Runtime Monitoring with Union-Find Structures

Normann Decker, Jannis Harder, Torben Scheffel, Malte Schmitz(✉), and Daniel Thoma

Institute for Software Engineering and Programming Languages, University of Lübeck, Lübeck, Germany
{decker,harder,scheffel,schmitz,thoma}@isp.uni-luebeck.de

Abstract. This paper is concerned with runtime verification of object-oriented software system. We propose a novel algorithm for monitoring the individual behaviour and interaction of an unbounded number of runtime objects. This allows for evaluating complex correctness properties that take runtime data in terms of object identities into account. In particular, the underlying formal model can express hierarchical interdependencies of individual objects. Currently, the most efficient monitoring approaches for such properties are based on lookup tables. In contrast, the proposed algorithm uses union-find data structures to manage individual instances and thereby accomplishes a significant performance improvement. The time complexity bounds of the very efficient operations on union-find structures transfer to our monitoring algorithm: the execution time of a single monitoring step is guaranteed logarithmic in the number of observed objects. The amortised time is bound by an inverse of Ackermann's function. We have implemented the algorithm in our monitoring tool Mufin. Benchmarks show that the targeted class of properties can be monitored extremely efficient and runtime overhead is reduced substantially compared to other tools.

1 Introduction

In practice, exhaustive verification of a system is often not an option because of economical or practical reasons, when third-party libraries are used or code is loaded dynamically at runtime from uncontrolled sources. In these cases, *Runtime Verification (RV)* can provide a reasonable lightweight alternative. Instead of analysing the whole behaviour of a system, RV focuses on techniques to observe a program's execution and evaluate correctness properties regarding this specific run. They allow for balancing the verification effort regarding the targeted correctness guarantees. For example, verification efforts can focus on specific, feasible parts such as low-level primitives or protocol implementations while the remaining parts are being monitored at runtime. Moreover, RV can be applied during software testing and debugging to obtain concise and specific information.

Work partially supported by the European Cooperation in Science and Technology (COST Action ARVI) and the German Federal Ministry for Education and Research (CONIRAS/01IS13029).

M. Chechik and J.-F. Raskin (Eds.): TACAS 2016, LNCS 9636, pp. 868–884, 2016.
DOI: 10.1007/978-3-662-49674-9_54

In software systems, a monitoring process is typically executed in parallel to a program under scrutiny. While this can provide a very detailed observation of the system's behaviour, it necessarily imposes runtime overhead for the whole system in terms of memory and computing resources. It is one of the main concerns in RV to keep this overhead as small as possible. This is particularly challenging for object-oriented systems. They require to track an unbounded number of runtime objects and evaluate their individual behaviour and interaction. Consider, for example, a Java collection object and iterator objects created for it. The number of iterators can become arbitrarily large. Once the collection is modified none of them is supposed to be used again, while iterators created for a different collection or after the modification have still a valid state. Thus, for each object some information, e.g. whether it is still valid to be used, may have to be stored and updated upon some program event.

The currently most efficient tools for monitoring object-oriented systems are JavaMOP [17] and MarQ [18]. They use data structures based on *lookup tables*, implemented as hash maps, to store this mapping of objects to their individual state. Unfortunately, this approach can quickly become infeasible since the number of table entries increases linearly with the number of maintained objects. A program event may affect all of them and thus require an update of the corresponding entries. Hence the cost of a *single* monitoring step can increase linearly with the length of the observed execution trace. Considering the example above, using many iterators quickly increases the lookup table. Every modification of the collection requires iterating through the table to update the entries of all derived iterator objects.

Contribution. We address this problem and propose a novel monitoring algorithm that uses *union-find data structures* to store the state of program objects. The essential idea is to store a mapping $c : \Delta \to Q$ from object (identifiers) Δ to monitoring information (states) Q in terms of *sets* $\Delta_q \subseteq \Delta$ of objects for each state $q \in Q$. Then, changing the state of all objects in some state q to some state q' can be done by merging Δ_q into $\Delta_{q'}$. On union-find structures this is a constant-time operation, independent of the size of the sets. Further, our data structure allows for selecting and updating more specific subsets of program objects. The user can provide a tree-like hierarchy for the program objects and refer to it in the specification. For example, every iterator object can be filed as a direct child of its corresponding collection. The data structure then provides efficient access to the set of, e.g., all children or ancestors of a particular object. Hence, upon the modification of a collection, all corresponding valid iterator objects can be marked invalid at once. Tree-like object relations are ubiquitous in programming and employed in many algorithms, data structures and architectures. For correctness properties expressed with respect to such a hierarchy, our algorithm provides extremely efficient runtime evaluation.

Outline. In the following Sect. 2 we define an operational model that allows for expressing the behaviour and hierarchical dependencies between individual objects. This model provides the conceptual basis for our monitoring approach and thus characterises formally the addressed type of correctness properties.

To provide a better understanding of the properties, we also identify a corresponding fragment of first-order temporal logic. Based on the operational model, we describe our data structure in Sect. 3. Our algorithm for efficiently processing runtime events and updating the data structure is presented in Sect. 4. We discuss the performance of our approach first by providing bounds for the time complexity of a monitoring step. Then, Sect. 5 is concerned with our implementation. We present benchmarks for a collection of properties providing evidence that our approach performs well in practice and in particular in comparison with the state-of-the-art tools JavaMOP and MarQ.

Related Work. A monitoring approach for object-oriented systems, where the instrumentation framework AspectJ is extended by a simple expression language, was already considered in [1]. It allows for matching observed events against patterns with free variables that are bound to values provided by the observation. Data in general, of which object IDs form a special case, was intensively studied for runtime verification leading to various approaches based on different specification formalisms and execution schemes [4–7,12,13,15,19,20]. Regarding efficient monitoring for object-oriented systems the influencial work by Chen and Rosu [19] on the *parametric trace slicing* technique is tailored specifically towards handling events carrying data in terms of object identifiers. It is implemented in the system JavaMOP [17] which is considered one of the best performing runtime verification tools. The trace slicing approach has been generalised to the concept of *quantified event automata (QEA)* [4] in order to increase expressiveness while still allowing for efficient evaluation. The tool MarQ [18] is based on QEA and can compete performance-wise even with JavaMOP. The essential idea of these frameworks is to evaluate a symbolic property on a set of projections of an input trace. Trace slicing specifically considers sequences of events which are parameterised by identifiers. A sequence is divided into sub-sequences, called slices, where all positions share common parameter values. The slices are then monitored independently. In contrast to our approach, only limited interdependencies between the different slices can be checked.

2 Projection Automata

The essential characteristics of an object are its *state* and *identity*. We therefore use a model that reflects both but provides a reasonable abstraction. Finite word automata are an established concept that is well suited for runtime verification because it naturally operates on sequences of inputs. Regarding identity, we employ the framework of *data words* to model observations that relate to a particular object. In this setting, an object is reduced to its mere identity and represented in terms of a so-called *data value*. Formally, we consider an infinite set Δ of such values in order to represent an arbitrary number of different objects. A finite set Σ of *symbols* represents the type of observations, e.g., a call to a particular method or the access to a variable. A data word is now a finite sequence $w = (a_1, d_1)(a_2, d_2)\ldots(a_n, d_n) \in (\Sigma \times \Delta)^*$ of letters consisting of a symbol $a \in \Sigma$ and a value $d \in \Delta$.

For representing the hierarchical relation between objects we impose additional structure on Δ in terms of a *tree-ordering* relation $\leq$. It models the relation between all possibly occurring objects as a forest. A tree-ordering is a partial ordering where every strictly descending chain $d_1 > d_2 > \dots$ is finite and such that for every non-minimal element $d \in \Delta$ the largest element $d' < d$ is unique. We call d' the *parent* of d, written $\mathsf{par}(d)$. The *level* of a value $d \in \Delta$ is defined as $\mathsf{lvl}(d) = 1$ if d is minimal and otherwise $\mathsf{lvl}(d) = \mathsf{lvl}(\mathsf{par}(d)) + 1$. We call $(\Delta, \leq)$ of *depth* ℓ if there are longest strictly descending chains of length ℓ. Additionally, we assume that $(\Delta, \leq)$ contains infinitely many minimal elements and that every non-minimal element $d \in \Delta$ has an infinite number of siblings $d \neq d' \in \Delta$ with $\mathsf{par}(d') = \mathsf{par}(d)$.

Definition 1 (Projection Automata). *A* projection automaton (PA) *is a tuple* $\mathcal{A} = (Q, \Sigma, \delta, q_0, \lambda)$ *where* Q *is a finite set of* states, Σ *is a finite* alphabet, $\delta : Q \times \Sigma \times \{<, =, >, \|\} \to Q$ *is the* transition function, $q_0 \in Q$ *is the* initial state *and* $\lambda : Q \to \mathbb{S}$ *is the* output labelling *for some semi-lattice* $(\mathbb{S}, \sqcap)$.

The operational semantics of PA is given in terms of configurations $c : \Delta \to Q$ *that map data values to states. The* run *of* $\mathcal{A}$ *on a data word* $w = (a_1, d_1) \dots (a_n, d_n)$ *is a sequence of configurations* $\rho_w = c_0 \dots c_n$ *such that the initial configuration is the constant function* $c_0 : \Delta \to \{q_0\}$ *and for all positions* $0 \leq i < n$ *and all data values* $d \in \Delta$ *we have* $c_{i+1}(d) = \delta(c_i(d), (a_{i+1}, \bowtie))$ *where* $\bowtie \in \{<, =, >, \|\}$ *and* $d_{i+1} \bowtie d$. *The* output *of* $\mathcal{A}$ *for the data word* w *is* $\mathcal{A}(w) := \bigsqcap_{d \in \Delta} \lambda(c_n(d))$.

Syntactically, a PA is a finite automaton with output (i.e., a Moore machine) over the input alphabet $\Sigma \times \{<, =, >, \|\}$ and the output alphabet $\mathbb{S}$. Intuitively, to every data value $d \in \Delta$, an instance of the automaton is associated that reads, instead of an input letter $(a, d') \in \Sigma \times \Delta$, the symbol $a \in \Sigma$ and the information how the observed value d' relates to itself, in terms of one of the symbols from $\{<, =, >, \|\}$. The output of all instances is then aggregated to a single verdict, hence the semi-lattice. Note that the restriction to a deterministic transition function is not essential since non-determinism (even alternation) can be eliminated by standard constructions.

Example. Recall the property that modifying a collection invalidates iterators previously created for it. The data values Δ can model these two types of objects by choosing an ordering $\leq$ with two levels: collection IDs are minimal (roots) and the iterator IDs $d_I \in \Delta$ created for a collection with ID $d_C \in \Delta$ are direct children of $d_C < d_I$. Given this structure on Δ, the PA in Fig. 1 (Iterator) expresses the property. Initially, all objects remain in state q_0. Upon the creation (c) of an iterator with ID $d_I \in \Delta$, this new iterator receives the letter $(\mathsf{c}, =)$ and changes its state to q_1. The corresponding collection receives $(\mathsf{c}, >)$ and all others receive $(\mathsf{c}, \|)$, thus staying in q_0. Upon the modification of some collection (m), all iterators for it receive $(\mathsf{m}, <)$ (the observed ID is strictly smaller) and if they happen to be in state q_1 move to state q_2. Finally, when `next()` is called on some iterator, this one reads the letter $(\mathsf{n}, =)$ and only if it happens to be in state q_2 it moves to the failure state. Figure 1 shows further examples to be discussed in Sect. 5.

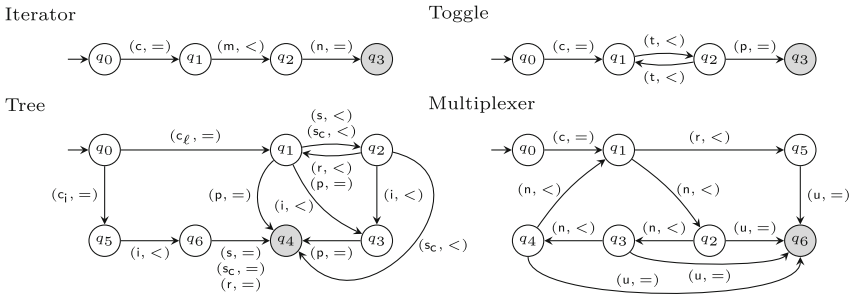

Fig. 1. Example properties formulated as PA with outputs $\top$ (white states) and $\bot$ (grey states). Missing edges are self-loops.

Projection automata are closely related to *class automata* [11] that feature an additional transducer but use only equality on the data domain. It can easily be shown that PA (like class automata) can simulate Minsky machines.

A Logical Perspective. Projection automata characterise precisely the properties that our monitoring algorithm can verify since it is based on their operational semantics. On the other hand, first-order extensions of temporal logics, in particular linear-time temporal logic (LTL), received much attention in RV [8–10,13] because they provide a very generic framework for specifying properties in a declarative fashion. In the following, we therefore discuss briefly how PA relate to temporal logic with first-order constraints. We identify a fragment of first-order logic that can be translated to PA and thus allows for using the very efficient algorithm presented in Sects. 3 and 4 instead of generic techniques.

The fragment consists of a logical language that uses a single variable x and a single constant d as well as zero-ary predicates (propositions) P_a, for $a \in \Sigma$, and a binary predicate $\leq$. Formulae of that language have the form $\forall_x \varphi$ where φ is defined by the grammar $\varphi ::= P_a \mid \varphi \wedge \varphi \mid \neg\varphi \mid \mathrm{X}\,\varphi \mid \varphi\,\mathrm{U}\,\varphi \mid t \leq t$ where $a \in \Sigma$ and $t \in \{x, \mathsf{d}\}$ is either the variable or the constant.

Each letter $(a, d) \in \Sigma \times \Delta$ in a data word can be considered as a structure s over the signature above with universe Δ. Such a structure s interprets the constant d as the value $d \in \Delta$, the proposition P_a as true, the propositions P_b, for $b \neq a$, as false and the binary predicate $\leq$ as the tree-order relation on Δ. For simplicity, however, let us define the semantics directly over data words as follows. The semantics of the terms d and x is given for an interpretation $d \in \Delta$ and a valuation $d_x \in \Delta$ as $[\![\mathsf{d}]\!](d, d_x) = d$ and $[\![x]\!](d, d_x) = d_x$. For data words $w \in (\Sigma \times \Delta)^*$, letters $(a, d) \in \Sigma \times \Delta$ and values d_x we let

$$
\begin{aligned}
(w, d_x) &\models \forall_x \varphi && \text{iff } (w, d'_x) \models \varphi \text{ for all } d'_x \in \Delta \\
((a, d)w, d_x) &\models P_a \\
((a, d)w, d_x) &\models t_1 \leq t_2 && \text{iff } [\![t_1]\!](d, d_x) \leq [\![t_2]\!](d, d_x) \\
((a, d)w, d_x) &\models \mathrm{X}\,\varphi && \text{iff } (w, d_x) \models \varphi \\
(w, d_x) &\models \varphi_1\,\mathrm{U}\,\varphi_2 && \text{iff } (w, d_x) \models \varphi_2 \vee (\varphi_1 \wedge \mathrm{X}(\varphi_1\,\mathrm{U}\,\varphi_2))
\end{aligned}
$$

The semantics of Boolean operators is defined as usual. To stay close to PA we include the empty word ϵ, e.g., $(\epsilon, d_x) \not\models P_a$ and $(\epsilon, d_x) \models \varphi_1 \,\mathrm{U}(\neg P_a)$.

From formulae φ as defined in Eq. 2 we can now construct a PA $\mathcal{A}_\varphi = (Q, \Sigma, \delta, q_0, \lambda)$ with outputs from the Boolean lattice $\mathbb{B} = \{\bot, \top\}$ such that $\mathcal{A}_\varphi(w) = \top$ if and only if $(w, d_x) \models \forall_x \varphi$ for some (hence every) $d_x \in \Delta$. Interpreting subformulae of the form P_a and $t_1 \leq t_2$ as *atomic* propositions we can apply standard automata construction techniques (see, e.g., [21]) and obtain a finite automaton $\mathcal{B}$ over the alphabet $\Gamma = 2^{AP}$ for $AP = \{P_a, t_1 \leq t_2 \mid a \in \Sigma, t_1, t_2 \in \{x, \mathsf{d}\}\}$. Due to the subset construction, the automaton $\mathcal{B}$ reads letters that cannot occur in our setting. For example, there is no letter $(a, d) \in \Sigma \times \Delta$ that induces a structure where P_a and P_b holds for $a \neq b$ or where $t \leq t$ does not hold for $t \in \{x, \mathsf{d}\}$. We remove these letters and corresponding edges in $\mathcal{B}$, keeping thus only letters of the form $g^a_M = \{P_a, x \leq x, \mathsf{d} \leq \mathsf{d}\} \cup M \in \Gamma$ where $M \subseteq \{x \leq \mathsf{d}, \mathsf{d} \leq x\}$ and $a \in \Sigma$. These have a unique correspondence to the symbols from $\Sigma \times \{<, =, >, \|\}$ and we thus obtain $\mathcal{A}_\varphi$ by renaming each such g^a_M to $(a, =)$ if $M = \{x \leq \mathsf{d}, \mathsf{d} \leq x\}$, to $(a, <)$ if $M = \{\mathsf{d} \leq x\}$, to $(a, >)$ if $M = \{x \leq \mathsf{d}\}$ and to $(a, \|)$ if $M = \emptyset$.

Note that this is essentially the generic construction presented in [13] instantiated for the temporal logic LTL defined accordingly and the theory of letters from $(\Sigma \times \Delta)$. Technically, removing edges with inconsistent labels can be considered as an optimisation step that is possible given the simple structure of the letters in a data word. We use LTL here due to its popularity in RV but can replace it by other logics that translate to finite automata.

3 Data Structure

Our monitoring algorithm is based on simulating the operational semantics of a given PA $\mathcal{A} = (Q, \Sigma, \delta, q_0, \lambda)$. It therefore operates on a data structure to represent configurations c of $\mathcal{A}$ that we describe in this section. The essential idea underlying our data structure is to store such a mapping $c : \Delta \to Q$ by *partitioning* Δ into subsets of data values with the same state assigned. At the same time, this partition should also reflect the ordering relation between values. Then, updating a configuration amounts only to a few operations on subsets of Δ and we organise our data structure such that these can be performed efficiently.

When processing a letter $(a, d) \in \Sigma \times \Delta$ the successive configuration c' maps every value $e \in \Delta$ to a state $\delta(c(e), (a, \bowtie))$, i.e., depending on the previous state $c(e)$ and the relation $\bowtie$ between d and e. Our data structure therefore provides efficient access to the subsets $\Delta_q = \{d \in \Delta \mid c(d) = q\}$, $\Delta_{d\bowtie} = \{e \in \Delta \mid d \bowtie e\}$ and $\Delta_{d\bowtie,q} = \Delta_q \cap \Delta_{d\bowtie}$ for $\bowtie \in \{<, =, >, \|\}$. Then, $(\Delta_{d\bowtie,q})_{q\in Q, \bowtie\in\{<,=,>,\|\}}$ is a partition of Δ that reflects the ordering and represents the mapping c. It allows for characterising the partition $(\Delta'_{d\bowtie,q})_{q\in Q, \bowtie\in\{<,=,>,\|\}}$ representing c' by

$$\Delta'_{d\bowtie,q} = \bigcup_{q' \mid q = \delta(q',(a,\bowtie))} \Delta_{d\bowtie,q'}.$$

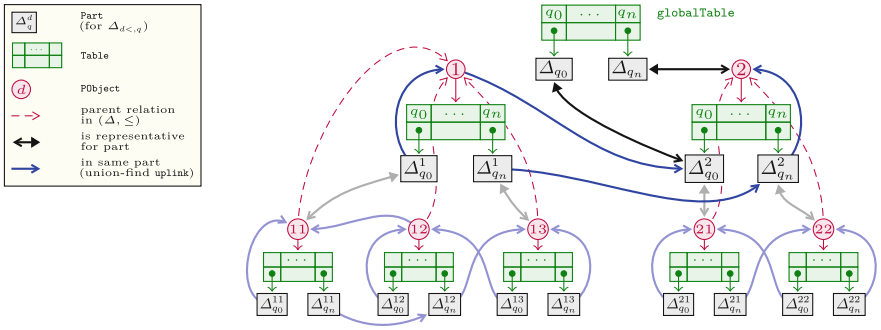

Fig. 2. Example for the shape of the data structure to represent PA configurations. `Part` objects are linked to the representative of their associated object collection (black and grey arrows). Non-representative elements of a collection have an uplink pointer (blue arrows) to the representative or another element. The data structure is divided into levels (indicated by colour saturation) that are only connected by the pointers between representatives and `Part` instances. Note, there is no directed connection from the global table to any of the objects within the left-hand segment of the object graph (Color figure online).

Intuitively, the input letter (a, d) can be dispatched as symbol $(a, \bowtie)$ to every part $\Delta_{d\bowtie}$, for each $\bowtie \in \{<, =, >, \|\}$ and then, within $\Delta_{d\bowtie}$, the subsets $\Delta_{d\bowtie,q} \subseteq \Delta_{d\bowtie}$ for $q \in Q$ are relabelled and merged according to how the letter $(a, \bowtie)$ changes the states q in $\mathcal{A}$. This is the abstract view of how our algorithm processes events.

Based on the ordering on Δ and the subsets $\Delta_{d<,q}$ and $\Delta_{d=,q}$ we can already describe the sets $\Delta_{d>,q}$ and $\Delta_{d\|,q}$ as

$$\Delta_{d>,q} = \bigcup_{i=1}^{\mathsf{lvl}(d)-1} \Delta_{\mathsf{par}^i(d)=,q} \quad \text{and} \quad \Delta_{d\|,q} = \Delta_q \setminus (\Delta_{d>,q} \cup \Delta_{d<,q} \cup \Delta_{d=,q}).$$

Therefore it suffices to store only Δ_q, $\Delta_{d<,q}$ and $\Delta_{d=,q}$ for every $d \in \Delta$ in our data structure. We next describe a concise representation of this (infinite) collection of subsets that allows for performing the necessary operations efficiently.

Components. We identify data values $d \in \Delta$ with program objects and hence use the latter directly in our data structure to represent data values. The only assumption that we need to make is that we can attach additional information to every object, if needed. We represent this information in terms of a class `PObject` that provides the three fields `part`, `uplink` and `table` to store reference pointers to other objects. Technically, we assume every program object in the system to extend this class. In practice, this can be accomplished, e.g., by means of program instrumentation. In the following we therefore regard any program object simply as instance of `PObject`. Additionally, our data structure for storing a PA configuration uses the classes `Part` and `Table`. An instance of `Table` will be used to represent a partition $(\Delta_{d<,q})_{q\in Q}$ of $\Delta_{d<}$ for some particular value $d \in \Delta$. These partitions can be thought of as a (one-dimensional) table indexed

by Q where each cell contains a part $\Delta_{d<,q}$ of the partition. An instance of `Part` will in turn represent such a part.

Based on these components we store the subsets of Δ in a hierarchical fashion as depicted in Fig. 2. To every `PObject` corresponding to some data value d we associate a `Table` instance that holds a `Part` object representing the subset $\Delta_{d<,q}$ for every state $q \in Q$. A `Part` object now maintains a *collection* of objects that represent subsets of $\Delta_{d<,q}$. The collection can contain both instances of `Part` and of `PObject` representing subsets $\Delta_{d'<,q}$ and $\Delta_{d'=,q}$, respectively, for direct children d' of d. While the former in turn represent a possibly empty collection of objects, the latter indicate that the set $\Delta_{d'=,q}$ is non-empty, i.e. $c(d') = q$. Every `PObject` in the collection again carries a table pointing to subsets one level deeper in the data structure and every `Part` object is associated with a possibly empty collection of objects.

At the top of the data structure there is one designated `Table` instance that we refer to as `globalTable`. It represents the partition $(\Delta_q)_{q \in Q}$ and hence maps every state $q \in Q$ to a `Part` object representing the part Δ_q. The collection of these `Part` objects now contain the program objects with minimal IDs d and corresponding sup-parts $\Delta_{d<,q}$.

Unobserved Values. A configuration assigns a state to all (infinitely many) data values whereas only finitely many objects are actually observed during execution. We consider an object (ID) observed if it is associated to some event that occurred or it has a smaller ID (wrt. $(\Delta, \leq)$) than an observed object. The mapping of unobserved values to states is stored symbolically: every `Table` object holds a `default` field storing a state $q \in Q$. An unobserved ID is mapped to the default state of the table attached to its larges ancestor. Note that all unobserved values with the same largest observed ancestor cannot be distinguished because they always fell into the same projection class along a run.

Union-Find. The object collections attached to `Part` instances are maintained using a nesting of union-find data structures. This is the most crucial aspect regarding the performance of the monitoring algorithm. It allows for efficiently performing all operations that are necessary to update a configuration: computing the *union* of two parts, to *insert* and *delete* elements and to identify (*find*) the `Part` object that holds a given element.

Recall that a union-find structure represents disjoint sets of objects organised as a tree. One element (if any) of each set is appointed *representative* and used as root while all others carry a reference to one other member of the same set. For convenience we consider objects that can be inserted into a union-find structure as `Findable`. We assume that `Part` as well as `PObject` extend this class providing the references `uplink` and `part`. The former links an element to its parent in the union-find tree but we use the term *uplink* to avoid confusion. The `part` field is only used by the representative to point to the `Part` object that holds the set.

Classically, the operations *find* and *insert* operate on representatives of a set but since we are mostly interested in the associated `Part` object we assume operations with signatures

```
fun find(obj: Findable): Part
proc insert(target: Part, obj: Findable)
```

where `find` returns the content of the `part` field of the representative and `insert` adds an object to the collection attached to a `Part` object. For the same reason we use the operation

```
proc moveAll(target: Part, source: Part)
```

that is derived from the basic operation *union* and moves all elements from the collection attached to `source` to the collection attached to `target`. Moreover, we assume the union-find structure provides an operation

```
proc delete(obj: Findable)
```

which can be implemented in different ways while maintaining the worst-case complexity of the other operations [3,16].

Helper Functions. To facilitate the presentation of the algorithm we employ the helper functions

```
fun part(table: Table, state: Q): Part
fun state(table: Table, part: Part): Q
fun createTable(parentTable: Table, default: Q): Table
```

that can easily be implemented based on the information present in the data structure. The function `part` returns the `Part` object that the given state is mapped to by the given table. Conversely, `state` returns the state that the given table maps to the given `Part` object. It is assumed that the latter is indeed referenced by the table and that the state is unique. The function `createTable` creates a new `Table` object with the given default state. For every table index $q \in Q$ a new `Part` object is created and moreover inserted into the part for q in `parentTable`. The object collection attached to itself is initially empty. Our algorithm accesses the ordering on Δ by means of `par` and the functions

```
fun hasParent(obj: PObject): Boolean
fun parentTable(obj: PObject): Table
```

where `hasParent(obj)` is true if the ID of `obj` is not minimal. For every program object `parentTable` returns the `Table` object associated with its parent or `globalTable` if it is minimal. It is assumed that the object and, if existent, its parent object have already been registered in the data structure as described below in Sect. 4. Note that the ordering is not represented in the data structure as described above. In Sect. 5 we discuss how the ordering information can be made available in our setting.

Output. Considering the output v of the PA $\mathcal{A}$ in configuration c we observe that $v = \bigsqcap_{d \in \Delta} \lambda(c(d)) = \bigsqcap_{q \mid c^{-1}(q) \neq \emptyset} \lambda(q)$ where $c^{-1}(q) = \{d \in \Delta \mid c(d) = q\}$ is the inverse of c. It hence suffices to evaluate which of the sets Δ_q are non-empty. Since evaluating every `Part` object in the data structure is not an option—in fact, `Part` objects are not necessarily reachable—we track the number of objects in a field `counter` attached to every `Part` object. When performing a specific operation, the local counters can easily be updated. By propagating local counter changes upwards the tree structure the counters for the parts Δ_q can invariantly provide the number of program objects mapped to a specific state.

Recall that the part corresponding to the default state q in a table virtually contains unobserved objects. These cannot be distinguished and we therefore treat them as a single one and add one to the counter value of that part.

4 Monitor Execution Algorithm

Based on the data structure described in the previous section we now present an algorithm that simulates one step of the operational semantics of some PA $\mathcal{A} = (Q, \Sigma, \delta, q_0, \lambda)$. The main procedure `step` of the algorithm is shown in Listing 2. It takes an event name $a \in \Sigma$ and a `PObject` instance and updates the data structure such that it represents the successor configuration of $\mathcal{A}$ after reading a letter $(a, d) \in \Sigma \times \Delta$ where d represents the object's ID. In the following, we identify `PObject` instances with data values from Δ representing their ID. Moreover, we identify `Part` objects with the subset of Δ they represent. The procedure `step` essentially dispatches the input letter (a, d) to the parts $\Delta_{d<}$, $\Delta_{d=}$, $\Delta_{d>}$ and $\Delta_{d\|}$ as symbols $(a, <)$, $(a, =)$, $(a, >)$ and $(a, \|)$, respectively. Assume the data structure encodes a configuration c of $\mathcal{A}$.

Updating $\Delta_{d=}$. Updating the part $\Delta_{d=}$ requires only to change the state $q = c(d)$ of the object d to another state $q' = \delta(q, (a, =))$. This is implemented by the procedure `changeState` depicted in Listing 1. It removes the object d from its current part $\Delta_{\mathsf{par}(d)<,q}$ and inserts it into the part $\Delta_{\mathsf{par}(d)<,q'}$. Removing d amounts to deleting d from the union-find structure associated with the `Part` object $\Delta_{\mathsf{par}(d)<,q}$ and consequently decrementing its counter. Subsequently, the procedure `setState` inserts d into the (collection associated with the) target part $\Delta_{\mathsf{par}(d)<,q'}$ and increments its counter to update the size information. As our data structure maintains nested parts, changing the size of a part requires to propagate this change to all enclosing parts. The procedure `updateCounter` realises this functionality. It calls `find` recursively to determine all enclosing parts until a top most part Δ_q is reached and updated.

Updating $\Delta_{d<}$. All elements from the part $\Delta_{d<}$ need to be updated according to the symbol $(a, <)$ upon reading (a, d). How this symbol changes the states of these can simply be described by the mapping $\mathtt{map} : Q \to Q$ with $\mathtt{map}(q) = \delta(q, (a, <))$. As we aim to be efficient we must not explicitly handle every element below the `Part` object $\Delta_{d<}$ in the data structure. Instead, we rearrange only the `Table` object associated to d: depending on `map`, the parts $\Delta_{d<,q}$ are joined or moved, i.e., new `Part` objects $\Delta'_{d<,q} := \bigcup_{q' \mid \mathtt{map}(q')=q} \Delta_{q'}$ are created for every state $q \in$. The function `applyMap` creates these new parts and computes their counters based on the counters of the original parts. After applying the mapping, it only remains to propagate the counter changes upwards in the data structure to all enclosing parts.

Notice that this way, the data structure becomes inconsistent since the changes are not automatically propagated downward the data structure to all larger objects. In a consistent state (cf. Fig. 2) every part object $\Delta_{e<,q}$ is contained in the collection of the part object $\Delta_{\mathsf{par}(e)<,q}$ for the same state q. Applying the map may, e.g., effectively relabel some $\Delta_{\mathsf{par}(e)<,q}$ to $\Delta_{\mathsf{par}(e)<,q'}$ and then

Listing 1. Procedures operating on the data structure

```
proc changeState(obj: PObject, q: Q) {
  updateCounter(find(obj), -1)
  delete(obj)
  setState(obj, q) }

proc setState(obj: PObject, target: Q) {
  val targetP =
    part(parentTable(obj), target)
  insert(targetP, obj)
  updateCounter(targetP, 1) }

proc updateCounter(startP: Part,
                   delta: Int) {
  if (startP == null) return
  startP.counter += delta
  updateCounter(find(startP), delta) }

proc changeStatesIncomp(obj: PObject,
    anchor: PObject, map: Q → Q) {
  val state = state(parentTable(obj),
    find(obj))
  if (hasParent(obj)) {
    changeStatesIncomp(
      par(obj), anchor, map)
  } else {
    globalTable =
      applyMap(globalTable, map)
    if (hasParent(anchor)) {
      pullUpdates(par(anchor)) } }
  changeState(obj, state) }

proc pullUpdates(obj: PObject) {
  if (hasParent(obj)) pullUpdates(par(obj))
  fun map(q: Q): Q = state(
        parentTable(obj),
        find(part(obj.table, q)))
  obj.table = applyMap(obj.table, map) }

proc changeStates(obj: PObject, map: Q → Q) {
  val oldTab = obj.table
  obj.table = applyMap(oldTab, map)
  foreach q in Q {
    updateCounter(part(parentTable(obj), q),
      part(obj.table, q).counter
        - part(oldTab, q).counter) } }

fun applyMap(tab: Table, map: Q → Q): Table = {
  val newTab = createTable(tab, map(tab.default))
  foreach q in Q {
    val source = part(tab, q)
    val target = part(newTab, map(q))
    moveAll(target, source)
    target.counter += source.counter }
  return newTab }

proc register(obj: PObject) {
  if (obj.table != null) return
  if (hasParent(obj)) register(par(obj))
  val default = parentTable(obj).default
  obj.table =
    createTable(parentTable(obj), default)
  updateCounter(part(obj.table, default), 1)
  setState(obj, default) }

proc dismissUpdates(obj: PObject) {
  foreach q in Q {
    val displaced = part(obj.table,q)
    delete(displaced)
    insert(part(parentTable(obj), q),
           displaced)
  }}
```

$\Delta_{e<,q}$ is enclosed by the part $\Delta_{\mathsf{par}(e)<,q'}$, although not being a subset. However, this inconsistency only means that the parts $\Delta_{e<,q}$ did not yet receive the transition from q to q'. We can recover the correct state by determining the outmost enclosing part and consulting the global table for its state. The procedure `pullUpdates` in Listing 1 implements this functionality. We will, however, only use it if necessary, meaning propagation of such changes is lazy. Note that, in contrast to `setStates` no counter updates must be propagated.

Updating $\Delta_{d\|}$. The essential idea for updating $\Delta_{d\|}$ is to save the state of d and all the ancestors $e < d$ of d, apply the update for $(a, \|)$ to the *global* table, i.e., to all objects, and then *restore* the saved states of the ancestors and d. That way precisely all incomparable objects are affected. Most of this process is implemented by the recursive procedure `changeStatesIncomp` shown in Listing 1. Notice, that before restoring the states of d and its ancestors, the changes made to the global table need to be propagated to d. Otherwise restoring would not have an effect and upon the next update the unintended modifications would still be applied. It remains to restore the state of the larger elements in the part $\Delta_{d<}$ afterwards. This is implemented independently in the procedure `dismissUpdates`. This procedure deletes for every q the part associated with q in the table of d from its current enclosing part and inserts it into the part associated with q in the parent table. Thus it corrects the inconsistency based on the information in the *local* table instead of the information in the global table, as done by `pullUpdates`.

Listing 2. Main procedure

```
1  proc step(obj: PObject, event: Σ) {
2    register(obj)
3    pullUpdates(obj)
4
5    fun mapGT(q: Q): Q = δ(q, (event, <))
6    changeStates(obj, mapGT)
7
8    changeState(obj,
9      δ(state(parentTable(obj), find(obj)),
10       (event, =)))
11   var obj2 = obj
12   while (hasParent(obj2)) {
13     obj2 = par(obj2)
14     changeState(obj2,
15       δ(state(parentTable(obj2), find(obj2)),
16         (event, >))) }
17   fun mapIC(q: Q): Q = δ(q,(event, ||))
18   changeStatesIncomp(obj, obj, mapIC)
19   dismissUpdates(obj)
20 }
```

Procedure `step`. Consider the main procedure `step` in Listing 2 called for an event $a \in \Sigma$ and object $d \in \Delta$. It first calls `register` to ensure d has been properly registered with our data structure. Notice that when creating a new table for the object, all parts are, technically, empty. However, the part corresponding to the default state in the table above virtually contains unobserved objects. We therefore increment its counter by one. Then, `pullUpdates` is used to ensure that the table associated with the observed object d is consistent with respect to the global table. In lines 5–6 and 8–10 of Listing 2 the parts $\Delta_{d<}$ and $\Delta_{d=}$ are updated, respectively, as described above. The lines 11–16 update the part $\Delta_{d>}$ of smaller objects according to the symbol $(a, >)$. This case can be handled by determining all affected objects explicitly using function `par`. Then the corresponding target state is computed and assigned similarly as in the case of $\Delta_{d=}$. Finally, lines 17–19 handle $\Delta_{d\|}$. As before a function `mapIC` is defined mapping source to target states for transitions labelled by $(a, \|)$ and the procedure `changeStatesIncomp` is called, followed by the restore operation as described above.

Complexity. It is crucial to know how the performance of a monitoring algorithm depends on the behaviour of the monitored program. For the following analysis, we fix a PA $\mathcal{A}$ with s control states and assume that the data domain $(\Delta, \leq)$ is of bounded depth ℓ. Let $A_k(i)$ be Ackermann's function defined as $A_0(i) := i + 1$ and $A_{k+1}(i) := A_k^{i+1}(i)$ where $f^j(x)$ is the function f iterated j times on x. Following [2], we define the inverse of Ackermann's function as $\alpha(i, j) := \min\{k \geq 2 \mid A_k(i) > j\}$ and $\alpha(i) := \alpha(i, i)$. We observe that the execution time of `step` is dominated by the calls to operations on union-find data structures and that it causes $\mathcal{O}(s \cdot \ell + \ell^2)$ calls to `find` and $\mathcal{O}(s \cdot \ell)$ calls to `union`- and `delete`-operations. If our data structure contains n program objects, the size of every union-find structure in it is bound by $s \cdot n$. Then, the `find`-operations can be realised in $\mathcal{O}(\log(s \cdot n))$ worst-case time and $\mathcal{O}(\alpha(s \cdot n))$ amortised time; all other operations can be realised in constant time [2]. Hence, for fixed s and ℓ, the worst-case and amortiseed execution time of `step` on a data structure containing n program objects is in $\mathcal{O}(\log(n))$ and in $\mathcal{O}(\alpha(n))$, respectively.

Note, that our data structure only requires space linear in the number of observed objects. Furthermore, the factor ℓ^2 for the number of `find`-calls arises only from the update of the set $\Delta_{d>}$ in lines 11–16 and $\Delta_{d\|}$ in lines 17–19 of Listing 2. There, `setState` is called at most ℓ times which causes in turn up to ℓ `find`-calls to adjust the counters. Updating the counters for ℓ consecutive

`setState`-calls could be implemented accumulatively with only ℓ `find`-calls instead. An optimised implementation of `step` therefore provides a worst-case and amortised time complexity in $\mathcal{O}(s \cdot \ell \cdot \log(n))$ and $O(s \cdot \ell \cdot \alpha(n))$, respectively.

5 Implementation and Evaluation

We have implemented our approach in Java as the tool Mufin. Properties are specified in Java by defining automata using a simple Java API. In addition the required tree-ordering on data values and the mapping of program events to unary logical events has to be provided. We use AspectJ intercept program events, such as method invokations, and dispatch them to Mufin.

In the presentation of the algorithm in Sect. 4 we assumed direct access to the tree-ordering on data values and used the function `par` to obtain the parent of a program object. An implemented of such a function depends on the setting as the order used for the specification may not be directly represented in the monitored program or might be hard to access. Mufin uses special events from which this order can be observed. Consider again the example from Sect. 1. When a new iterator is created the implementation can access both, the iterator and the corresponding collection. As the collection has to be the parent of the iterator the implementation can store this information, e.g. using a pointer from the iterator to the collection. Since our monitoring algorithm requires that all smaller objects are known when an event occurs, we also require these special events to occur on an object before any other events. The implementation detects when an event occurs on an object where the parent object is not yet known or when a special event occurs that conflicts with a previously observed event.

While we assumed to use program objects directly in the conceptual presentation, our implementation adds only one additional field to program objects that points to auxiliary objects actually contained in the data-structure. As program objects are not referenced from inside the union-find structure, they can be garbage collected as soon as they are no longer referenced by the original program. Also, the `delete` operation simply marks these auxiliary objects as deleted and they are only cleaned up during `find`-operations. The obvious consequence is that unnecessary auxiliary objects might pile up within a union-find structure. However, this does not happen as long as events occur regularly involving every observed program object. The assumption that almost all program objects, that are not ready for garbage collection, will always occur in some future event seems to be reasonable for many applications. The advantage of this approach is that garbage collection does not require any additional consideration. Classical union-find structures only require upward references in direction of the representative element of a part. Efficient implementations of the `delete`-operation also require further references in the reverse direction. Assuming that `find`-operations are performed regularly on most elements, most elements will not be referenced by any other element. Once they are no longer reference by a program object they will thus be garbage collected. Using an implementation with efficient deletes would require to use the API of the Java garbage collector in order to trace when some observed program object is garbage collected

which would come with some performance overhead on its own. While this is an option when requiring strict guarantees, our benchmarks show that our simpler approach works well.

Instrumenting the elementary object class requires to modify the Java Virtual Machine (JVM). To avoid this, Mufin can also use a hash table to map program objects to auxiliary objects instead of a reference. This variation, called Mufin Light, has a notable impact on runtime and memory overhead, however, the advantage of our algorithm remains as our benchmarks show.

Evaluation. Mufin took part in the Java track of the recent *2nd Competition on Runtime Verification* [14]. We selected the seven benchmarks with properties expressible in our formalism of the 14 submitted to the competition. All benchmarks comprise a property and a small program generating a sequence of events. Monitoring the given property involves keeping track of nearly all the objects of the program. Therefore, the benchmarks are very well suited to compare the performance of different tools. For real-world applications a far smaller overhead can be expected as usually only a fraction of objects and events will be observed. Projection automata for the benchmarks are depicted in Fig. 1.

Benchmarks. The first group of benchmarks comprises *Iterator*, already described in Sect. 1, and three variations: *SafeIterator* uses the same property but instantiates far more objects (several millions instead of about ten). *MapIterator* enforces a similar property where iterators are created for key sets of a map and modifications occur on the map, thereby requiring three instead of only two levels of objects. It also creates several millions of objects. *DelayedIterator* is a variation of *Iterator* where the next-method may be called one time after a modification of the collection without failing. These benchmarks are very common for the evaluation of online monitoring tools, e.g. in [19] only properties of this kind are considered. *Multiplexer* aims to show the effect of a property requiring more control states. It models a multiplexer with four channels where an arbitrary number of clients is connected to each channel. New clients can be attached to the active channel (c), removed (r) and used (u) and the active channel can be switched (n). Using a client attached to an inactive channel violates the property. *Toggle* is designed to demonstrate the effect of a global action affecting a large number of objects. Objects can be created (c) and the state of all existing objects can be toggled (t). Objects may only be processed (p) if they are in one of their two internal states. *Tree* provides a scenario were the maximal level of observed objects is not known in advance. Objects are created as inner nodes (c_i) or leafs (c_l) of a tree. Messages sent (s) on any node are dispatched to corresponding leafs with an input buffer of size one and processed (p) there. Conversely, a reset (r) clears the buffer of corresponding leafs. A critical send operation (s_c) requires the buffer of all receiving leafs to be empty. Finally, any node can be invalidated (i) effectively removing it from the tree.

Results. We executed the benchmarks with Mufin, Mufin Light, JavaMOP and MarQ and measured execution time and memory consumption of the complete JVM process. Figure 3 shows relative time and memory overhead, i.e. additional

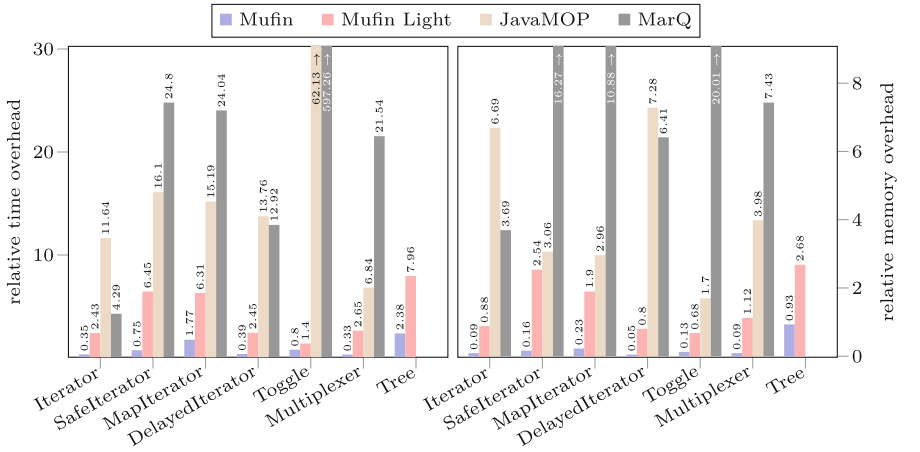

Fig. 3. Relative time and memory overhead of the tools Mufin, JavaMOP and MarQ while monitoring the given properties on the benchmark programs. A relative time overhead of 1 means that the absolute monitoring overhead is equal to the execution time of the non-instrumented program. (The difference between the instrumented and non-instrumented benchmark is the absolute overhead.)

time and memory consumption divided by that of the unmonitored program. Mufin (in both variants) is always multiple, often more than ten, times faster than JavaMOP and MarQ while consuming far less memory. Comparing Mufin with Mufin Light shows a notable impact of the global hash table but the performance benefit of our approach clearly persists. Comparing *Iterator*, *DelayedIterator* and *Multiplexer* shows that the number of states in a specification has only a small effect on the overhead of Mufin. Comparing *SafeIterator* and *MapIterator* shows that the impact of an addition level is small as well. The measurements for *SafeIterator* and *MapIterator* also show that Mufin handles large numbers of instantiated objects far better than the other tools. The results for *Toogle* demonstrate the massive impact of actions affecting many objects at once. In this benchmark almost every step affects around 10 000 objects rendering the previous approaches practically infeasible. The benchmark *Tree* can not be specified using the formalisms of the other tools. It shows that the overhead of Mufin grows for a greater depth of the ordering and thus of the data structure (in this case up to 7) but remains acceptable. The memory overhead of Mufin Light is significantly larger than that of Mufin, the latter remaining very small (below 1) in all cases. This is most likely due to hash tables that can only be filled up to a certain degree without becoming extremely inefficient. Some variations in memory consumption may be due to the allocation strategy of the JVM and the memory measurements therefore only show a general tendency. Mufin is available for download[1].

[1] http://www.isp.uni-luebeck.de/mufin.

6 Conclusion

Our investigations on monitoring temporal properties of object-oriented systems show that complex constraints, including hierarchical dependencies between individual objects, can be evaluated efficiently at runtime. We demonstrated that union-find data structures are a valuable algorithmic tool for runtime analysis. In the proposed monitoring algorithm they provide strict guarantees on the execution time of a monitoring step. This ensures that the accumulated runtime overhead grows effectively only linear with the execution time of the monitored program. Our benchmarks show that the conceptual benefits actually apply in practice and can outperform the currently most efficient monitoring tools JavaMOP and MarQ. Our formal model and logical characterisation provide a good understanding of the class of properties our approach can be applied to. Since we exploit their inherent hierarchical structure we clearly pay performance by expressiveness. However, since hierarchical structures are ubiquitous in computing they still cover a wide range of relevant specifications. The class of properties monitorable with our approach can be further extended. For example, some iterator implementations provide a remove method that deletes the current object from the underlying collection. It invalidates all other iterators of the same collection. To handle such constraints, further predicates are needed to address more types of subsets of objects, in this case the set of all siblings of an object. Given our data structure, the algorithm can be extended accordingly. Exploiting the ability to measure the number of objects assigned to some state provides further a basis for evaluating quantitative properties. The underlying model could easily be extended, e.g., by constraints on the number of children of an object in a certain state.

References

1. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L.J., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to AspectJ. In: Proceedings of Object-Oriented Programming, Systems, Languages, and Applications 2005, pp. 345–364. ACM (2005)
2. Alstrup, S., Li Gørtz, I., Rauhe, T., Thorup, M., Zwick, U.: Union-find with constant time deletions. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 78–89. Springer, Heidelberg (2005)
3. Alstrup, S., Thorup, M., Gørtz, I.L., Rauhe, T., Zwick, U.: Union-find with constant time deletions. ACM Trans. Algorithms **11**(1), 1–28 (2014)
4. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.: Quantified event automata: towards expressive and efficient runtime monitors. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 68–84. Springer, Heidelberg (2012)
5. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-based runtime verification. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 44–57. Springer, Heidelberg (2004)

6. Barringer, H., Havelund, K.: TraceContract: a scala DSL for trace analysis. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 57–72. Springer, Heidelberg (2011)
7. Barringer, H., Rydeheard, D.E., Havelund, K.: Rule systems for run-time monitoring: from Eagle to RuleR. In: Sokolsky, O., Taşıran, S. (eds.) RV 2007. LNCS, vol. 4839, pp. 111–125. Springer, Heidelberg (2007)
8. Basin, D., Klaedtke, F., Müller, S.: Policy monitoring in first-order temporal logic. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 1–18. Springer, Heidelberg (2010)
9. Basin, D.A., Klaedtke, F., Müller, S., Zalinescu, E.: Monitoring metric first-order temporal properties. J. ACM **62**(2), 15 (2015)
10. Bauer, A., Küster, J.-C., Vegliach, G.: From propositional to first-order monitoring. In: Legay, A., Bensalem, S. (eds.) RV 2013. LNCS, vol. 8174, pp. 59–75. Springer, Heidelberg (2013)
11. Bojańczyk, M., Lasota, S.: An extension of data automata that captures XPath. Logical Methods Comput. Sci. 8(1) (2012)
12. 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: Proceedings of Temporal Representation and Reasoning 2005, pp. 166–174. IEEE Computer Society (2005)
13. Decker, N., Leucker, M., Thoma, D.: Monitoring modulo theories. Int. J. Softw. Tools Technol. Transfer 1–21 (2015)
14. Falcone, Y., Nickovic, D., Reger, G., Thoma, D.: Second international competition on runtime verification. In: Bartocci, E., et al. (eds.) RV 2015. LNCS, vol. 9333, pp. 405–422. Springer, Heidelberg (2015). doi:10.1007/978-3-319-23820-3_27
15. Havelund, K.: Rule-based runtime verification revisited. STTT **17**(2), 143–170 (2015)
16. Kaplan, H., Shafrir, N., Tarjan, R.E.: Union-find with deletions. In: Proceedings of Symposium on Discrete Algorithms 2002, pp. 19–28. ACM/SIAM (2002)
17. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Rosu, G.: An overview of the MOP runtime verification framework. STTT **14**(3), 249–289 (2012)
18. Reger, G., Cruz, H.C., Rydeheard, D.: MarQ: monitoring at runtime with QEA. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 596–610. Springer, Heidelberg (2015)
19. Rosu, G., Chen, F.: Semantics and algorithms for parametric monitoring. Logical Methods Comput. Sci. **8**(1), 1–47 (2012)
20. Stolz, V., Bodden, E.: Temporal assertions using AspectJ. Electr. Notes Theor. Comput. Sci. **144**(4), 109–124 (2006)
21. Vardi, M.Y.: An automata-theoretic approach to linear temporal logic. In: Moller, F., Birtwistle, G. (eds.) Logics for Concurrency. LNCS, vol. 1043, pp. 238–266. Springer, Heidelberg (1996)