteq$) valuation that matches them (and undefined if they do not match). Two valuations θ and θ' are consistent, written $\mathsf{consistent}(\theta, \theta')$, if for every x in $\mathsf{dom}(\theta) \cap \mathsf{dom}(\theta')$ we have $\theta(x) = \theta'(x)$. We also write $\theta_1 \sqsubseteq \theta_2$ iff $\mathsf{dom}(\theta) \subseteq \mathsf{dom}(\theta')$ and θ_1 and θ_2 are consistent.

A *guard* $g \in Guard = Env \rightarrow \mathbb{B}$ is a predicate on valuations and an *assignment* $\gamma \in Assign = Env \rightarrow Env$ is a function from valuations to valuations. We do not fix a guard or assignment language, but use programming language notation in examples.

Trace Slicing Slicing a trace means projecting it to a subtrace, called a slice, with respect to a valuation, which identifies the relevant events in the trace. An event e is *relevant* to a valuation θ and event alphabet $\mathcal{N}(Z)$ if there is an event pattern in $\mathcal{N}(Z)$ that matches e with respect to θ, i.e.

$$\mathsf{relevant}(e, \theta, \mathcal{N}(\mathsf{Z})) \quad \textit{iff} \quad \exists ep \in \mathcal{N}(Z) : \mathsf{matches}(e, \theta(ep))$$

Trace slicing is then defined as follows. Giving a valuation θ, the θ-*slice* of trace τ with respect to event alphabet $\mathcal{N}(Z)$ is the trace $\tau \downarrow_{\theta}^{\mathcal{N}(Z)}$, defined as follows:

$$\langle\,\rangle \downarrow_{\theta}^{\mathcal{N}(\mathsf{Z})} = \langle\,\rangle \qquad e.\tau \downarrow_{\theta}^{\mathcal{N}(\mathsf{Z})} = \begin{cases} e.(\tau \downarrow_{\theta}^{\mathcal{N}(\mathsf{Z})}) & \text{if } \mathsf{relevant}(e, \theta, \mathcal{N}(\mathsf{Z})), \\ \tau \downarrow_{\theta}^{\mathcal{N}(\mathsf{Z})} & \text{otherwise.} \end{cases}$$

Example 7. Consider the AuctionBidding QEA from page 15 and the trace

$$\tau = \mathsf{list(b,5).bid(b,1).list(d,2).bid(b,2).bid(d,1).sell(d).bid(b,2)}$$

we compute (for $\mathcal{N}(Z) = \{\mathsf{list}(i,r), \mathsf{bid}(i,a), \mathsf{sell}(i)\}$) the trace slices

$$\begin{aligned} \tau \downarrow_{[i \,\mapsto\, \mathsf{b}]}^{\mathcal{N}(\mathsf{Z})} &= \mathsf{list(b,5).bid(b,1).bid(b,2).bid(b,2)} \\ \tau \downarrow_{[i \,\mapsto\, \mathsf{d}]}^{\mathcal{N}(\mathsf{Z})} &= \mathsf{list(d,2).bid(d,1).sell(d)} \end{aligned}$$

Event Automata An event automaton $\mathscr{E}$ over the event alphabet $\mathscr{N}(Z)$ is a tuple $(\mathscr{N}(Z), Q, q_0, \delta, F)$ where Q is a finite set of states with $q_0 \in Q$ an initial state and $F \subseteq Q$ a set of final states, and $\delta \subseteq Q \times \mathscr{N}(Z) \times \mathit{Guard} \times \mathit{Assign} \times Q$ is a finite set of transitions between states labelled with an event pattern, a guard, and an assignment. Furthermore, there are exactly two states that have no outgoing transitions: **`success`** $\in F$ and **`failure`** $\notin F$.

A *configuration* is a tuple $(q, \theta) \in Q \times \mathit{Env}$. A trace τ is in the language of the event automaton $\mathscr{E}$, written $\tau \in \mathscr{L}(\mathscr{E})$, if there exists a state $q \in F$ such that $(q_0, [\,]) \xrightarrow{\tau} (q, \theta)$ for some valuation θ, where $\xrightarrow{\tau}$ is the transitive lifting of $\xrightarrow{e}$ defined by

$$(q,\theta) \xrightarrow{e} \begin{cases} (q', \gamma(\theta')) & \mathit{if} \quad \begin{array}{r} \exists (q, ep, g, \gamma, q') \in \delta : \mathsf{matches}(e, ep) \wedge \\ \theta' = \theta \dagger \mathsf{match}(e, ep) \wedge g(\theta') \end{array} \\ (q, \theta) & \mathit{otherwise} \end{cases}$$

Quantified Event Automata A *quantified event automaton* (QEA) over the event alphabet $\mathscr{N}(X \cup Y)$ is a tuple $(\Lambda, \mathscr{E})$ where $\Lambda \in (\{\forall, \exists\} \times X)^*$ is a sequence of quantifications and $\mathscr{E}$ is an event automata over the same alphabet, with X and Y disjoint sets of *quantified* and *free* variables respectively. The QEA is well-formed if Λ mentions each of the variables in X exactly once.

The domain of a (quantified) variable is derived form a trace τ by matching against event patterns in the alphabet as follows:

$$\mathsf{dom}_\tau(x) := \{\mathsf{match}(e, ep)(x) \mid e \in \tau \wedge ep \in \mathscr{N}(X \cup Y) \wedge \mathsf{matches}(e, ep) \wedge \\ x \text{ is a parameter of } ep\}.$$

A trace τ is accepted by the QEA if $\tau \models_{[\,]} \Lambda . \mathscr{E}$ where $\models_\theta$ is defined as

$$\begin{array}{lll} \tau \models_\theta (\forall x)\Lambda'.\mathscr{E} & \text{iff} & \text{for all } d \text{ in } \mathsf{dom}_\tau(x) \text{ we have } \tau \models_{\theta \dagger [x \mapsto d]} \Lambda'.\mathscr{E} \\ \tau \models_\theta (\exists x)\Lambda'.\mathscr{E} & \text{iff} & \text{for some } d \text{ in } \mathsf{dom}_\tau(x) \text{ we have } \tau \models_{\theta \dagger [x \mapsto d]} \Lambda'.\mathscr{E} \\ \tau \models_\theta \langle\,\rangle.\mathscr{E} & \text{iff} & \tau \downarrow_\theta^{\mathscr{N}(X \cup Y)} \in \mathscr{L}(\mathscr{E}(\theta)) \end{array}$$

where $\mathscr{E}(\theta)$ denotes the instantiation of the event automaton as expected i.e. by replacing variables by values in event patterns, guards and assignments. Informally, checking acceptance using this definition consists of building valuations θ of quantified variables, slicing the trace with respect to θ, checking per-slice acceptance, and finally combining the results to produce a verdict.

To describe QEA textually we rely on the (not formally defined) language used earlier where a state and its transitions may be written as

```
[accept] [next/skip](state){
  $id(p_1,\ldots,p_n)$ if [guard] do [assign] → state list
}
```

The optional **`accept`** modifier captures if the state is in F. The **`next/skip`** modifiers refer to the implicit closure of the state, i.e. what should happen when a transition for an event does not exist; **`next`** closes to a failure state and **`skip`** introduces self-looping transitions. The below semantics assumes **`skip`**. Each transition starting at the state is given with (optional) guards and assignments.

Algorithm 2 A basic monitoring algorithm for QEA.

```
$M \leftarrow [[\,] \mapsto \{(q_0, [\,])\}]$
for event $e \in \tau$ do
    $New \leftarrow \{\theta \mid \forall (x_i \mapsto v_i) \in \theta : \exists ep \in \mathscr{N}(X \cup Y) : (x_i \mapsto v_i) \in \mathsf{match}(e, ep) \wedge x_i \in X\}$
    for $\theta \in \mathsf{dom}(M)$ sorted from largest to smallest do
        $Extensions \leftarrow \big\{\theta \dagger \theta' \mid \theta' \in New \wedge \mathsf{consistent}(\theta, \theta') \wedge \mathsf{relevant}\big(e, \theta \dagger \theta', \mathscr{N}(X \cup Y)\big)\big\}$
        for $\theta_{Ext} \in Extensions$ do
            if $\theta = \theta_{Ext}$ or $\theta_{Ext} \notin \mathsf{dom}(M)$ then
                $C \leftarrow \{(q, \theta_{free}) \mid \exists c \in M(\theta) : c \xrightarrow{e}_{\mathscr{E}(\theta_{Ext})} (q, \theta_{free}) \wedge \mathsf{dom}(\theta_{free}) \cap X = \emptyset\}$
                $M \leftarrow M \dagger [\theta_{Ext} \mapsto C]$
```

Example 8. We can now complete the example given in Example 7 by computing the configurations reached by each slice to obtain the following runs:

$$(\texttt{start}, [\,]) \xrightarrow{\mathsf{list(b,5)}} (\texttt{listed}, [r \mapsto 5, c \mapsto 0]) \xrightarrow{\mathsf{bid(b,1)}}$$

$$(\texttt{listed}, [r \mapsto 5, a \mapsto 1, c \mapsto 1]) \xrightarrow{\mathsf{bid(b,2)}} (\texttt{listed}, [r \mapsto 5, a \mapsto 2, c \mapsto 2])$$

$$\xrightarrow{\mathsf{bid(b,2)}} (\textbf{\texttt{failure}}, [r \mapsto 5, a \mapsto 2, c \mapsto 2])$$

and

$$(\texttt{start}, [\,]) \xrightarrow{\mathsf{list(d,2)}} (\texttt{listed}, [r \mapsto 2, c \mapsto 2]) \xrightarrow{\mathsf{bid(d,1)}}$$

$$(\texttt{listed}, [r \mapsto 2, a \mapsto 1, c \mapsto 2]) \xrightarrow{\mathsf{sell(d)}} (\textbf{\texttt{failure}}, [r \mapsto 2, a \mapsto 1, c \mapsto 2]).$$

As $\mathsf{dom}_\tau(i) = \{\mathsf{b}, \mathsf{d}\}$ and both runs reach the **`failure`** state, we conclude that $\tau \not\models \forall i.\,\mathsf{AuctionBidding}$. The two runs fail as we cannot satisfy the guard needed to take a transition and as the state modifier is **`next`** this leads to implicit failure (not directly captured in the above semantics).

5.3 Monitoring Algorithm

The semantics introduced previously is non-incremental; it is necessary to first extract the domains of quantified variables before slicing and checking the trace. To address this we introduce an incremental algorithm and discuss optimisations.

A Basic Incremental Algorithm Algorithm 2 presents a basic incremental algorithm for monitoring QEA. This assumes a QEA described using the notation discussed previously. The algorithm maintains a mapping M from valuations (of quantified variables) to sets of configurations. The valuations may be *partial* with respect to quantified variables in X as the events building a full valuation may appear incrementally. The algorithm does not show how M can be used to determine a verdict but this follows the definition of acceptance above; in the case of pure universal quantification all configuration sets must contain a final state. This gives a verdict for the current trace prefix, which can be lifted to a four-valued domain providing anticipatory results [10].

Event	**Updates to M**	**Using $\theta \in \mathsf{dom}(M)$**
$\mathsf{create}(\mathsf{m}_1,\mathsf{c}_1)$	$([m \mapsto \mathsf{m}_1, c \mapsto \mathsf{c}_1] \mapsto \mathsf{createdC})$	$[\,]$
$\mathsf{create}(\mathsf{m}_1,\mathsf{c}_2)$	$([m \mapsto \mathsf{m}_1, c \mapsto \mathsf{c}_2] \mapsto \mathsf{createdC})$	$[\,]$
$\mathsf{iterator}(\mathsf{c}_1,\mathsf{i}_1)$	$([m \mapsto \mathsf{m}_1, c \mapsto \mathsf{c}_1, i \mapsto \mathsf{i}_1] \mapsto \mathsf{createdI})$	$[m \mapsto \mathsf{m}_1, c \mapsto \mathsf{c}_1]$
$\mathsf{update}(\mathsf{m}_1)$	$([m \mapsto \mathsf{m}_1, c \mapsto \mathsf{c}_1] \mapsto \mathsf{createdC})$ $([m \mapsto \mathsf{m}_1, c \mapsto \mathsf{c}_2] \mapsto \mathsf{createdC})$ $([m \mapsto \mathsf{m}_1, c \mapsto \mathsf{c}_1, i \mapsto \mathsf{i}_1] \mapsto \mathsf{updated})$	$[m \mapsto \mathsf{m}_1, c \mapsto \mathsf{c}_1]$ $[m \mapsto \mathsf{m}_1, c \mapsto \mathsf{c}_2]$ $[m \mapsto \mathsf{m}_1, c \mapsto \mathsf{c}_1, i \mapsto \mathsf{i}_1]$
$\mathsf{iterator}(\mathsf{c}_2,\mathsf{i}_2)$	$([m \mapsto \mathsf{m}_1, c \mapsto \mathsf{c}_2, i \mapsto \mathsf{i}_2] \mapsto \mathsf{createdI})$	$[m \mapsto \mathsf{m}_1, c \mapsto \mathsf{c}_2]$
$\mathsf{next}(\mathsf{i}_1)$	$([m \mapsto \mathsf{m}_1, c \mapsto \mathsf{c}_1, i \mapsto \mathsf{i}_1] \mapsto \textbf{failure})$	$[m \mapsto \mathsf{m}_1, c \mapsto \mathsf{c}_1, i \mapsto \mathsf{i}_1]$

Fig. 3. Illustrating the updates to M for the *UnsafeMapIterator* example.

For each event, the algorithm first computes any potentially new values for variables in X. Then, for each existing valuation θ in M, it tries to extend θ with this new information and update M accordingly. Key to this approach is the way in which M is iterated over; from the largest valuations to the smallest (wrt $\sqsubseteq$). This ensures that when a new valuation is added it extends the largest existing consistent valuation; this is the principle of *maximality*. Maximality is ensured by the check on line 7 i.e. if this check fails then θ_{Ext} has already been added, possibly earlier in this iteration by extending a larger valuation.

The set C computed on line 8 is the set of new configurations for θ_{Ext}. This uses $\rightarrow_{\mathscr{E}(\theta)}$ i.e. the transition relation for the instantiated event automaton $\mathscr{E}(\theta)$. Importantly, a transition cannot be taken if it captures quantified variables; this may be possible as θ_{Ext} can be partial with respect to X.

Example 9. Figure 3 considers the `UnsafeMapIterator` QEA from page 15 which has the alphabet $\mathscr{N}(X \cup Y) = \{\mathsf{create}(m,c), \mathsf{iterator}(c,i), \mathsf{update}(m), \mathsf{next}(i)\}$ for $X = \{m,c,i\}$ and $Y = \emptyset$. We use the running trace from page 6 and use single states to represent configurations as the property is deterministic without free variables. The table gives the valuation $\theta \in \mathsf{dom}(M)$ used to make the update; note that new valuations follow the previously described notion of maximality. The final event produces a valuation in the **`failure`** state, meaning that the trace is rejected.

Indexing Approaches This basic algorithm is still not efficient enough for effective monitoring as it requires a linear search of M for every event and M can grow very large. One solution is to use an *index* to identify the relevant valuations in M. In the following we describe the *value-based* indexing approach as, whilst other approaches exist [57, 59], this is the most prominent approach in the literature and in use in tools. These alternative approaches also make heavy use of indexing on values and therefore the approach described here is also the most relevant in general.

Value-based indexing was introduced in the JavaMOP tool [56] and uses the values in an event to lookup the valuations in M that the current event is *relevant* to. As motivation consider some examples. When considering valuations possibly occurring at runtime, the event $\mathsf{update}(\mathsf{c}_1)$ is only relevant to valuations that already bind c_1, which could be found via direct lookup. However, to find valuations relevant to $\mathsf{iterator}(\mathsf{c}_1,\mathsf{i}_1)$

we must, e.g., find $[m \mapsto \mathsf{m}_1, c \mapsto \mathsf{c}_1]$ which does not refer to i_1 but refers to more than c_1. Therefore, looking up the valuation or its subparts directly will not suffice.

To implement the necessary lookup a map $U : \mathit{Env} \nrightarrow \wp(\mathit{Env})$ is maintained such that valuations in M are mapped to by their sub-valuations of interest. It can be complex to compute which sub-valuations are required and in the worst case all sub-valuations can be used. Algorithm 2 can be updated to use U by firstly ensuring U is

1. *sub-valuation-closed*: for any $\theta \in \mathsf{dom}(M)$, we have $\theta' \in \mathsf{dom}(U)$ if $\theta' \sqsubset \theta$,
2. *relevance-closed*: for any $\theta' \in \mathsf{dom}(U)$ and $\theta \in \mathsf{dom}(M)$, if $\theta' \sqsubseteq \theta$ then $\theta \in U(\theta')$.

These two conditions ensure that U can be used to find θ given any sub-valuation of θ. Secondly one must ensure that M is

3. *union-closed*: if two consistent valuations are in $\mathsf{dom}(M)$, their union is in $\mathsf{dom}(M)$.

This last condition is already ensured by Algorithm 2. If these properties are maintained (see e.g. [56], Algorithm C) then it is sufficient to update the configurations for valuations in $\{\theta\} \cup U(\theta)$ for each $\theta \in \mathit{New}$ i.e. line 4 of Algorithm 2 becomes

for $\theta \in \mathit{New} \cup \bigcup_{\theta' \in \mathit{New}} U(\theta')$ sorted from largest to smallest **do**

The amount of work needed to process each event is now bounded by the size of θ and $U(\theta)$, which are related to the size of X and density of values in τ (a pathological case could lead to $U(\theta)$ being proportional to the size of τ). This is a significant improvement; in certain cases a previously linear complexity becomes constant i.e. where there is a single quantified variable.

Example 10. After the third event ($\mathsf{iterator}(\mathsf{c}_1, \mathsf{i}_1)$) in the above example, U would contain the following mappings:

$$\begin{array}{ll}
[\,] & \mapsto \{[m \mapsto \mathsf{m}_1, c \mapsto \mathsf{c}_1], [m \mapsto \mathsf{m}_1, c \mapsto \mathsf{c}_2], [m \mapsto \mathsf{m}_1, c \mapsto \mathsf{c}_1, i \mapsto \mathsf{i}_1]\} \\
[m \mapsto \mathsf{m}_1] & \mapsto \{[m \mapsto \mathsf{m}_1, c \mapsto \mathsf{c}_1], [m \mapsto \mathsf{m}_1, c \mapsto \mathsf{c}_2], [m \mapsto \mathsf{m}_1, c \mapsto \mathsf{c}_1, i \mapsto \mathsf{i}_1]\} \\
[i \mapsto \mathsf{i}_1] & \mapsto \{[m \mapsto \mathsf{m}_1, c \mapsto \mathsf{c}_1, i \mapsto \mathsf{i}_1]\} \\
[c \mapsto \mathsf{c}_1] & \mapsto \{[m \mapsto \mathsf{m}_1, c \mapsto \mathsf{c}_1], [m \mapsto \mathsf{m}_1, c \mapsto \mathsf{c}_1, i \mapsto \mathsf{i}_1]\} \\
[c \mapsto \mathsf{c}_2] & \mapsto \{[m \mapsto \mathsf{m}_1, c \mapsto \mathsf{c}_2]\} \\
[m \mapsto \mathsf{m}_1, c \mapsto \mathsf{c}_1] & \mapsto \{[m \mapsto \mathsf{m}_1, c \mapsto \mathsf{c}_1, i \mapsto \mathsf{i}_1]\} \\
[m \mapsto \mathsf{m}_1, i \mapsto \mathsf{i}_1] & \mapsto \{[m \mapsto \mathsf{m}_1, c \mapsto \mathsf{c}_1, i \mapsto \mathsf{i}_1]\} \\
[c \mapsto \mathsf{c}_1, i \mapsto \mathsf{i}_1] & \mapsto \{[m \mapsto \mathsf{m}_1, c \mapsto \mathsf{c}_1, i \mapsto \mathsf{i}_1]\}
\end{array}$$

On event $\mathsf{update}(\mathsf{m}_1)$, $\mathit{New} = \{[m \mapsto \mathsf{m}_1]\}$, and $U([m \mapsto \mathsf{m}_1])$ gives the relevant valuations to update. Then on event $\mathsf{iterator}(\mathsf{c}_2, \mathsf{i}_2)$, $\mathit{New} = \{[c \mapsto \mathsf{c}_2, i \mapsto \mathsf{i}_2]\}$ and we add the required $[m \mapsto \mathsf{m}_1, c \mapsto \mathsf{c}_2, i \mapsto \mathsf{i}_2]$ to M using $[m \mapsto \mathsf{m}_1, c \mapsto \mathsf{c}_2] \in U([c \mapsto \mathsf{c}_2])$ as we did before but without searching M.

As a final note, it is possible to statically (from $\mathscr{N}(X \cup Y)$) detect which entries in U may be used e.g. in this example we know we will never query using m and i together. This information can be used to optimise the entries stored in U.

6 Rule-Based Monitoring

6.1 Overview

Rule systems have been extensively studied within the artificial intelligence community, and used for example in expert systems. It turns out that with slight modifications these systems are applicable to runtime verification. A rule system can abstractly be seen as a collection of *rules*, each of the form: $c_1, \ldots, c_n \Rightarrow a$, consisting of a list of conditions c_i and an action a. A rule system executes on a rule state, referred to here as the *database*, which abstractly can be considered as a set of *facts* (named data records). A condition can for example be a fact pattern or the negation thereof. A rule will *fire* if each pattern on the left-hand side matches a fact in the database (in the case of negation: no matching fact exists), in which case the rule right-hand side executes. Multiple occurrences of a variable on the left-hand side must match the same value. In the case that all conditions on a rule's left-hand side match, producing an environment of bound variables, the right-hand side action is executed, adding and/or deleting facts to and from the database. A special fact is the *error* fact.

We here present LOGFIRE [47], a rule-based monitoring framework implemented as an internal DSL, essentially an API/library, in the SCALA programming language. The *UnsafeMapIterator* property can be formulated as follows:[12]

```
class UnsafeMapIterator extends Monitor {
  val create, iterator, update, next = event
  val createdC, createdI, updated = fact

  r1: create(m,c) ⇒ createdC(m,c)
  r2: createdC(m,c), iterator(c,i) ⇒ createdI(m,i)
  r3: createdI(m,i), update(m) ⇒ updated(i)
  r4: updated(i), next(i) ⇒ error
}
```

The property is expressed as a SCALA class that extends a pre-defined class `Monitor`, which provides all the LOGFIRE features. The `UnsafeMapIterator` class defines four rules, named $\mathtt{r}_1$, ..., $\mathtt{r}_4$. Each rule name is followed by the symbol '`:`' followed by a list of conditions on the left of the '⇒' symbol, and an action on the right. The monitored events are create, iterator, update, and next. An event will only be present in the database long enough to evaluate all the left-hand sides of rules to determine which can fire, followed by the removal of the event, and execution of the right-hand sides. Three facts are generated: $\mathsf{createdC}(m,c)$ representing that the collection c has been extracted from the map m, $\mathsf{createdI}(m,i)$ representing that the iterator i has been extracted from the collection of the map m, and $\mathsf{updated}(i)$ representing that the iterator i no longer is safe to iterate over since the corresponding map has been updated.

Rule $\mathtt{r}_1$ states that upon observation of a $\mathsf{create}(m,c)$ event, a $\mathsf{createdC}(m,c)$ fact is generated. Here m and c are free variables that get bound when the pattern $\mathsf{create}(m,c)$ matches a fact (in this case an event) in the database. These bindings will be passed to $\mathsf{createdC}(m,c)$. Rule $\mathtt{r}_2$ states that upon observation of an $\mathsf{iterator}(c,i)$ event in the presence of a $\mathsf{createdC}(m,c)$ fact, a $\mathsf{createdI}(m,i)$ fact is generated. Similarly for the two remaining rules, noting that `error` denotes the error fact. Note that left-hand sides of rules do not need to refer to events, and can be purely fact-triggered, although this is not the case for the rules $\mathtt{r}_1$, ..., $\mathtt{r}_4$. LOGFIRE furthermore allows to mix rule-based programming and general purpose programming by allowing variables and methods to be declared and used in monitor classes, and by allowing any SCALA code in conditions and as actions. In the following, however, focus will be on the pure rule-based fragment of this language.

[12] The syntax has been modified slightly from SCALA to a more mathematical notation.

6.2 Syntax and Semantics

We present the syntax and semantics of an idealised simple rule-based language named LF illustrating the rule-based capabilities of LogFire[13]. The syntax of LF is defined by the following grammar, where *id* ranges over event and fact names in $\mathscr{N}$ (fact names are assumed included in $\mathscr{N}$), x over variable names in $\mathscr{V}$, v over values in $\mathbb{D}$, and *exp* over expressions (not defined further):

$$\begin{array}{ll}
rs ::= \overline{\pi} & fp ::= id(\bar{p}) \\
\pi ::= id \;:\; \bar{c} \Rightarrow \; \overline{\alpha} & p \;::= x \mid v \\
c \;::= fp \mid \mathbf{not}(fp) \mid \mathbf{when}(exp) & \alpha ::= \mathbf{insert}(id(\overline{exp})) \mid \mathbf{remove}(id) \mid \mathbf{error}
\end{array}$$

The above definition uses meta-variables ranging over types as follows: $rs \in \mathsf{RS}$ (Rule Systems), $\pi \in \mathsf{R}$ (Rules), $c \in \mathsf{C}$ (Conditions), $fp \in \mathsf{FP}$ (Fact Patterns), $p \in \mathsf{P}$ (Parameters), $\alpha \in \mathsf{A}$ (Actions), and $exp \in \mathsf{Exp}$ (Expressions). A rule system *rs* consists of a list of rules. A rule π consists of a name, followed by a non-empty list of conditions, forming the left-hand side of the rule, followed on the right-hand side by a non-empty list of actions. A condition c can be a fact pattern *fp*, corresponding to a fact that must be present in the database; or the negation $\mathbf{not}(fp)$ of a fact pattern, requiring that no matching fact exists; or a filter expression $\mathbf{when}(exp)$, which has to evaluate to true on the names bound so far in the conditions occurring earlier in the rule. A fact pattern *fp* consists of a fact name, and a list of parameter patterns. A parameter pattern p can either be a variable x or a literal value v, such as for example an integer or a string (not further specified). Finally, an action α can be a fact insertion $\mathbf{insert}(id(\overline{exp}))$, where the identifier is the name of the fact and the expression list (an expression can for example represent a computation, such as $x+1$) evaluates to a list of fact arguments; or a fact removal $\mathbf{remove}(id)$, where the identifier is the name of a fact occurring on the rule left-hand side; or an **error** action adding the error fact.

A fact is a tuple $(id, \langle v_1, \ldots, v_n \rangle)$ consisting of a name $id \in \mathscr{N}$ and a sequence of values $v_i \in \mathbb{D}$. A fact is typically written as $id(v_1, \ldots, v_n)$. The type of facts is denoted by $\mathbb{F} = \mathscr{N} \times \mathbb{D}^*$. A database is a set of facts of type $\mathbb{DB} = \wp(\mathbb{F})$. Monitored events are just facts. In order to show intent in later definitions, however, we introduce the type $\mathbb{E} = \mathbb{F}$ to represent events. As mentioned above, a rule is evaluated by first evaluating the left-hand side, resulting in an environment binding free variables occurring in event and fact patterns to values in the actual event and in actual facts. An environment is a map of type $Env = \mathscr{V} \rightharpoonup \mathbb{D}$. Finally, when executing a rule, the result is a change request (D,A) of type $\mathbb{CH} = \mathbb{DB} \times \mathbb{DB}$, consisting of a set of facts D to be deleted and a set of facts A to be added. We will encounter semantic definitions which produce sets of change requests. For this we need a function $\mathsf{merge} : \wp(\mathbb{CH}) \to \mathbb{CH}$, which merges the deleted facts respectively added facts, with the simple definition:

$$\mathsf{merge}(ch) = \left(\bigcup\{D \mid (D,A) \in ch\}, \bigcup\{A \mid (D,A) \in ch\} \right)$$

Definition 2. *Let $\tau \in \mathbb{E}^*$ be a trace and $rs \in \mathsf{RS}$ be a rule system. The relation $\tau \models rs$ (τ satisfies rs) is defined as: $\tau \models rs$ iff $\mathsf{error} \notin T[\![rs]\!](\emptyset)(\tau)$; where the function $T[\![\_]\!]$:*

[13] Providing a full definition of LogFire would be too space consuming for this presentation.

$\mathsf{RS} \rightarrow \mathbb{DB} \rightarrow \mathbb{E}^* \rightarrow \mathbb{DB}$, *here applied to the rule system, an initial empty set of facts, and the trace, is defined in Figure 4.*

Function T (Figure 4) is curried, and is applied to a rule system, a database, and a trace, returning a database with facts deleted and added. Function E evaluates the rule system against a single event. The special bottom value $\perp$ denotes an "error value". Given a type T, the type $T_\perp$ denotes $T \cup \{\perp\}$. Function X evaluates one step of the rule system against a database, which is assumed to contain the event just submitted. Function *Xrec* evaluates the rule system against a database, executing (after the event has been removed) recursively until no rules can fire ($\perp$ returned from a call of X). Function R evaluates a single rule against a database. Function *LHS* evaluates the rule's conditions against a database and the environment obtained so-far by evaluating previous conditions of the rule. Function C evaluates a single condition against a database and an environment. Function *Fdb* evaluates a fact pattern against a database by matching the pattern against each fact in the database. A binding $id \mapsto \kappa$ is introduced, and later used in the semantics of remove actions of the form: **remove**(id) (it is assumed that $\mathbb{D}$ contains facts). Function *Ffact* evaluates a fact pattern against a fact. Function *Args* evaluates a list of fact pattern parameters against a list of actual arguments to a fact. Function *Arg* evaluates a single fact pattern parameter against a single fact argument. Function *RHS* evaluates the right-hand side of a rule, a list of actions, in an environment generated by evaluating the left-hand side. Function A evaluates a single action in an environment, returning a change request. Function *Exp* (not further defined) evaluates an expression in an environment, resulting in a value.

6.3 Monitoring Algorithm

In principle, the semantics shown in Section 6.2 is sufficient for execution[14]. However, it is inefficient in that for each event, we process in function X each rule (rule overhead), in *LHS* each condition (condition overhead), and in *Fdb* each database fact (fact overhead). In the typical RV case the number of rules and conditions are small and fixed but the facts grow, resulting in the fact overhead potentially becoming the main source of inefficiency. These inefficiencies are addressed by the RETE algorithm (although only to some degree in the case of fact overhead), developed by Charles L. Forgy in the 1970s [40], and explained in careful detail in [36]. The name *Rete* means *network* in Latin, and reflects the way rules are represented and facts stored by the algorithm. In order to illustrate the algorithm we shall consider the *UnsafeMapIterator* example. For illustration purposes, we shall add a new rule `r`$_5$ to the rule system, in addition to rule `r`$_4$, to reflect that it is also an error to observe a call of hasNext() on an unsafe iterator:

```
r4: updated(i), next(i) ⇒ error
r5: updated(i), hasNext(i) ⇒ error
```

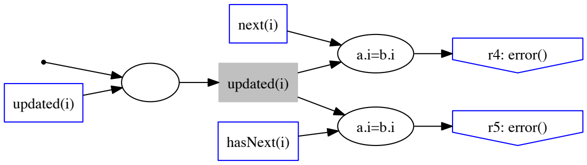
The two rules share the prefix `updated(i)`. They (ignoring here the other rules) are translated into the RETE network shown in Figure 5 (top). This data structure represents the full structure of the rules, and in addition stores all received events and generated facts during monitoring. In general, a RETE network consists of four kinds of nodes:

[14] A Scala version of this semantics has been developed.

```
$T[\![\_]\!] : \mathsf{RS} \to \mathbb{DB} \to \mathbb{E}^* \to \mathbb{DB}$
$T[\![rs]\!](db)(\tau) =$
  if $\tau = \langle\ \rangle$ then $db$ else
    let $db' = E[\![rs]\!](db)(\mathsf{head}(\tau))$ in
    if $\mathsf{error} \in db'$ then $db'$ else
      $T[\![rs]\!](db')(\mathsf{tail}(\tau))$

$E[\![\_]\!] : \mathsf{RS} \to \mathbb{DB} \to \mathbb{E} \to \mathbb{DB}$
$E[\![rs]\!](db)(e) =$
  let $db' = X[\![rs]\!](db \cup \{e\})$ in
  if $db' = \bot$ then $db$ else
    $Xrec[\![rs]\!](db' \setminus \{e\})$

$X[\![\_]\!] : \mathsf{RS} \to \mathbb{DB} \to \mathbb{DB}_\bot$
$X[\![rs]\!](db) =$
  let $(D, A) =$
    $\mathsf{merge}(\{R[\![\pi]\!](db) \mid \pi \in rs\})$
  in
  if $D \cup A = \emptyset$ then $\bot$ else
    $(db \setminus D) \cup A$

$Xrec[\![\_]\!] : \mathsf{RS} \to \mathbb{DB} \to \mathbb{DB}$
$Xrec[\![rs]\!](db) =$
  let $db' = X[\![rs]\!](db)$ in
  if $db' = \bot$ then $db$ else
    $Xrec[\![rs]\!](db')$

$R[\![\_]\!] : \mathsf{R} \to \mathbb{DB} \to \mathbb{CH}$
$R[\![id\ :\ \bar{c} \Rightarrow\ \overline{\alpha}]\!](db) =$
  let $\Theta = LHS[\![\bar{c}]\!](db)([\,])$ in
  $\mathsf{merge}(\{RHS[\![\overline{\alpha}]\!](\theta) \mid \theta \in \Theta\})$

$LHS[\![\_]\!] : \mathsf{C}^* \to \mathbb{DB} \to Env \to \wp(Env)$
$LHS[\![c_1, \ldots, c_n]\!](db)(\theta) =$
  if $n = 0$ then $\{\theta\}$ else
    let $\Theta = C[\![c_1]\!](db)(\theta)$ in
    $\bigcup\{LHS[\![c_2, \ldots, c_n]\!](db)(\theta') \mid \theta' \in \Theta\}$

$C[\![\_]\!] : \mathsf{C} \to \mathbb{DB} \to Env \to \wp(Env)$
$C[\![fp]\!](db)(\theta) = Fdb[\![fp]\!](db)(\theta)$

$C[\![\mathbf{not}(fp)]\!](db)(\theta) =$
  let $\Theta = Fdb[\![fp]\!](db)(\theta)$ in
  if $\Theta = \emptyset$ then $\{\theta\}$ else $\emptyset$

$C[\![\mathbf{when}(exp)]\!](db)(\theta) =$
  if $Exp[\![exp]\!](\theta)$ then $\{\theta\}$ else $\emptyset$
```

```
$Fdb[\![\_]\!] : \mathsf{FP} \to \mathbb{DB} \to Env \to \wp(Env)$
$Fdb[\![id(p_1, \ldots, p_n)]\!](db)(\theta) =$
  $\bigcup\{$
    let $\theta' = Ffact[\![id(p_1, \ldots, p_n)]\!](\kappa)(\theta)$ in
    if $\theta' = \bot$ then $\emptyset$ else
      $\{\theta' \dagger [id \mapsto \kappa]\}$
    $|$
      $\kappa \in db$
  $\}$

$Ffact[\![\_]\!] : \mathsf{FP} \to \mathbb{F} \to Env \to Env_\bot$
$Ffact[\![id_1(p_1, \ldots, p_n)]\!](id_2(v_1, \ldots, v_n))(\theta) =$
  if $id_1 \neq id_2$ then $\bot$ else
    $Args[\![p_1, \ldots, p_n]\!](v_1, \ldots, v_n)(\theta)$

$Args[\![\_]\!] : \mathsf{P}^* \to \mathbb{D}^* \to Env \to Env_\bot$
$Args[\![p_1, \ldots, p_n]\!](v_1, \ldots, v_n)(\theta) =$
  if $n = 0$ then $\theta$ else
    let $\theta' = Arg[\![p_1]\!](v_1)(\theta)$ in
    if $\theta' = \bot$ then $\bot$ else
      $Args[\![p_2, \ldots, p_n]\!](v_2, \ldots, v_n)(\theta \dagger \theta')$

$Arg[\![\_]\!] : \mathsf{P} \to \mathbb{D} \to Env \to Env_\bot$
$Arg[\![p]\!](v)(\theta) =$
  if $p \in \mathscr{V}$ then
    if $p \in \mathsf{dom}(\theta)$ then
      if $\theta(p) = v$ then $\theta$ else $\bot$
    else
      $\theta \dagger [p \mapsto v]$
  else
    if $p = v$ then $\theta$ else $\bot$

$RHS[\![\_]\!] : \mathsf{A}^* \to Env \to \mathbb{CH}$
$RHS[\![\overline{\alpha}]\!](\theta) =$
  $\mathsf{merge}(\{A[\![\alpha]\!](\theta) \mid \alpha \in \overline{\alpha}\})$

$A[\![\_]\!] : \mathsf{A} \to Env \to \mathbb{CH}$
$A[\![\mathbf{insert}(id(exp_1, \ldots, exp_n))]\!](\theta) =$
  $(\emptyset, \{id(Exp[\![exp_1]\!](\theta), \ldots, Exp[\![exp_n]\!](\theta))\})$

$A[\![\mathbf{remove}(id)]\!](\theta) =$
  $(\{\theta(id)\}, \emptyset)$

$A[\![\mathbf{error}]\!](\theta) =$
  $(\emptyset, \{\mathsf{error}\})$

$Exp[\![\_]\!] : \mathsf{Exp} \to Env \to \mathbb{D}$
  $\ldots$
```

Fig. 4. Semantics of LF.



Event	**Rule**	**Added facts**
create(m_1,c_1)	r_1	createdC(m_1,c_1)
create(m_1,c_2)	r_1	createdC(m_1,c_2)
iterator(c_1,i_1)	r_2	createdI(m_1,i_1)
update(m_1)	r_3	updated(i_1)
iterator(c_2,i_2)	r_2	createdI(m_1,i_2)
next(i_1)	r_4	**error**

Fig. 5. Top: RETE network for UnsafeMapIterator rules r_4 and r_5. Bottom: result of applying algorithm to example trace.

- *alpha memories*: white rectangular nodes. There is an alpha memory for each kind of event and fact. When a new event is received or fact generated, it is inserted into the corresponding alpha memory, which can be viewed as a set of events/facts.
- *beta memories*: grey rectangular nodes, containing so-called *tokens*. A token (an alternative representation of what we called an environment in the semantics) is a list of events/facts matching a prefix of one or more rules. The left-most •-node symbolises an initial beta memory (a singular set containing an empty token). This is introduced to make the behaviour of join nodes (see below) uniform.
- *join nodes*: round processing nodes, each connected to an alpha memory and a beta memory, the input nodes, and an output node: a beta memory or an action node. When a fact or token arrives in a connected input alpha or beta memory: the other (beta or alpha) memory is searched for matches. The join node contains the fact pattern of a condition occurring in one or more rules. A match occurs in this example when the alpha node's i parameter equals the beta node's i parameter (in the graph expressed as $a.i = b.i$). Each match results in a new token created from the old token by appending the event/fact from the alpha memory. The new token is sent to the child beta memory or action node.
- *action nodes*: downwards arrow shaped nodes deleting and/or adding facts.

Let's summarise how the algorithm works. When an event is received, it is added to the appropriate alpha memory. This again triggers the connected child join node to execute a search in its connected input beta memory for a matching token, each of which is a list of previous matching facts to a rule prefix. For each such match a new token is generated by appending the event to the input token. The new extended token is then sent to the child beta node or action. If the child node is an action it will execute, and add/remove facts. If the child node is another beta node, then that will again trigger its connected child join node to search its connected input alpha node for matches, etc. Likewise when a fact is added, it is inserted into its appropriate alpha node, and the process is the same as just described.

The application of the semantics in Figure 4 to our trace is shown in Figure 5. For each event is shown which rule it causes to fire, and which fact is added to the database by that rule (in this example no facts are removed). We can illustrate the algorithm using the RETE network as well. Since the network in Figure 5 is partial and only reflects two rules, the illustration will be partial as well. When the $\mathsf{updated}(\mathsf{i}_1)$ fact is generated in the 4th step (in Figure 5), it is inserted in the left-most alpha memory. This triggers the connected child join node to search its connected input beta memory for matches. This is the initial beta memory containing an empty token matching everything, and the $\mathsf{updated}(\mathsf{i}_1)$ fact is thus propagated to the child beta node as the token $\langle\mathsf{updated}(\mathsf{i}_1)\rangle$ (a list containing that fact). The 6th $\mathsf{next}(\mathsf{i}_1)$ event is inserted in the top-most alpha node, which again causes its child join node to search for matches in its input beta memory, which now contains the $\langle\mathsf{updated}(\mathsf{i}_1)\rangle$ token. This is a match, and the token $\langle\mathsf{updated}(\mathsf{i}_1), \mathsf{next}(\mathsf{i}_1)\rangle$ is sent to the $\mathtt{r}_4$ action node, causing an error to be generated.

The RETE algorithm reduces overhead by adding a fact to only the relevant alpha memory, thereby restricting evaluation to that corresponding condition, restricting evaluation to only rules connected to that alpha memory, and restricting evaluation to only that fact. The RETE algorithm optimises situations where two or more rules have a common condition prefix, sharing conditions. The RETE algorithm needs a couple of modifications for runtime verification, however, as described in [47]. First of all, for each update to an alpha or beta memory, the other memory is searched sequentially for matches. This is inefficient in the case of large data volumes in these memories. An indexing approach can address this problem. Second, events need to be handled differently than facts: they should only be around long enough to trigger rules to execute, but should be deleted as soon as this objective is reached. This corresponds to the removal of the event e in the semantic function E in Section 6.2.

7 Stream Processing

7.1 Overview

Runtime verification can be seen as a special case of stream processing, in which the observable system behaviour is represented by a set of input streams, and the monitored property is represented by a (Boolean) output stream of verdicts. The LOLA framework [29] was the first to explicitly cast runtime verification as stream computation. Inspired by functional stream computation languages like Lustre [45] and Esterel [23], LOLA proposed a minimalistic language in which output streams are specified using expressions over the same or other streams. These expressions establish dependencies between the current value of an output stream with values of the same or other streams at the current, past, or future positions. Evaluation is synchronous, i.e. there is a global index into all streams representing the current progress of evaluation. Output streams are not restricted to contain Boolean values and thus the framework goes beyond property checking and allows for quantitative analyses to be carried over, such as computing statistics over the observed system behaviour. In this rest of this section, we present the LOLA framework,[15] mainly following the presentation in [24].

We start by formalising in LOLA the *UnsafeMapIterator* property. This property is somehow unnatural for stream processing, as it considers *event streams* and not *data streams* (i.e. sequences of data values). There are several approaches to encode events as stream elements, and we use one which allows for fewer intermediary streams. Namely, we assume that a stream element is a set of tuples of data values. For input streams, these sets are always singletons, as one element encodes one event. For instance, if at some position j of the monitored trace the event $\mathsf{create}(\mathsf{m}_1,\mathsf{c}_1)$ occurs, then, at position j, the input stream create contains the element $\{(\mathsf{m}_1,\mathsf{c}_1)\}$, otherwise (if a different event type occurs at j) it contains the empty set. Note that we assume here four input streams, one for each event type. The following stream equations specify the property.

$$\begin{aligned}
\mathsf{createdC} &= \mathsf{create} \cup \mathsf{createdC}[-1|\emptyset] \\
\mathsf{createdI} &= (\mathsf{iterator} \bowtie \mathsf{createdC}) \cup \mathsf{createdI}[-1|\emptyset] \\
\mathsf{updated} &= (\mathsf{update} \bowtie \mathsf{createdI}) \cup \mathsf{updated}[-1|\emptyset] \\
\mathsf{ok} &= \mathsf{next} \not\subseteq \pi_{\langle i \rangle}(\mathsf{updated})
\end{aligned}$$

The formalisation uses three intermediary streams, and one output stream, namely ok. Each stream equation represents an equality between the element at the (implicit) position j in the stream on the left hand side of the equation and the elements of the streams occurring on the right hand side of the equation, at positions j' obtained from j by an offset, for any position j in the input streams. The first equation relates, recursively, the jth element of $\mathsf{createdC}$ with the jth element of create and the $(j-1)$th element of $\mathsf{createdC}$ (unless $j=0$, see below). In general, the expression s refers to the value of the stream s at the current position j, and the expression $s[-1|v]$ refers to the value of s at the position with offset -1 with respect to the current position, that is, $j-1$, if $j>0$, and otherwise it refers to the value v, i.e. a default value given after the $|$ symbol. LOLA allows for any computable function to be used for obtaining output stream elements from input stream elements. In this example we use relational algebra operators (see also page 7).[16] Thus, the stream $\mathsf{createdC}$ contains the tuples (m,c) of collections c created so far from maps m. Similarly, the streams $\mathsf{createdI}$ and $\mathsf{updated}$ contain the tuples (c,i,m) of iterators i created so far from collections c, in turn created from maps m; with m referring to updated maps in case of the stream $\mathsf{updated}$. Finally, the stream ok is the Boolean stream representing whether the property is satisfied at the current position, is computed by checking whether the iterator i (if any) is among those for which the corresponding map was updated. We end this section by noting the similarity between this formalisation and the ones QEA and LOGFIRE have used.

7.2 Syntax and Semantics

We assume a finite set of interpreted, typed function symbols f, where f denotes a computable function of some type $T_1 \times \cdots \times T_k \to T$. By abuse of notation, we identify

[15] An implementation can be found at https://www.react.uni-saarland.de/tools/lola/.

[16] We abuse notation and apply them on unnamed relations, as their attributes are as expected, e.g. $\langle m,c \rangle$ for $\mathsf{createdC}$, and $\langle c,i,m \rangle$ for $\mathsf{createdI}$ and $\mathsf{updated}$.

function symbols with their interpretation. Note that 0-ary function symbols, that is, constants, are associated with individual values of some type. We also assume a set of typed *stream variables*.

A *stream of type* T is a finite sequence over T. A stream is *Boolean* if its type is $\mathbb{B}$. For a finite set Z of stream variables, a *stream valuation over* Z is a partial function θ over stream variables assigning to each variable $z \in Z$, a stream $\theta(z)$ such that the streams associated with the different variables in Z have the same length n for some $n \geq 0$. We also say that n is the length of θ, which is denoted by $|\theta|$.

Stream Expressions Given a finite set Z of stream variables, the set of *stream expressions exp of type* T *over* Z is inductively defined by the following syntax:

$$exp := z \mid z[\ell|c] \mid f(exp_1, \ldots, exp_k)$$

where $z \in Z$ is a variable of type T, $\ell \neq 0$ is a non-zero integer, c is a constant of type T, $k \geq 0$ is a positive integer, $f \in F$ is a function symbol of some type $T_1 \times \cdots \times T_k \to T$, and $exp_1, \ldots, exp_k$ are stream expression of type $T_1, \ldots, T_k$, respectively. Informally, $z[\ell|c]$ refers to the value of e at the position obtained from the current position offset by ℓ, and the constant c is the default value assigned to positions from which the offset is after the end or before the beginning of the stream.

Stream expressions e of type T over Z are interpreted over stream valuations θ of type T over Z. The valuation of *exp* with respect to θ, written $[\![exp]\!](\theta)$, is the stream of type T and length $|\theta|$ inductively defined as follows for all $0 \leq i < |\theta|$:

- $[\![z]\!](\theta)(i) = \theta(z)(i)$, for all $z \in Z$,
- $[\![z[\ell|c]]\!](\theta)(i) = \begin{cases} [\![z]\!](\theta)(i+\ell) & \text{if } 0 \leq i+\ell < |\theta|, \\ c & \text{otherwise,} \end{cases}$
- $[\![f(exp_1, \ldots, exp_k)]\!](\theta)(i) = f\big([\![exp_1]\!](\theta)(i), \ldots, [\![exp_k]\!](\theta)(i)\big)$.

Example 11. Consider the Boolean stream expression $exp := x \vee x[1|\mathsf{true}]$ over $\{x\}$. For every stream valuation θ over $\{x\}$ such that $\theta(x) \in (\mathsf{false}\,\mathsf{true})^+$, i.e. alternating false and true. The valuation of *exp* with respect to θ is the Boolean stream $\mathsf{true}^{|\theta|}$, that is, the sequence of length $|\theta|$ where each element is true.

Specification Language Given a finite set X of stream variables and a set $Y = \{y_1, \ldots, y_n\}$, with $n \geq 1$, of stream variables of type $T_1, \ldots, T_n$ respectively, with $X \cap Y = \emptyset$, a LOLA *specification* E *over (input variables)* X *and (output variables)* Y is a set of equations

$$\{y_1 = exp_1, \ldots, y_n = exp_n\}$$

where $exp_1, \ldots, exp_n$ are stream expressions over $X \cup Y$ of type $T_1, \ldots, T_n$ respectively. Note that there is exactly one equation for each output variable.

A *stream valuation of* E is a stream valuation over $X \cup Y$. An *input* (resp. *output*) of E is a stream valuation over X (resp. Y). The LOLA specification E describes a relation, written $[\![E]\!]$, between inputs θ_X of E and outputs θ_Y of E, defined as follows: $(\theta_X, \theta_Y) \in [\![E]\!]$ iff $|\theta_X| = |\theta_Y|$ and for each equation $y = exp$ of E,

$$[\![y]\!](\theta) = [\![exp]\!](\theta)$$

where $\theta = \theta_X \cup \theta_Y$, defined as expected. The stream valuation $\theta_X \cup \theta_Y$ is a *valuation model* of E (associated with the input θ_X) if $(\theta_X, \theta_Y) \in [\![E]\!]$. Note that in general, for a given input θ_X, there may be zero, one, or multiple valuation models associated with θ_X. A LOLA specification E is *well-defined* iff for each input θ_X, there is exactly one valuation model of E associated with θ_X.

A distinction can be made between streams that are meant to represent the output corresponding to some input and intermediate streams that only facilitate the computation of the intended output. Such a distinction is not essential here.

Example 12. We present next an alternative formalisation of the *UnsafeMapIterator* property. We assume the same input streams as in Section 7.1, namely create, iterator, update, and next.[17] Note that their type is of the form $\wp(T_1 \times \ldots \times T_k)$, with $T_1 \ldots, T_k$ among *Map*, *Collection*, *Iterator*. The following stream equations specify the property.

$$\begin{aligned}\mathsf{created} &= \mathsf{create} \cup \mathsf{created}[-1|\emptyset]\\ \mathsf{notupdated} &= \big((\mathsf{iterator} \bowtie \mathsf{created}) \cup \mathsf{notupdated}[-1|\emptyset]\big) \rhd \mathsf{update}\\ \mathsf{ok} &= \mathsf{next} \subseteq \pi_{\langle i \rangle}(\mathsf{notupdated})\end{aligned}$$

The stream ok is the Boolean stream representing the satisfaction or the violation of the property at each position in the event sequence, i.e. $\mathsf{ok}(j) = \mathsf{false}$ iff there is a violation at position j. The auxiliary stream created of type $\wp(\mathit{Map} \times \mathit{Collection})$ stores at position j all tuples (m, c) that have appeared in the input stream create up to (and including) position j. The auxiliary stream notupdated of type $\wp(\mathit{Map} \times \mathit{Collection} \times \mathit{Iterator})$ stores at position j all tuples (c, i, m) such that (a) at some previous position j', the tuples (c, i) and (m, c) appeared in the stream iterator and respectively the stream created, and (b) between position j' and j the tuple (m) has not appeared in the stream update. In other words, notupdated stores those iterators that are safe to call next() on (along with the related map and collection objects).

We note the similarity of this LOLA specification with the computation performed by the FOTL-based monitoring algorithm for the corresponding FOTL specification (see Example 3 on page 8). Namely, there is a direct correspondence between how the streams created, notupdated, and ok are computed and how the satisfying elements of the formulas β', γ', and respectively φ are computed. In particular, given the trace from Example 1 (on page 6), the output streams notupdated and created are given by the columns $[\![\beta']\!]^j$ and respectively $[\![\gamma']\!]^j$ of Table 1 (on page 9), while for ok we have $\mathsf{ok}(i) = \mathsf{true}$ for $i \in \{0, \ldots, 4\}$ and $\mathsf{ok}(5) = \mathsf{false}$.

Well-formed Specifications Note that well-definedness is a semantic restriction on LOLA specifications. To detect ill-defined specifications, like $y = y$ or $y = \neg y$, we present next a syntactic restriction that guarantees well-definedness.

Let E be a LOLA specification over X and Y. A *dependency graph* for E is a weighted and directed multi-graph with vertex set $X \cup Y$. There is an edge (z, z', w) from z to z' with weight w iff the expression *exp* contains $z'[w|c]$ as a subexpression

[17] In examples we do not make a distinction between stream variables and their denoted streams, that is, we identify x and $\theta(x)$.

of *exp*, where *exp* is z's expression in E, i.e. $(z = exp) \in E$. Intuitively, the edge records that the value of (the stream denoted by) z at a particular position depends on the value of z', offset by w positions. Note that there can be multiple edges between z and z' with different weights on each edge. Vertices $x \in X$ have no outgoing edges.

A *walk* of a graph is a sequence $v_1, e_1, v_2, \ldots, v_k, e_k, v_{k+1}$ of vertices and edges, for $k \geq 1$, such that $e_i = (v_i, v_{i+1}, w_i)$, for all i with $1 \leq i \leq k$. The walk is *closed* iff $v_1 = v_{k+1}$. The weight of a walk is the sum of weights of its edges.

A LOLA specification specification is *well-formed* if there is no closed walk with total weight zero in its dependency graph. Every well-formed LOLA specification is well-defined [29]. The converse is not true. For instance, the specification $y = y \wedge \neg y$ is well-defined, but not well-formed.

7.3 Monitoring Algorithm

The monitoring algorithm takes as input a LOLA specification E over X and Y, with E assumed to be well-formed, and an input valuation θ_X of E, which is processed iteratively. That is, at the $(i+1)$st iteration, the monitor receives the values of all input streams at position i, namely $\theta_X(x)(i)$, for $x \in X$. The goal of the monitoring algorithm is to incrementally compute the output valuation θ_Y. Concretely, the monitor outputs at each iteration the newly computed values $\theta_Y(y)(j)$, where $y \in Y$ and $j \in \{0, \ldots, |\theta_X| - 1\}$.

Before presenting the algorithm, we introduce some additional notation. Let $\mathscr{X}_Z$ be the set of variables $\{z_j \mid z \in Z, 0 \leq j \leq |\theta_X|\}$, for $Z \in \{X, Y\}$, and let $\mathscr{X} := \mathscr{X}_X \cup \mathscr{X}_Y$. Given a stream expression *exp* and a position $j \in \{0, \ldots, |\theta_X| - 1\}$, we denote by $t_j(exp)$ the following term over $\mathscr{X}$ defined inductively over the structure of *exp*: if $exp = z$ then $t_j(exp) = z_j$, if $exp = f(exp_1, \ldots, exp_k)$ then $t_j(exp) = f(t_j(exp_1), \ldots, t_j(exp_k))$, and if $exp = z[\ell|c]$ then $t_j(exp) = z_{j+\ell}$ if $0 \leq j + \ell < |\theta_X|$ and $t_j(exp) = c$ otherwise.

The monitoring algorithm maintains two sets of equations:

- A set R of *resolved equations* $z_i = c$, where $z_i \in \mathscr{X}$ and c is a constant.
- A set U of *unresolved equations* $y_i = t$, where $y_i \in \mathscr{X}_Y$ and t is a non-ground term over $\mathscr{X}$.

Initially both stores are empty. At the $(i+1)$st iteration, the values $\theta_X(x)(i)$ for $x \in X$ become available and the monitor carries out the following steps:

1. The equation $x_i = \theta_X(x)(i)$ is added to R, for each $x \in X$.
2. The equation $y_i = t_i(exp)$ is added to U, for each equation $(y = exp) \in E$.
3. The equations in U are simplified as much as possible, using the following rules:
   - Partial evaluation rules for function applications, such as $0 + a \to a$.
   - If $(y_j = c) \in R$, then every occurrence of y_j in (the terms in) U is substituted by c and possibly simplified further.

   If an equation becomes of the form $y_j = c$, it is removed from U and added to R; furthermore, $\theta_Y(y)(j)$ is set to c.
4. Equations $z_{i-k} = c$ are removed from R, where

$$k := \max(\{0\} \cup \{\ell \mid \ell > 0 \text{ and } z[-\ell|c] \text{ is a subexpression in } E\})$$

position	R	U
0	$x_0 = \mathsf{false}, x'_0 = \mathsf{true}$	$y_0 = x'_0 \vee (x_0 \wedge y_1)$
	$y_0 = \mathsf{true}$	-
1	$x_1 = \mathsf{false}, x'_1 = \mathsf{false}$	$y_1 = x'_1 \vee (x_1 \wedge y_2)$
	$y_1 = \mathsf{false}$	-
2	$x_2 = \mathsf{true}, x'_2 = \mathsf{false}$	$y_2 = x'_2 \vee (x_2 \wedge y_3)$
	-	$y_2 = y_3$
6	$x_6 = \mathsf{true}, x'_6 = \mathsf{false}$	$y_2 = y_3, y_3 = y_4, y_4 = y_5, y_5 = y_6, y_6 = x'_6$
	$y_2 = \mathsf{false}, y_3 = \mathsf{false}, y_4 = \mathsf{false},$ $y_5 = \mathsf{false}, y_6 = \mathsf{false}$	-

Table 3. Sample execution of the monitoring algorithm.

Concerning the last step, we have that, for any position j, the position $j+k$ is the latest future position for which the monitor requires the value of $(\theta_X \cup \theta_Y)(z)(j)$. Thus the equation $z_{i-k} = c$ can be safely removed from R at position i. This is important as it places a bound on the amount of history that needs to be stored. Also, note that the well-formedness condition ensures that each equation in U is eventually resolved.

Example 13. To illustrate the last point, consider the specification $y = y[-3|0] + x$. The value of k for y is 3 and for x is 0. This indicates that for any input stream σ, the equation $x_j = \sigma(j)$ can be removed from R at position j itself. Similarly, $y_j = \tau(j)$ can be removed from R at (or after) position $j+3$, where τ is the output stream.

If in a specification all offsets are negative, that is, the stream expressions only refer to current or previous stream positions, then at the end of each iteration all equations are resolved, i.e. $U = \emptyset$, because all new terms in U can be evaluated and simplified to constants. The specifications from Section 7.1 and Example 12 fall in this category. We therefore illustrate next the algorithm on a specification which contains positive offsets.

Example 14. Consider the specification $y = x' \vee (x \wedge y[1|\mathsf{false}])$ over $\{x, x'\}$ and $\{y\}$, corresponding to the LTL specification $x \mathsf{W} x'$ over finite traces, where W denotes the "weak until" operator. That is, y_j stores the satisfaction of $x \mathsf{W} x'$ on the word encoding the suffixes of the streams x and x' starting at position j. The associated equations are:

$$y_j = \begin{cases} x'_j \vee (x_j \wedge y_{j+1}) & \text{if } j < n-1, \\ x'_j & \text{otherwise (that is, } j = n-1) \end{cases}$$

for $0 \leq j < n$, with n the input streams' length. Let x, x' be the following input streams.

x	false	false	true	true	true	true	true
x'	true	false	false	false	false	false	false

Table 3 lists the contents of the sets R and U at various stream positions j. For each position j there are two rows in the table; the first row lists the contents of R and U after executing steps 1 and 2 of the algorithm, while the second row does the same after executing steps 3 and 4. At position 0, we add $x_0 = x(0)$, i.e. $x_0 = \mathsf{false}$, and $x'_0 = x'(0)$, i.e. $x'_0 = \mathsf{true}$, to R, and $y_0 = x'_0 \vee (x_0 \wedge y_1)$ to U. The equation for y_0 simplifies to

$y_0 = \mathsf{true}$, and is thus moved to R. At position 1, we have $x_1 = \mathsf{false}$ and $x'_1 = \mathsf{false}$ in R and thus we can set $y_1 = \mathsf{false}$, which is also added to R. From $j = 2$ until $j = 5$, we have $x_j = \mathsf{true}$ and $x'_j = \mathsf{false}$. At each of these positions the equations $y_j = y_{j+1}$ are added to U. The set U now contains the equations $y_2 = y_3, y_3 = y_4, \ldots, y_5 = y_6$. At position 6, we have $x_6 = \mathsf{true}$ and $x'_6 = \mathsf{false}$ with the added information that the trace has ended, i.e. $y_6 = x'_6$. Thus we set $y_6 = \mathsf{false}$ and add it to R. This lets us resolve the equations in U and set $y_j = \mathsf{false}$, for all the positions j from 2 to 6.

We end this section by noting that a run of the algorithm on the *UnsafeMapIterator* property can be easily simulated by the reader; see the remark from Example 12.

8 Discussion

In this section we briefly discuss and compare the five different approaches outlined. It is common to compare approaches on *expressiveness* of the specification languages, their *elegance*, and *efficiency* of monitoring algorithms. However, due to lack of available complete results, our comparison is inherently subjective, rather than objective based on concrete data. Hopefully future research and future editions of the runtime verification competition (CRV) [14, 38, 60] will improve on this situation. Some first steps in this direction have been taken in [61]. We also compare the approaches in terms of the type of data structure used to store data values and of the type of produced output.

Expressiveness The FOTL-based approach supports extensions to real-time constraints and aggregation operators, which allow for a wide range of practically relevant properties to be specified in a convenient manner. FOTL specifications are furthermore compatible with an interpretation over infinite traces, which means that specifications can be used for model checking purposes as well as for monitoring purposes. MMT provides a generic framework that allows to use a variety of data algebras with existing monitoring approaches for temporal logics. This genericity allows for a wide range of specifications to be expressed. Furthermore, the MMT approach can monitor any FOTL formula that can be put in prenex normal form—a different kind of restriction than that imposed by the FOTL-based approach. The expressiveness of QEA, LF, and LOLA depend on the their guard, assignment, expression, and function languages. Assuming rich such languages, these systems are Turing complete, and therefore more expressive than temporal logics such as FOTL and TDL. However, it is unclear that full Turing completeness is required for a practically useful RV specification language. The extended state machines of QEA allow fundamentally to write programs, assuming a general guard and assignment language. The rule-based LOGFIRE allows a form of programming where arbitrary data can be passed as arguments to facts, thereby simulating a program state. Added to that is that LOGFIRE is an internal DSL (an API) allowing full fledged programming in SCALA. The LOLA framework phrases runtime verification as a stream processing problem. The framework goes beyond property checking and generally supports computation of any data from traces (quantitative analysis). FOTL, QEA, and LOGFIRE also support such quantitative analysis. Finally, we note that in LOLA much of the performed stream computations are not specified within the approach, but instead through interpreted functions.

Elegance First-order temporal logics such as FOTL and MMT are quite standard, and allow for very elegant specifications, compared to the other approaches described here. Although this should not be a surprise, it is quite a commonly stated opinion (folklore) that temporal logics are hard to use by practitioners. However, as discussed in [48], we believe that in many cases a temporal logic specification is the most convenient form of formalisation, and that temporal logics certainly deserve to be taken seriously for runtime verification purposes, preferably augmented with other constructs such as sequencing, scopes, etc. Specifications in the other formalisms are less elegant in the average case due to the fact that they operate at a lower level of operational abstraction. For example, in QEA states in state machines have to explicitly named. This issue could be alleviated by extending the approach to multiple plugins, as is done in the JAVAMOP work which QEA builds on. Similarly, in LOGFIRE, intermediate facts have to be named, created, and deleted. One way to look at QEA, LOGFIRE, and LOLA, is as low-level formalisms to be targets of translations from higher-level logics. For instance, a translation from FOTL's monitorable fragment to LOLA seems straightforward.

Efficiency The analysis of the complexity of monitoring algorithms for specifications with data has not received much attention so far. Of the presented approaches, we know that the FOTL-based algorithm has polynomial time and space complexity in the number of data values in the trace (see [17] for details), while LOLA's algorithm uses time and space that is linear in the length of the trace under the assumption that interpreted functions execute in linear time [29].[18] We note that under the anticipation requirement [55] (which asks that a verdict should be output as soon as every extension of the current trace leads to the same verdict), the monitoring problem often becomes a hard one because it requires to solve the satisfiability problem for the considered specification language, which is usually a hard problem for expressive languages (e.g., for FOTL it is undecidable). Anticipation is partially supported among the systems presented here by MMT and QEA. Finally, part of the reason for the scarcity of worst-case complexity analyses is that such results often offer little insight into the efficiency of the tools implementing the monitoring algorithms, an aspect that we consider next.

Each of the monitoring approaches presented here has been implemented. There is not enough data to make thorough comparisons between the performance of these tools. However, based on the results of the runtime verification competitions, and experimental evaluation sections in various papers, we can still formulate some observations. The most efficient tools so far explored in the literature appear to be those based on the slicing approach, which was introduced in systems such as TRACEMATCHES and MOP, and carried further in MARQ [59]. The key advantage of parametric trace slicing is that it admits efficient indexing approaches that have a significant impact on monitoring overhead. The generic nature of MMT, which allows to combine any data algebra with temporal logic, and the use of an SMT solver to check, for each incoming event, the generated constraints, makes performance an issue. Performance can be improved significantly by using a dedicated decision procedure instead of a generic solver. Furthermore, on a particular class of properties, namely LTL over tree-ordered ids (and a particular theory, namely that of equality), the MMT algorithm lent itself to a highly effective optimisation [30], implemented in the MUFIN tool. LOGFIRE's implementa-

[18] This assumption is not satisfied for our formalisations of the *UnsafeMapIterator* property.

tion, which uses the RETE algorithm, is rather complex, and does not seem to yield the same efficient solution for runtime verification as trace slicing. As documented in [47], however, LOGFIRE performs well compared to other rule systems.

Data Structures We focus here on the general nature of the data structures used for representing the observed history in the trace at any point during monitoring. With this perspective three different approaches emerge. Both the FOTL and the LOGFIRE algorithms store observed data explicitly as data records, that is, tuples of data values. The FOTL-based algorithm operates with relations (sets of such tuples), which can also be seen as database tables, while LOGFIRE operates with individual tuples, stored as facts in a network. The QEA-based algorithm stores a mapping from valuations to automata states. Valuations can also be seen as data records. Valuations are indexed such that the relevant ones can be found efficiently. The MMT and LOLA systems approach the problem differently by storing constraints between variables and data values; their denotation are the data records stored in the other approaches. As such the five different algorithms use three main approaches to storage of data during monitoring: data record collections, indexed mappings, and constraints.

Monitor Output The systems presented yield different forms of output. FOTL outputs sets of tuples, representing violations. MMT yields a verdict from an arbitrary truth-value lattice, which can include values like "unknown". QEA yields a verdict from a 4-valued Boolean logic.[19] LOGFIRE by default only outputs a result (an error trace) if the specification is violated, but for each event it offers access to the set of all facts generated so far. LOLA produces a data value at each step during monitoring as part of an output stream. QEA, LOGFIRE, and to some extent FOTL, also can produce any form of output from a trace, although these systems were not created for this purpose.

9 Related Work

In the following we discuss some related work, grouping it by the five presented approaches. As with the FOTL and MMT approaches, a number of other runtime verification approaches also use formalisms based on extensions of (linear) temporal logics with variables modeling event parameters. All these (linear) temporal logic extensions thus exhibit variable quantification, either implicitly or explicitly. In most extensions [44, 64, 63, 18, 12, 46, 19], the domain of a quantifier is restricted to the data appearing at the current position in the trace. When a single event can occur at a position in the trace, as in this chapter and in [44, 12], the domain thus consists of at most one value and quantification has the flavor of the so-called freeze quantification [7]. In all these works, quantification is handled algorithmically by encoding (at runtime) quantifiers with a finite number of conjunctions ($\forall$) and disjunctions ($\exists$), one for each variable instantiation encountered during runtime. The monitoring algorithms are either based on a translation from the underlying propositional formulas to automata, as in [63, 19], or on a syntax-oriented tableaux-like procedure, as in [18, 12, 46]. In contrast to the above, in the FOTL and MMT approaches quantification is over the whole data domain as it is in classic first-order logic. This is also the case with [27], which presents a similar monitoring algorithm to that in [17].

[19] Assuming a guard and assignment language such that checking QEA emptiness is decidable.

The presented FOTL monitoring approach shares similarities with algorithms for checking temporal integrity constraints of databases and for specifying temporal database triggers, and in particular with [26], which the approach in [17] extends. The MMT framework takes an indirect approach for monitoring first-order temporal logics, by providing a way to lift propositional monitors to the setting of data values. Thus, in its aim to achieve a temporal logic independent solution, the MMT approach presents similarities with the MOP framework [25, 56].

Trace slicing was introduced in [5] with a suffix-matching semantics and then extended to total-matching (which is non-trivial) in the JAVAMOP work [56]. The latter work also introduces different notions of *matching* which are difficult to capture in the quantification framework introduced earlier, for example, on non-total and connected[20] bindings. The JAVAMOP language, however, introduces an unnecessary restriction on expressiveness by enforcing a unique mapping from event names to parameter names, disallowing an event name to be used with different parameters in a specification. This prevents, for example, a property like *"a lock acquired by a thread t cannot be acquired by another thread t' until first released by t"* as here the lock action refers to two different variables. On the other hand, JAVAMOP supports infinite-state specifications as context-free grammars (CFGs), which are not supported directly by any of the other formalisms. CFG properties can be only expressed indirectly, and much less elegantly, in other formalisms (including QEA and LOGFIRE) by simulating push-down automata. The work of [8] extends the parametric trace slicing approach with constraints, similar to the combination of free variables and guards in QEA. The work in [30] introduces more efficient monitoring algorithm by restricting specifications to those with hierarchical relationships between quantified variables. Other parametric monitoring approaches, that like QEA are automata-based, include LARVA [28] and ORCHIDS [43].

The RETE-based LOGFIRE is inspired by the RULER rule system [9, 13] (not based on RETE), which again was influenced by the EAGLE system [11] (a linear μ-calculus supporting parametric monitoring with past time, future time, and sequencing operators). Several RETE-based external rule DSLs exist, such as DROOLS [2] and CLIPS [1]. HAMMURABI [41] (actor-based) and ROOSCALOO [3] (RETE based) are two other internal SCALA rule DSLs. Unlike LOGFIRE, none of these rule systems treat events specially. A RETE-based system for aspect-oriented programming with history pointcuts is described in [49].

The stream-based approach of LOLA resembles synchronous programming languages such as Lustre [45] and Esterel [23]. The approach was extended in LOLA 2.0 [39] with two new language features, namely template stream expressions and dynamic stream generation, which support a notion of slicing similar to that found in QEA.

10 Conclusion

We have described five different formalisms for parameterised runtime verification. The field of runtime verification is still young and there is no clear agreement on what constitutes a good specification formalism. This is in contrast to the field of e.g. model

[20] Bindings whose values are explicitly connected by events in the trace.

checking, where LTL and CTL have become de facto standards. Part of the reason is possibly data parameterisation, which opens up new doors as to what a specification language can look like, as this chapter illustrates.

References

1. Clips website. http://clipsrules.sourceforge.net.
2. Drools website. http://www.jboss.org/drools.
3. Rooscaloo website. https://github.com/daveray/rooscaloo.
4. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases: The Logical Level. Addison Wesley (1994)
5. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to AspectJ. SIGPLAN Not. 40, 345–364 (October 2005)
6. Alur, R., Etessami, K., Madhusudan, P.: A temporal logic of nested calls and returns. In: TACAS. LNCS, vol. 2988, pp. 467–481. Springer (2004)
7. Alur, R., Henzinger, T.A.: A really temporal logic. J. ACM 41(1), 181–204 (1994)
8. Ballarin, C.: Two generalisations of Roşu and Chen's trace slicing algorithm A. In: Proc. of the 5th Int. Conf. on Runtime Verification (RV'14). pp. 15–30. Springer (2014)
9. Barringer, H., Rydeheard, D., Havelund, K.: Rule systems for run-time monitoring: from Eagle to RuleR. In: Proc. of the 7th Int. Workshop on Runtime Verification (RV'07). LNCS, vol. 4839, pp. 111–125. Springer (2007)
10. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.: Quantified event automata: Towards expressive and efficient runtime monitors. In: Proc. of the 18th Int. Symposium on Formal Methods (FM'12). pp. 68–84. Springer (2012)
11. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-based runtime verification. In: VMCAI. LNCS, vol. 2937, pp. 44–57. Springer (2004)
12. Barringer, H., Havelund, K.: TraceContract: A Scala DSL for trace analysis. In: Proc. of the 17th Int. Symposium on Formal Methods (FM'11). LNCS, vol. 6664, pp. 57–72 (2011)
13. Barringer, H., Rydeheard, D.E., Havelund, K.: Rule systems for run-time monitoring: From Eagle to RuleR. Journal of Logic and Computation 20(3), 675–706 (2010)
14. Bartocci, E., Falcone, Y., Bonakdarpour, B., Colombo, C., Decker, N., Havelund, K., Joshi, Y., Klaedtke, F., Milewicz, R., Reger, G., Rosu, G., Signoles, J., Thoma, D., Zalinescu, E., Zhang, Y.: First international competition on runtime verification: rules, benchmarks, tools, and final results of crv 2014. Int. J. Softw. Tools Technol. Trans. pp. 1–40 (2017)
15. Basin, D., Harvan, M., Klaedtke, F., Zălinescu, E.: MONPOLY: Monitoring usage-control policies. In: Proc. of the 2nd Int. Conf. on Runtime Verification (RV'11). LNCS, vol. 7186, pp. 360–364. Springer (2012)
16. Basin, D.A., Klaedtke, F., Marinovic, S., Zălinescu, E.: Monitoring of temporal first-order properties with aggregations. Form. Method. Syst. Des. 46(3), 262–285 (2015)
17. Basin, D.A., Klaedtke, F., Müller, S., Zălinescu, E.: Monitoring metric first-order temporal properties. J. ACM 62(2), 15 (2015)
18. Bauer, A., Goré, R., Tiu, A.: A first-order policy language for history-based transaction monitoring. In: Proc. of the 6th Int. Colloquium on Theoretical Aspects of Computing (ICTAC). LNCS, vol. 5684, pp. 96–111. Springer (2009)
19. Bauer, A., Küster, J., Vegliach, G.: The ins and outs of first-order runtime verification. Form. Method. Syst. Des. 46(3), 286–316 (2015)
20. Bauer, A., Leucker, M., Schallhart, C.: Monitoring of real-time properties. In: FSTTCS. LNCS, Springer (2006)

21. Bauer, A., Leucker, M., Schallhart, C.: The good, the bad, and the ugly, but how ugly is ugly? In: RV. LNCS, Springer (2007)
22. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. (2011)
23. Berry, G.: Proof, language, and interaction. chap. The Foundations of Esterel, pp. 425–454. MIT Press (2000)
24. Bozzelli, L., Sánchez, C.: Foundations of boolean stream runtime verification. Theoret. Comput. Sci. 631, 118–138 (2016)
25. Chen, F., Roşu, G.: Parametric trace slicing and monitoring. In: Proc. of the 15th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09). pp. 246–261. Springer (2009)
26. Chomicki, J.: Efficient checking of temporal integrity constraints using bounded history encoding. ACM Trans. Database Syst. 20(2), 149–186 (1995)
27. Chowdhury, O., Jia, L., Garg, D., Datta, A.: Temporal mode-checking for runtime monitoring of privacy policies. In: Proc. of the 26th Int. Conf. on Computer Aided Verification (CAV'14). LNCS, vol. 8559, pp. 131–149. Springer (2014)
28. Colombo, C., Pace, G.J., Schneider, G.: LARVA — safer monitoring of real-time Java programs (tool paper). In: Proc. of the 7th IEEE Int. Conf. on Software Engineering and Formal Methods. pp. 33–37. SEFM '09, IEEE Computer Society (2009)
29. D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: LOLA: Runtime monitoring of synchronous systems. In: Proc. of the 12th Int. Symposium on Temporal Representation and Reasoning. pp. 166–174. IEEE Computer Society (2005)
30. Decker, N., Harder, J., Scheffel, T., Schmitz, M., Thoma, D.: Runtime monitoring with union-find structures. In: Proc. of the 22nd Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 9636, pp. 868–884. Springer (2016)
31. Decker, N., Leucker, M., Thoma, D.: Impartiality and anticipation for monitoring of visibly context-free properties. In: RV. LNCS, Springer (2013)
32. Decker, N., Leucker, M., Thoma, D.: jUnitRV–Adding Runtime Verification to jUnit, pp. 459–464. Springer (2013)
33. Decker, N., Leucker, M., Thoma, D.: Monitoring modulo theories. In: Proc. of the 20th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'14). pp. 341–356. Springer (2014)
34. Decker, N., Leucker, M., Thoma, D.: Monitoring modulo theories. Int. J. Softw. Tools Technol. Trans. 18(2), 205–225 (2016)
35. Dong, W., Leucker, M., Schallhart, C.: Impartial anticipation in runtime-verification. In: Proc. of 6th Int. Symposium on Automated Technology for Verification and Analysis (ATVA 2008). pp. 386–396. Springer (2008)
36. Doorenbos, R.B.: Production Matching for Large Learning Systems. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA (1995)
37. Eisner, C., Fisman, D., Havlicek, J., Lustig, Y., McIsaac, A., Campenhout, D.V.: Reasoning with temporal logic on truncated paths. In: Proc. of the 15th Int. Conf. on Computer Aided Verification (CAV 2003). LNCS, vol. 2725, pp. 27–39. Springer (2003)
38. Falcone, Y., Ničković, D., Reger, G., Thoma, D.: Second international competition on runtime verification: CRV 2015. In: Proc. of the 15th Int. Conf. on Runtime Verification (RV'15). LNCS, vol. 9333, pp. 405–422. Springer (2015)
39. Faymonville, P., Finkbeiner, B., Schirmer, S., Torfah, H.: A Stream-Based Specification Language for Network Monitoring, pp. 152–168. Springer (2016)
40. Forgy, C.: Rete: A fast algorithm for the many pattern/many object pattern match problem. Artificial Intelligence 19, 17–37 (1982)

41. Fusco, M.: Hammurabi - a Scala rule engine. In: Scala Days 2011, Stanford University, California (2011)
42. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database systems: The complete book. Pearson Education (2009)
43. Goubault-Larrecq, J., Olivain, J.: A smell of ORCHIDS. In: Proc. of the 8th Int. Workshop on Runtime Verification (RV'08). LNCS, vol. 5289, pp. 1–20. Springer (2008)
44. Håkansson, J., Jonsson, B., Lundqvist, O.: Generating online test oracles from temporal logic specifications. Int. J. Softw. Tools Technol. Trans. 4(4), 456–471 (2003)
45. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous dataflow programming language lustre. Proc. of the IEEE 79(9), 1305–1320 (September 1991)
46. Hallé, S., Villemaire, R.: Runtime enforcement of web service message contracts with data. IEEE Trans. Services Computing 5(2), 192–206 (2012)
47. Havelund, K.: Rule-based runtime verification revisited. Int. J. Softw. Tools Technol. Trans. 17(2), 143–170 (2015)
48. Havelund, K., Reger, G.: Runtime verification logics - a language design perspective. In: KIMfest - a conference in honour of Kim G. Larsen on the occasion of his 60th birthday, Aalborg University, 19-20 August 2017. LNCS, Springer (2017)
49. Herzeel, C., Gybels, K., Costanza, P.: Escaping with future variables in HALO. In: Proc. of the 7th Int. Workshop on Runtime Verification (RV'07). LNCS, vol. 4839, pp. 51–62. Springer (2007)
50. Hodkinson, I.M., Wolter, F., Zakharyaschev, M.: Decidable fragment of first-order temporal logics. Annals of Pure and Applied Logic 106(1-3), 85–134 (2000)
51. Holzmann, G.J.: The Spin Model Checker – Primer and Reference Manual. Addison-Wesley (2004)
52. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to automata theory, languages, and computation
53. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Syst. 2(4), 255–299 (1990)
54. Leucker, M., Sánchez, C.: Regular linear temporal logic. In: ICTAC. LNCS, Springer (2007)
55. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebr. Program. 78(5), 293–303 (2009)
56. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Roşu, G.: An overview of the MOP runtime verification framework. Int. J. Softw. Tools Technol. Trans. 14(3), 249–289 (2012)
57. Purandare, R., Dwyer, M.B., Elbaum, S.: Monitoring Finite State Properties: Algorithmic Approaches and Their Relative Strengths, pp. 381–395. Springer (2012)
58. Reger, G.: Automata Based Monitoring and Mining of Execution Traces. Ph.D. thesis, University of Manchester (2014)
59. Reger, G., Cruz, H.C., Rydeheard, D.: MarQ: Monitoring at runtime with QEA. In: Proc. of the 21st Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15). pp. 596–610. Springer (2015)
60. Reger, G., Hallé, S., Falcone, Y.: Third international competition on runtime verification: CRV 2016. In: Proc. of the 16th Int. Conf. on Runtime Verification (RV'16). LNCS, vol. 10012, pp. 21–40. Springer (2016)
61. Reger, G., Rydeheard, D.: From first-order temporal logic to parametric trace slicing. In: Proc. of 6th Int. Conf. on Runtime Verification (RV'15). pp. 216–232. Springer (2015)
62. Roşu, G., Chen, F.: Semantics and algorithms for parametric monitoring. Logical Methods in Computer Science 8(1) (2012)
63. Stolz, V.: Temporal assertions with parameterized propositions. J. Logic Comput. 20(3), 743–757 (2010)
64. Stolz, V., Bodden, E.: Temporal assertions using AspectJ. In: Proc. of the 5th Int. Workshop on Runtime Verification (RV'05). ENTCS, vol. 144(4), pp. 109–124. Elsevier (2006)