

# Runtime Verification Based on Register Automata

Radu Grigore<sup>1</sup>, Dino Distefano<sup>1</sup>, Rasmus Lerchedahl Petersen<sup>2</sup>, and Nikos Tzevelekos<sup>1</sup>

<sup>1</sup>Queen Mary University of London and <sup>2</sup>Microsoft Research

**Abstract.** We propose TOPL automata as a new method for runtime verification of systems with unbounded resource generation. Paradigmatic such systems are object-oriented programs which can dynamically generate an unbounded number of fresh object identities during their execution. Our formalism is based on register automata, a particularly successful approach in automata over infinite alphabets which administers a finite-state machine with boundedly many input-storing registers. We show that TOPL automata are equally expressive to register automata and yet suitable to express properties of programs. Compared to other runtime verification methods, our technique can handle a class of properties beyond the reach of current tools. We show in particular that properties which require value updates are not expressible with current techniques yet are naturally captured by TOPL machines. On the practical side, we present a tool for runtime verification of Java programs via TOPL properties, where the trade-off between the coverage and the overhead of the monitoring system is tunable by means of a number of parameters. We validate our technique by checking properties involving multiple objects and chaining of values on large open source projects.

## 1 Introduction

Runtime verification [19,22] connotes the monitoring of program executions in order to detect specific error traces which correspond to violations of sought safety properties. In contrast to its static counterpart, runtime verification checks only certain program executions, yet the error reports are accurate as detected violations represent real bugs in the program. In the case of systems with dynamic generation of resources, such as object references in Java, runtime verification faces the key challenge of reasoning about a potentially *unbounded* number of parameter values representing resource identities. Hence, the techniques applicable in this realm of programs must be able to deal with infinite alphabets (this idiom is also known as parametric monitoring). Leading runtime verification techniques tackle the issue using different approaches, such as reducing the problem to checking projections of execution traces over bounded sets of data values (*trace slicing*) [26,20,3,2], or employing abstract machines whose transition rules are explicitly parameterised [9,7,8].

Another community particularly interested in reasoning over similar data domains, albeit motivated by XML reasoning and model-checking, is the one working on automata over infinite alphabets. Their research has been prolific in developing a wide range of paradigms and accompanying logics, with varying degrees of expressivity and effectiveness (see [28] for an overview from 2006). A highly successful such paradigm is that of *Register Automata* [21,24], which are finite-state machines equipped with a fixed number

of registers where input values can be stored, updated and compared with subsequent inputs. They provide a powerful device for reasoning about temporal relations between a possibly unbounded number of objects in a finite manner. In this work we propose a foundational runtime verification method based on a novel class of machines called *TOPL automata*, which connects the field with the literature on automata over infinite alphabets and, more specifically, with register automata.

The key features of our machines are: (1) the use of registers, and (2) the use of sets of active states (non-determinism). From the verification point of view, registers allow us to use a fixed amount of specification variables which, however, can be *re-bound* (i.e. have their values updated). On the other hand, by being able to spawn several active states, we can select different parts of the same run to be stored and processed. These features give us the expressive power to capture a wide range of realistic program properties in a *finite* way. A specific such class of properties concerns *chaining* or *propagation*, which are of focal importance in areas like dynamic taint analysis [25] as well as dynamic shape analysis.<sup>1</sup> In the latter case, we aspire to reason at runtime about particular shapes of dynamically allocated data-structures irrespectively of their size. For example, checking

“*the shape of the list should not contain cycles*” (1)

for lists of any size and in a finite way, requires two activities. First, being able to change the value of the variables in the specification while traversing the list (re-binding). Second, keeping correlations of different elements in the list at the same time (multiple active states).

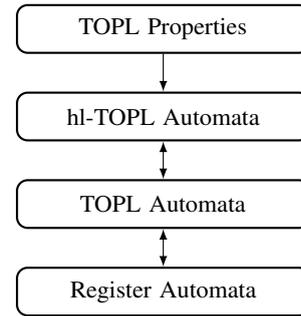
The aim of this work is to exploit the flexibility and the power of registers to address certain properties not expressible with other approaches while, on the other hand, making it easy for programmers to express properties of their code. More precisely, we start from register automata and extend them driven by typical properties required in real-world object-oriented systems. This process results in the definition of two new classes of automata:

- *TOPL* automata, which are low-level and are used for simplifying the formal correspondence with register automata;
- *hl-TOPL* automata, which are high-level and naturally express temporal properties about programs.

We moreover define the ***Temporal Object Property Language (TOPL)***, a formal language which maps directly onto *hl-TOPL* automata and is used for expressing runtime specifications. *TOPL* is a Java-programmer-friendly language where properties look like small Java programs that violate the desired program behaviour. The hierarchy of presented concepts is depicted in Figure 1.

We complement and validate our theoretical results with a practical runtime verification tool for Java programs. The tool can be used by programmers to rigorously express

<sup>1</sup> Although shape analysis is mainly a static technique, we will see in Section 2 that, when doing run-time monitoring, being able to reason about shapes may be vital.



**Fig. 1.** Diagram of the main concepts. The target of each arrow is at least as expressive as its source.

temporal properties about programs, which are then automatically checked by the system. Although the formal correspondence to register automata is completely hidden from the user, it provides a concrete automata-theoretic foundation which allows us to know formally the advantages and limitations of our technique, and also reuse results from the register automata literature. Moreover, our tool can be tuned in terms of coverage, overhead and trace reporting by means of a number of parameters.

*Contributions.* This paper builds upon [18], which introduced the language of TOPL properties and drafted the corresponding automata. Here we clarify the latter, provide a formal correspondence to automata over infinite alphabet, and devise and test a practical tool implementation. In summary, the contributions of the present work are:

- We introduce TOPL and hl-TOPL automata, two classes of abstract machines for verifying systems over infinite alphabets. We prove that both formalisms are equally expressive to register automata by constructing formal reductions between them. The reductions allow us to transfer results from the register automata setting to ours (e.g. decidability of language emptiness, language closures, etc.).

- We define TOPL, a formal specification language designed for expressing program properties involving object interactions over time in a way that is familiar to object-oriented programmers. We moreover present a formal semantics for TOPL, thus making it suitable for static and dynamic program analysis.

- We implement a tool for automatically checking for violations of TOPL properties in Java programs at runtime. A number of parameters are provided for tuning the precision of the system. We furthermore report on experiments in which we ran our tool on large open-source projects. The results are encouraging: for example, we have found an interesting and previously unknown concurrency bug in the DaCapo suite [13].

## 2 Motivating Examples

Interaction among objects is at the core of the object-oriented paradigm. Consider for example Java collections. A typical property one would want to state is

*If one iterator modifies its collection then other iterators of the same collection become invalid, i.e. they cannot be used further.* (2)

The formalisation of the above constraint is non-trivial since it needs to keep track of *several objects* (at least two iterators and one collection) and their *interaction over time*.

A slightly more complex scenario is described in Figure 2. Class `CharArray` manipulates an array of chars, while class `Concat` concatenates two objects of type `Str`. Both classes implement the `Str` interface. Consider the case where `Concat` is used for implementing a *rope*.<sup>2</sup> The operations of a rope (e.g. insert, concat, delete) may update its shape and the references to its root. In this case we may have two or more collections *sharing* some elements. Hence, iterators operating on those different collections may invalidate each other. We need to modify (2), increasing its complexity:

*If one iterator modifies its collection then other iterators of collections sharing some of its elements become invalid, i.e. they cannot be used further.* (3)

---

<sup>2</sup> A rope is a data structure for efficiently storing and manipulating very long strings.

```

interface Str {
    void set(int i, char c);
    char get(int i);
    int len();
    Itr iterator();
}
interface Itr {
    boolean hasNext();
    char next();
    void set(char c);
}
class CharArray implements Str {
    char [] data;
    // ...
}
class Concat implements Str {
    Str s, t;
    public static Concat make(Str s, Str t) {
        /* ... */
    }
    // ...
}

```

**Fig. 2.** A first example: Java code.

*On the need for re-binding.* Let us now suppose we want to perform *taint checking* on input coming from a web form. What we want to check is the property:

Any value introduced by the `input()` method should not reach the `sink()` method without first passing through the `sanitizer()` method. (4)

Although the property may seem simple, its difficulty can vary depending on the context. Consider the case where the input is constructed by concatenating strings from a web form, for example by using ropes implemented with class `Concat`. The number of user inputs, and therefore of concatenations, is not known a priori and is in general unbounded. Consequently, we may end up having an unbounded number of tainted objects. In a temporal specification, we would then need either one logical variable for each of them, or the ability to *rebind* (or *update*) variables in the specification so that we can trace taint propagation. For an unbounded number of objects, rebinding specification variables with different values during the computation helps in keeping the specification finite.

The need for rebinding of variables in the specification arises also in other contexts. For example, when reasoning about the evolving shape of dynamically allocated data-structures. Consider the following loop which uses a list:

```
while (l.next() != null) { ... }
```

If the list `l` contains a cycle, the loop will diverge. Being a violation of a liveness property (termination), divergence cannot be observed at runtime in finite time and therefore it is harder to debug. If we obtained the list by calling a third-party library, we would want to check the property (1) from the Introduction. The *finite* encoding of such properties requires the ability to update the values of specification variables.

### 3 TOPL Automata

We start by presenting some basic definitions. We fix  $V$  to be an infinite set of *values*, with its members denoted by  $v, u$  and variants. Given an arity  $n$ , a *letter*  $\ell$  is an element  $(v_1, \dots, v_n) \in \Sigma$ , where  $\Sigma = V^n$  is the *alphabet*. For  $\ell = (v_1, \dots, v_n)$ , we set the notation  $\ell(i) = v_i$ . Given a size  $m$ , we define the set of stores to be  $S = V^m$ . For a store  $s = (u_1, \dots, u_m)$ , we write  $s(i)$  for  $u_i$ . A *register*  $i$  is an integer from the set  $\{1, \dots, m\}$  identifying a component of the store.

A *guard*  $g$  is a formula in a specified logic, interpreted over pairs of letters and stores; we write  $(s, \ell) \models g$  to denote that the store  $s \in S$  and the letter  $\ell \in \Sigma$  satisfy the guard  $g$ , and we denote the set of guards by  $G$ . An *action*  $a$  is a small program which, given an input letter, performs a store update. That is, the set of actions is some set  $A \subseteq \Sigma \rightarrow S \rightarrow S$ .

Given an alphabet  $\Sigma = V^n$  and a (memory) size  $m$ , we shall define TOPL automata to operate on the set of labels  $\Lambda = G \times A$ , where  $G$  and  $A$  are given by:

$$\begin{aligned} G &::= \text{eq } i j \mid \text{neq } i j \mid \text{true} \mid G \text{ and } G \\ A &::= \text{nop} \mid \text{set } i := j \mid A; A \end{aligned}$$

with  $i \in \{1, \dots, m\}$  and  $j \in \{1, \dots, n\}$ . If  $n = 1$ , then  $(\text{eq } i)$  stands for  $(\text{eq } i 1)$ ;  $(\text{neq } i)$  for  $(\text{neq } i 1)$ ; and  $(\text{set } i)$  stands for  $(\text{set } i := 1)$ . The guards are evaluated as follows.

$$\begin{aligned} (s, \ell) \models \text{eq } i j &\text{ if } s(i) = \ell(j), & (s, \ell) \models \text{true} &\text{ always,} \\ (s, \ell) \models \text{neq } i j &\text{ if } s(i) \neq \ell(j), & (s, \ell) \models g_1 \text{ and } g_2 &\text{ if } (s, \ell) \models g_1 \text{ and } (s, \ell) \models g_2. \end{aligned}$$

The TOPL actions are built up from the empty action,  $\text{nop}(\ell)(s) = s$ ; the assignment action,  $(\text{set } i := j)(\ell)(s) = s[i \mapsto \ell(j)]$  (where  $s[i \mapsto v](k) = s(k)$  if  $k \neq i$ , and  $v$  otherwise); and action composition,  $(a_1; a_2)(\ell) = a_1(\ell) \circ a_2(\ell)$ .

We can now define our first class of automata.

**Definition 1.** A **TOPL automaton** with  $m$  registers, operating on  $n$ -tuples, is a tuple  $\mathcal{A} = \langle Q, q_0, s_0, \delta, F \rangle$  where:

- $Q$  is a finite set of states, with initial one  $q_0 \in Q$  and final ones  $F \subseteq Q$ ;
- $s_0 \in S$  is an initial store;
- $\delta \subseteq Q \times \Lambda \times Q$  is a finite transition relation.

A **configuration**  $x$  is a pair  $(q, s)$  of a state  $q$  and a store  $s$ ; we denote the set of configurations by  $X = Q \times S$ . The *initial* configuration is  $(q_0, s_0)$ . A configuration is *final* when its state is final. The configuration graph of a TOPL automaton  $\mathcal{A}$  as above is a subset of  $X \times \Sigma \times X$ . We write  $x_1 \xrightarrow{\ell}_{\mathcal{A}} x_2$  to mean that  $(x_1, \ell, x_2)$  is in the configuration graph of  $\mathcal{A}$  (we may omit the subscript if  $\mathcal{A}$  is clear from the context).

**Definition 2.** Let  $\mathcal{A}$  be a TOPL automaton. The **configuration graph** of  $\mathcal{A}$  consists of exactly those configuration transitions  $(q_1, s_1) \xrightarrow{\ell}_{\mathcal{A}} (q_2, s_2)$  for which there is a TOPL-automaton transition  $(q_1, (g, a), q_2) \in \delta$  such that  $(s_1, \ell) \models g$  and  $a(\ell)(s_1) = s_2$ .

The **language**  $\mathcal{L}(\mathcal{A})$  of  $\mathcal{A}$  is the set of words that label walks from the initial configuration to some final one:  $\mathcal{L}(\mathcal{A}) = \{ \ell_1 \dots \ell_k \mid x_0 \text{ initial, } x_k \text{ final, } \forall i \leq k. x_{i-1} \xrightarrow{\ell_i}_{\mathcal{A}} x_i \}$ .

A TOPL automaton is **deterministic** when its configuration graph contains no two distinct transitions that have the same source  $x_1$  and are labeled by the same letter  $\ell$ , that is,  $x_1 \xrightarrow{\ell} x_2$  and  $x_1 \xrightarrow{\ell} x_3$  with  $x_2 \neq x_3$ .

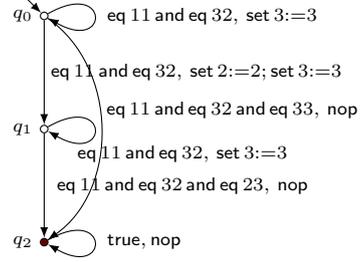
*Example 3.* Consider the language  $\{ abc \in V^3 \mid a \neq c \text{ and } b \neq c \}$ . It is recognized by the following TOPL automaton with 2 registers over the alphabet  $\Sigma = V$ . The values in the initial store  $s_0$  can be arbitrary.

- $Q = \{1, 2, 3, 4\}$ ,  $q_0 = 1$  and  $F = \{4\}$ ;
- $\delta = \{(1, (\text{true}, \text{set } 1), 2)\} \cup \{(2, (\text{true}, \text{set } 2), 3)\} \cup \{(3, (\text{neq } 1 \text{ and neq } 2, \text{nop}), 4)\}$ .

*Example 4.* An involved example is one accepting the language

$$\mathcal{L}_{v_0} = \{ (\text{next}, v_0, v_1)(\text{next}, v_1, v_2) \cdots (\text{next}, v_n, v) \mid \forall i \neq j. v_i \neq v_j \wedge \exists k. v = v_k \}$$

which detects whether a linked list starting from  $v_0$  contains a cycle. Here,  $\text{next}, v_0, \dots, v_n \in V$ , and  $\text{next}, v_0$  are fixed. The automaton, depicted on the side, contains three registers and its initial store is  $(\text{next}, v_0, v_0)$ . Its initial state is  $q_0$ , and its final one  $q_2$ . Register 1 always contains the constant  $\text{next}$ , and eq 11 checks at every transition whether the first value of the letter is  $\text{next}$ . The third register is used for storing the value of the current next node.



Finally, the second register is non-deterministically fed with the value of a list node (set 2 := 2 in  $q_0 \rightarrow q_1$ ), to be matched with a subsequent value (eq 23 in  $q_1 \rightarrow q_2$ ) and thus lead to an error (i.e. final state  $q_2$ ). We can also reach an error by receiving a letter with its next value being the same as its own value (eq 32 and eq 33 in  $q_0 \rightarrow q_2$ ).

*Relation to Register Automata.* There is a natural connection between TOPL automata and *Register Automata* [21,24]. In particular, register automata are TOPL automata with  $n = 1$  and labels from  $\Lambda_R \subseteq \Lambda$ , where

$$\Lambda_R = \{ (\text{fresh}, \text{set } i) \mid i \in \{1, \dots, m\} \} \cup \{ (\text{eq } i, \text{nop}) \mid i \in \{1, \dots, m\} \}$$

and  $\text{fresh} \equiv (\text{neq } 1 \text{ and } \text{neq } 2 \text{ and } \cdots \text{ and } \text{neq } m)$ .<sup>3</sup> In fact, we can show that the restrictions above are not substantial, in the sense that TOPL automata are reducible to register automata, and therefore equally expressive. In the following statement we use the standard injection  $f : (V^n)^* \rightarrow V^*$  such that  $f(\mathcal{L}(\mathcal{A})) = \{ v_1^1 \dots v_1^n \cdots v_k^1 \dots v_k^n \mid (v_1^1, \dots, v_1^n) \cdots (v_k^1, \dots, v_k^n) \in \mathcal{L}(\mathcal{A}) \}$ .

**Proposition 5 (TOPL to RA).** *There exists an algorithm that, given a TOPL automaton  $\mathcal{A}$ , builds a register automaton  $\mathcal{A}'$  such that  $\mathcal{L}(\mathcal{A}') = f(\mathcal{L}(\mathcal{A}))$ . If  $\mathcal{A}$  has  $m$  registers,  $|\delta|$  transitions,  $|Q|$  states, and works over  $n$ -tuples, then  $\mathcal{A}'$  has  $2m + 1$  registers,  $|\delta'| = O(n(2m)^{2m}|\delta|)$  transitions and  $O((2m)^m|Q| + |\delta'|)$  states.*

*High-level Automata* TOPL automata seem to be lacking the convenience one would desire for verifying actual programs. In particular, when writing down a monitor for a specific violation, one may naturally not want to specify all other possible behaviours of the program (which may, though, be of relevance to other monitors). In fact, program behaviours not relevant to the violation under consideration can be *skipped*, ignored altogether. A possible solution for the latter could be to introduce loops with empty guards and actions, with the hope to consume the non-relevant part of the program behaviour. However, such a solution would not have the desired effect: the empty loops could also consume input relevant to the monitored violation.

The above considerations lead us to introduce a new kind of automaton where inputs can be skipped. That is, at each configuration  $x$  of a such an automaton, if an input does

<sup>3</sup> Here a register automaton corresponds directly to what in [24] is called a 1N-RA.

not match any of the guards of the available transitions from  $x$  the automaton will skip that input and examine the next one. In order to accommodate cases where we want specific transitions to happen consecutively, without skipping in between, we allow our automata to operate on sequences of letters, rather than single ones.

**Definition 6.** A *high-level TOPL automaton (hl-TOPL)* is a tuple  $\mathcal{A} = \langle Q, q_0, s_0, \delta, F \rangle$  where:

- $Q$  is a finite set  $Q$  of states, with initial one  $q_0 \in Q$  and final ones  $F \subseteq Q$ ;
- $s_0 \in S$  is an initial store;
- $\delta \subseteq Q \times \Lambda^* \times Q$  is a finite transition relation.

Although the definition of the syntax of high-level automata is very similar to that of ordinary TOPL automata, their semantics is quite different. A **high-level configuration (hl-configuration)** is a pair  $(x, w)$  of a configuration  $x$  and a word  $w$ ; we denote the set of hl-configurations by  $Y = X \times \Sigma^*$ . We think of  $w$  as *yet to be processed*. A hl-configuration is *initial* when its configuration is the initial configuration; that is, it has the shape  $((q_0, s_0), w)$ . A hl-configuration is *final* when its state is final and its word is the empty word; that is, it has the shape  $((q, s), \epsilon)$ , where  $q \in F$  and  $\epsilon$  is the empty word. The hl-configuration graph is a subset of  $Y \times \Sigma^* \times Y$ . We write  $y_1 \xrightarrow{w}_{\mathcal{A}} y_2$  to mean that  $(y_1, w, y_2)$  is in the hl-configuration graph of  $\mathcal{A}$ .

The following concept simplifies the definition of the hl-configuration graph. For each store  $s$  and sequence of pairs  $(g_i, a_i)$  of guards and actions ( $i = 1, \dots, d$ ), we construct a TOPL automaton

$$\mathcal{T}(s, (g_1, a_1), \dots, (g_d, a_d))$$

with set of states  $\{0, \dots, d\}$ , out of which 0 is initial and  $d$  is final, initial store  $s$ , and transitions  $(i-1, (g_i, a_i), i)$  for each  $i = 1, \dots, d$ . Recall that, in this case,  $\ell_1 \dots \ell_d$  is accepted by the automaton when there exist configurations  $x_0, x_1, \dots, x_d$  such that  $x_0 = (0, s)$  and  $x_{i-1} \xrightarrow{\ell_i} x_i$ , for each  $i = 1, \dots, d$ . If the store of  $x_d$  is  $s'$  we say that the automaton can accept  $\ell_1 \dots \ell_d$  with store  $s'$ .

**Definition 7.** The **configuration graph** of a hl-TOPL automaton  $\mathcal{A}$  consists of two types of transitions:

- Standard transitions, of the form  $((q_1, s_1), ww') \xrightarrow{w} ((q_2, s_2), w')$ , when there exists  $(q_1, \bar{\lambda}, q_2) \in \delta$  such that  $\mathcal{T}(s_1; \bar{\lambda})$  can accept  $w$  with store  $s_2$ .
- Skip transitions, of the form  $(x, \ell w) \xrightarrow{\ell} (x, w)$ , when no standard transition starts from  $(x, \ell w)$ .

The **language**  $\mathcal{L}(\mathcal{A})$  of  $\mathcal{A}$  is the set of words that label paths from an initial hl-configuration to a final one:  $\mathcal{L}(\mathcal{A}) = \{w_1 \dots w_k \mid y_0 \text{ initial, } y_k \text{ final, } \forall i \leq k. y_{i-1} \xrightarrow{w_i} y_i\}$ .

*Remark 8.* Note that a TOPL automaton  $\mathcal{A}$  can be technically seen as a high-level one with singleton transition labels. However, its language is in general different from the one we would get if we interpreted  $\mathcal{A}$  as a high-level machine. For example, let  $\mathcal{A}$  be the TOPL automaton consisting of one transition labelled with the guard eq 1, from the initial state to the final state. The alphabet is  $\Sigma = V$  and the initial store has one register

containing value  $v$ . The language of  $\mathcal{A}$  consists of one word made of one letter, namely  $v$ . On the other hand, because of skip transitions, the language of  $\mathcal{A}$  seen as a hl-TOPL automaton consists of all words that contain the letter  $v$ .

*Example 9.* Consider the following hl-TOPL automaton with 2 registers over the alphabet  $\Sigma = V = \{A, B\}$ .

- $Q = \{1, 2, 3\}$ ,  $q_0 = 1$  and  $s_0 = (A, B)$  and  $F = \{3\}$ ,
  - $\delta$  consists of  $(1, [(\text{eq } 2, \text{nop}), (\text{eq } 1, \text{nop}), (\text{eq } 2, \text{nop})], 2)$  and  $(1, [(\text{eq } 1, \text{nop})], 3)$ .
- The language of this is automaton consists of those words in which the first  $A$  is not surrounded by two  $B$ s.

We next present transformations between the two different classes of automata we introduced. First, we can transform TOPL automata to high-level ones by practically disallowing skip transitions: we obfuscate the original automaton  $\mathcal{A}$  with extra transitions to a non-accepting sink state, in such a way that no room for skip transitions is left.

**Proposition 10 (TOPL to hl-TOPL).** *There exists an algorithm that, given a TOPL automaton  $\mathcal{A}$  with  $|Q|$  states, at most  $d$  outgoing transitions from each state, and guards with at most  $k$  conjuncts, it builds a hl-TOPL automaton  $\mathcal{A}'$  with  $|Q| + 1$  states and at most  $(d + k^d)|Q|$  transitions such that  $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$ .*

The converse is more difficult. A TOPL automaton simulates a given hl-TOPL one by delaying decisions. Roughly, there are two modes of operation: (1) store the current letter in registers for later use, and (2) simulate the configuration transitions of the original automaton. The key insight is that Step 2 is entirely a static computation. To see why, a few details about Step 1 help.

The TOPL automaton has registers to store the last few letters. The states encode how many letters are saved in registers. The states also encode a repartition function that records which TOPL register simulates a particular hl-TOPL register or a particular component of a past letter. The repartition function ensures that distinct TOP registers hold distinct values. Thus, it is possible to perform equality checks between hl-TOPL registers and components of the saved letters using only the repartition function. Similarly, it is possible to simulate copying a component of a saved letter into one of the hl-TOPL registers by updating the repartition function. Because it is possible to evaluate guards and simulate actions statically, the run of the hl-TOPL automaton can be completely simulated statically for the letters that are saved in registers.

**Proposition 11 (hl-TOPL to TOPL).** *There exists an algorithm that, given a hl-TOPL automaton  $\mathcal{A}$ , builds a TOPL automaton  $\mathcal{A}'$  such that  $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$ . If  $\mathcal{A}$  is over the alphabet  $V^n$  with  $m$  registers,  $|Q|$  states, and  $|\delta|$  transitions of length  $\leq d$ , then  $\mathcal{A}'$  is over the alphabet  $V$  with  $m' = m + (d - 1)n$  registers,  $O(d^2(m + 1)^m|Q|)$  states, and  $O(d^2(m + 1)^{(m+n)}|\delta|)$  transitions.*

*Remark 12.* Although Propositions 10 and 11 imply that hl-TOPL and TOPL automata are equally expressive, the transformations between them are non-trivial and substantially increase the size of the machines (especially in the hl-TOPL-to-TOPL direction). This discrepancy is explained by the different goals of the two models: TOPL automata are meant to be easy to analyse, while high-level automata are meant to be convenient for

specifying properties of object-oriented programs. The runtime monitors implement the high-level semantics directly.

Since both TOPL and hl-TOPL automata can be reduced to register automata, using known results for the latter [24] we obtain the following.

**Theorem 13.** *TOPL and hl-TOPL automata share the following properties.*

1. *The emptiness and the membership problems are decidable.*
2. *The language inclusion, the language equivalence and the universality problems are undecidable in general.*
3. *The languages of these automata are closed under union, intersection, concatenation and Kleene star.*
4. *The languages of these automata are not closed under complementation.*

The first point of Theorem 13 guarantees that monitoring with TOPL automata is decidable. On the other hand, by the second point, it is not possible to automatically validate refactorings of TOPL automata. Closure under regular operations, apart from negation, allows us to write specifications as negation-free regular expressions. The final point accentuates the difference between property violation and validation.

## 4 TOPL Properties

In this section we describe the user-level *Temporal Object Property Language (TOPL)*, which provides a programmer-friendly way to write down hl-TOPL automata relevant to runtime verification. The full syntax of the language was presented in [18]. Below we give the main ingredients and define the translation from the language to our automata.

A TOPL property comprises a sequence of *transition statements*, of the form

$$\text{source} \rightarrow \text{target} : \text{label}$$

where `source` and `target` are identifiers representing the states of the described automaton. The sequence of statements thus represents the transition relation of the automaton. Each property must include distinguished vertices `start` and `error`, which correspond to the initial and (unique) final states respectively.

The set of labels has been crafted in such a way that it captures the observable events of program executions. Observable events for TOPL properties are method calls and returns, called *event ids*, along with their parameter values. The set of event ids is given by the grammar:

$$E ::= \text{call } m \mid \text{ret } m$$

where  $m$  belongs to an appropriate set of *method names*. Each method name has an *arity*, which we shall in general leave implicit. The set  $V_L$  of possible parameter values is a set of values specified by the programming language (e.g. Java) plus a dummy value  $\perp$ . The set of all values is  $V = V_L \cup E$ .

Labels of TOPL properties refer to registers via *patterns*. A register  $v$  is called a *property variable* and has three associated patterns:

- the uppercase pattern  $V$  matches any value and writes it in the property variable  $v$ ;

- the lowercase pattern  $v$  reads the value of the property variable  $v$  and only matches that value; and
- the negated lowercase pattern  $!v$  reads the value of the property variable  $v$  and only matches different values.

In addition, every element of  $V$  acts as a pattern that matches only the value it denotes, and a wildcard ( $*$ ) pattern matches any value. The set of all patterns is denoted by  $Pat$ . A transition label can take one of the three forms:

$$l ::= call\ m(x_1, \dots, x_k) \mid ret\ x := m \mid x := m(x_1, \dots, x_k)$$

where  $x, x_1, \dots, x_k \in Pat$ . Note that the latter two forms are distinct – the last one incorporates a call and a matching return. Finally, a TOPL property is *well-formed* when it satisfies the conditions:

- each label must contain an uppercase value pattern at most once;
- any use of a lowercase pattern (i.e. a read) must be preceded by a use of the corresponding uppercase pattern (i.e., a write) on all paths from `start`.

From now on we assume TOPL properties to be well-formed.

*From TOPL to automata.* We now describe how a TOPL property  $P$  yields a corresponding hl-TOPL automaton  $\mathcal{A}_P$ . First, if  $n$  is the maximum arity of all methods in  $P$ , the alphabet of  $\mathcal{A}_P$  will be:

$$\Sigma_P = E \times V_L^{n+1}$$

where the extra register is used for storing return values. Note that  $\Sigma_P$  follows our previous convention of alphabets: it is a sub-alphabet of  $\Sigma = V^{n+2}$ . For example, if  $P$  has a maximal arity 5, the event  $call\ m(a, b, c)$  would be understood as  $(call\ m, \perp, a, b, c, \perp, \perp)$  by  $\mathcal{A}_P$ . Here the first component is the event id, the second is a filler for the return value, the next three are the parameter values and the rest are paddings which are used in order for all tuples to have the same length. The event  $ret\ r = m$  would be understood as  $(ret\ m, r, \perp, \perp, \perp, \perp, \perp)$  by  $\mathcal{A}_P$ .

We include in  $\mathcal{A}_P$  one register for each property variable in  $P$  and, in order to match elements from  $V$ , we include an extra register for each element mentioned by  $P$  (this includes all the method names of  $P$ ). Each extra register contains a specified value in the initial state of  $\mathcal{A}_P$  and is never overwritten. The rest of the registers in the initial state are empty. We write  $Pat_P$  for the set of patterns of property variables appearing in  $P$ .

We next consider how labels are translated. The first two forms of label (*call* and *ret*) describe observable events and are translated into one-letter transitions in  $\mathcal{A}_P$ , while the latter form is translated into two-letter transitions. Let  $\{1, \dots, N\}$  be the set of registers of  $\mathcal{A}_P$ . We define three functions:  $reg : Pat_P \rightarrow (N \cup \{\perp\})$  associates a register to each pattern (with  $reg(*) = \perp$ ), while  $grd : Pat_P \times N \rightarrow G$  and  $act : Pat_P \times N \rightarrow A$  give respectively the guard and action corresponding to each pattern and register. We set:

$$grd(x, j) = \begin{cases} true & \text{if } x = V \\ eq\ reg(v)\ j & \text{if } x = v \\ neq\ reg(v)\ j & \text{if } x = !v \\ eq\ reg(x)\ j & \text{if } x \in V \\ true & \text{if } x = * \end{cases} \quad act(x, j) = \begin{cases} set\ reg(v) := j & \text{if } x = V \\ nop & \text{if } x = v \\ nop & \text{if } x = !v \\ nop & \text{if } x \in V \\ nop & \text{if } x = * \end{cases}$$

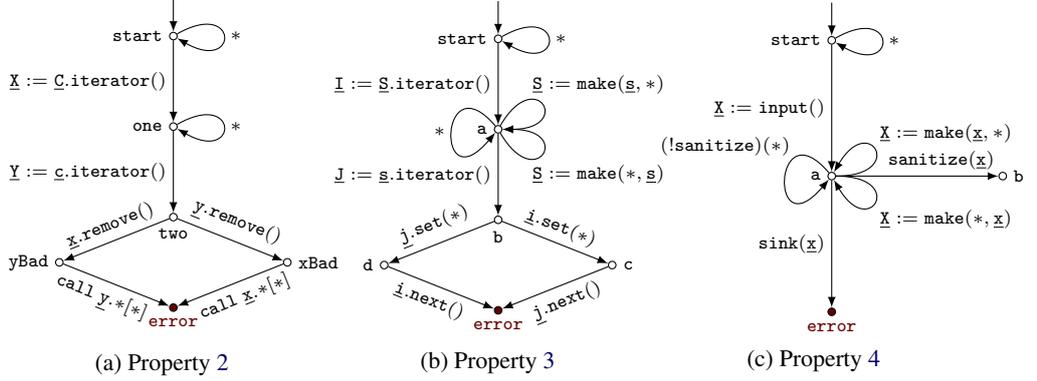


Fig. 3. TOPL formalisations of the example properties from Section 2.

We can now interpret labels of  $P$  into labels of  $\mathcal{A}_P$ . For each a label  $l$  of  $P$ , we define its translation  $\llbracket l \rrbracket = (\llbracket l \rrbracket_G, \llbracket l \rrbracket_A)$ , where  $\llbracket - \rrbracket_G$  and  $\llbracket - \rrbracket_A$  are given as follows.

$$\llbracket l \rrbracket_G = \begin{cases} \text{grd}(m, 1) \text{ and } \text{grd}(x_1, 3) \text{ and } \dots \text{ and } \text{grd}(x_k, k+2) & \text{if } l = \text{call } m(x_1, \dots, x_k) \\ \text{pred}(m, 1) \text{ and } \text{eq } \text{reg}(x) 2 & \text{if } l = \text{ret } x := m \end{cases}$$

$$\llbracket l \rrbracket_A = \begin{cases} \text{act}(x_1, 3) \text{ and } \dots \text{ and } \text{act}(x_k, k+2) & \text{if } l = \text{call } m(x_1, \dots, x_k) \\ \text{act}(x, 2) & \text{if } l = \text{ret } x := m \end{cases}$$

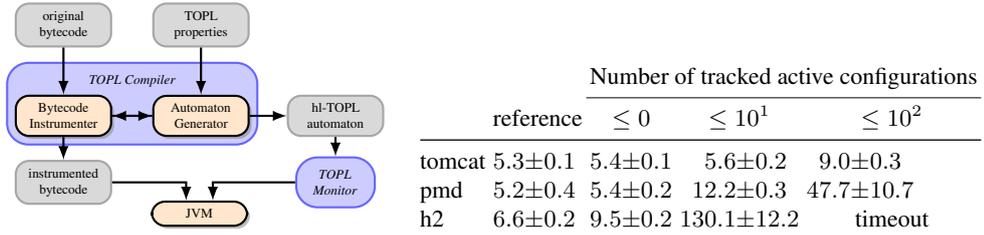
Finally, for the label  $x := m(x_1, \dots, x_k)$ , observe its right-hand-side refers to a call, while its left-hand-side refers to a return. We take this label to mean that  $m$  is called with parameters matching  $x_1, \dots, x_k$  and returns a value matching  $x$ , and no event is observed in the meantime. This is because an intermediate call, for instance a recursive call, could disconnect the method call and the return value. Thus, this label translates into a transition of length two:

$$\llbracket x := m(x_1, \dots, x_k) \rrbracket = \llbracket \text{call } m(x_1, \dots, x_k) \rrbracket \llbracket \text{ret } x := m \rrbracket$$

*Examples.* Figure 3 displays the formal versions of the first three properties that are discussed in Section 2.

(a) This example illustrates how multiple related objects are tracked. In state `two`, the property tracks all pairs of two iterators  $x$  and  $y$  for the same collection  $c$ . If  $x.\text{remove}()$  is called, then state `yBad` becomes active, which precludes further use of  $y$ 's methods. State `xBad` is symmetric.

(b) This example illustrates how chaining of values is tracked, while at the same time tracking multiple related objects. Recall that Property (3) refers to the code in Figure 2 (on page 4). In state `a`, the iterator  $i$  refers to the string  $s$  or some substring of  $s$ . In state `b`, the iterator  $j$  refers to  $s$ . As opposed to the previous property, the two iterators  $i$  and  $j$  are not necessarily for the same collection, but rather for a collection and one of its sub-collections. This property does not refer to the Java standard library, which does



**Fig. 4. Left:** Architecture of the TOPL tool. **Right:** Experimental Results. Times are in seconds, averaged over 10 runs (not in convergence mode).

not implement ropes. There exist, however, several independent libraries that follow the pattern in Figure 2 (e.g. <http://ahmadsoft.org/ropes/>).

(c) This example illustrates sanitization of values, in addition to chaining. In state *a*, the property keeps track of the tainted object *x*. An object is *tainted* if it comes from a specific input method or was made from tainted objects, and was not sanitized. A tainted object must not be sent to a sink.

Of course, the input of the TOPL compiler is not in graphical form. Below we include the actual representation for a property of type (c) without the sanitization option. It specifies actual methods that provide tainted inputs, make tainted objects out of tainted objects, and constitute sinks.<sup>4</sup> We refer the reader to [18] for more example properties.

```
property Taint
  prefix <javax.servlet.http.HttpServletRequest>
  prefix <java.lang.String>
  prefix <java.sql.Statement>
  start -> start: *
  start -> tracking:  $\underline{X}$  := *.getParameter[*]
  tracking -> tracking: *
  tracking -> tracking:  $\underline{X}$  :=  $\underline{x}$ .concat(*)
  tracking -> tracking:  $\underline{X}$  := *.concat( $\underline{x}$ )
  tracking -> error: *.executeQuery( $\underline{x}$ )
```

## 5 Implementation and Experiments

The TOPL tool<sup>5</sup> checks at runtime whether Java programs violate TOPL properties. It consists of a compiler and a monitor (see Figure 4, left).

Given the bytecode of a Java project and several TOPL properties, the compiler produces instrumented bytecode and a hi-TOPL automaton. An instrumented method emits a call event, runs the original bytecode, and then emits a return event. Emitting an event is encoded by a call to the method `check(Event)` of the monitor. The Event

<sup>4</sup> Note that, to ease the task of writing TOPL properties, we have included a `prefix` directive: `prefix p` produces from every method name *m*, an extra name *pm*; it further produces, from any transition involving *m*, a similar transition involving *pm*.

<sup>5</sup> <http://rgrig.github.com/topl>

structure contains an integer identifier and an array of `Objects`. The identifier is unique for each site from which the method check is called. The compiler achieves two tasks that are interdependent: instrumenting the bytecode, and translating properties into an automaton. The instrumentation could be done on all methods of the Java project, but this would lead to high runtime overhead. Instead, the compiler instruments only the methods that are mentioned by the TOPL properties to be checked. Conversely, the translation of properties into automata depends on the Java project's code. To see why, consider a transition guarded in a property by the method name pattern  $m$ . The compiler instruments all the methods whose (fully qualified) names match the pattern  $m$ , and all the methods that override methods whose names match the pattern  $m$ , thus taking into account inheritance. All these instrumented methods emit events with identifiers from a certain set of integers, which depends on the inheritance structure of the Java project. The method name pattern  $m$  is essentially compiled into a set of integer event identifiers.

The monitor is an interpreter for the hl-TOPL automaton that the compiler produces. Its implementation closely follows the semantics from [Section 3](#). For example, the monitor maintains a set of active configurations, which are those reachable by a path labeled by the events seen so far. There are, however, several differences. First, the number of active configurations is not bounded in theory, but a bound may be enforced in practice. Monitoring becomes slower as the number of active configurations increases. As a pragmatic compromise, the user may impose an upper bound, thus trading soundness for efficiency. That is, if the user imposes a bound then monitoring is faster, but property violations may be missed (on the other hand, a reported violation of a property is always a real violation). Second, the implementation includes several optimizations. For example, the guards produced by method name patterns, which require the current event id to be from a certain set of integers, is implemented as a hashtable lookup rather than as a linear search, as the formal semantics would suggest. Third, the implementation saves extra information in order to provide friendlier error messages. For instance, the user may ask the TOPL monitor to save and report the path taken in the configuration graph, or full call stack traces for each event.

*Experimental Results.* We measured the overhead on the test suite DaCapo [13], version 9.12. DaCapo is a collection of automated tests that exercise large portions of code from open-source projects and the Java standard libraries. DaCapo itself has been used for many experiments by the research community. Hence, we did not expect to find any bugs, but aimed instead at measuring the overhead. We checked two types of properties with TOPL. First, properties that express correct usage of the standard Java libraries. Second, properties that express temporal constraints which we extracted from the code comments of three open-source projects (H2, PMD, Tomcat) included in DaCapo. H2 is a database server for which we checked properties on the calling order of some interface methods. For example, a client should not attempt to ask for a row from a cursor unless the latter has been previously advanced. PMD looks for bugs, dead code and other problems in Java code. One of the five properties we checked is “Only if a scope replies that it knows a name, it can be asked for that name's definition”. Tomcat is a highly concurrent servlet server. Servlets are Java programs running in a webserver, extracting data from `ServletRequest`s and sending data to `ServletResponse`s. A response has two associated incoming channels: a stream and a writer. They should not be both used

	original	HasNext		UnsafeIterator		UnsafeMapIterator		UnsafeFileWriter	
		st=3	st=10	st=3	st=10	st=3	st=10	st=3	st=10
avro	8.1	27.8	60.5	163.3	323.1	194.5	179.9	8.3	5.9
batik	1.2	18.1	3.0	3.8	3.8	3.1	3.3	1.3	1.2
eclipse	17.4	24.2	24.0	30.9	41.7	27.2	28.0	22.9	22.8
fop	0.3	0.9	1.9	3.5	3.6	2.7	2.7	0.3	0.3
h2	6.2	5.9	6.8	8.3	20.0	13.5	11.2	6.4	6.0
python	1.9	19.8	46.1	81.5	83.0	62.8	62.7	1.9	1.8
luindex	0.8	0.8	0.8	0.8	0.9	1.0	0.9	0.8	0.9
lusearch	1.5	1.5	1.5	15.0	16.0	13.8	12.8	1.5	1.7
pmd	3.1	19.9	42.6	93.5	240.3	102.6	105.6	3.2	3.3
sunflow	3.9	3.8	3.9	4.0	3.8	3.9	3.9	3.9	4.3
tomcat	2.5	4.2	8.3	22.9	50.9	30.0	31.0	2.6	2.7
xalan	1.5	14.5	7.1	425.0	360.9	272.0	276.5	1.5	1.2

**Table 1.** Experiment on small properties (taken from [25]) run on the DaCapo benchmarks (in convergence mode). HasNext checks that no iterator is advanced without first enquiring hasNext. UnsafeIterator checks that no iterator is advanced after the iterated collection has been modified. UnsafeMapIterator checks that no iterator on keys/values of a map is advanced after the map has been updated. UnsafeFileWriter checks that no file is written to after it was closed. Column original gives the running times (in seconds) for projects without instrumentation of Java standard libraries. The other columns report instrumented runs, with a maximum of 3 and 10 active configurations.

concurrently. But the servlet, before forwarding the response, must call `flush` on the stream, on the writer, or on the response itself. This is one of the properties we checked. Interestingly, while experimenting with Tomcat, TOPL discovered a concurrency bug (a data race) in the DaCapo’s infrastructure which would manifest sometimes as null dereference.

Although our tool is not currently optimised, we measured both time and space overhead. It turns out that space overhead is negligible, below the variance caused by the randomness of garbage collection. Thus, we only report on time overhead, in Figure 4 and Table 1. The relative overhead is meaningful only if the reference runtime is not close to 0, and this is most distinctively the case for test eclipse whose runtime is over 10 s. The (geometric) average overhead in that case is  $\times 1.5$  with  $\leq 3$  active configurations, and  $\times 1.6$  with  $\leq 10$  active configurations. Figure 4 shows the effect of tuning the active configurations in terms of overhead. All experiments were performed on an Intel i5 with 4 cores at 3.33 GHz with 4 GiB of memory, running Linux 2.6.32 and Java VM 1.6.0\_20.

## 6 Related work.

*JavaMOP* [23] and *Tracematches* [2] are based on slicing: A slice is a projection of a word over a finite alphabet; different slices are fed, independently, to machines that handle finite alphabets. Tracematches use regular expressions to specify recognisers over finite alphabets. JavaMOP supports several other logics, via a plugin mechanism, and slices are assigned categories, which can be match/fail or taken from some other set.

Because slices are analyzed independently, it is not possible to express examples such as (1) and (4), which use an unbounded number of register assignments.

*Quantified Event Automata (QEA)* [6] extend the slicing mechanism of JavaMOP with the goal of improving expressivity. Similarly to TOPL automata, QEAs have guards and assignments, which can be arbitrary predicates and transformations of the memory content respectively. In contrast, our automata impose specific restrictions, which follow the expressive power of RAs. In addition, QEAs introduce quantifiers, which can be seen as a way to impose a hierarchy on slices. Systems based on machines with parametric transition rules, such as *RuleR* [9], *LogScope* [7] and *TraceContract* [8], are related to QEAs and are also very similar in spirit to the TOPL approach. RuleR is tuned towards high expressivity and in particular can handle context-free grammars with parameters, which go beyond the reach of TOPL. By comparison, TOPL automata seem a simpler formalism, and this paper demonstrates how they are closely related to standard automata-theoretical models.

*QVM* [3] is a runtime monitoring approach tailored to deployed systems. It achieves high efficiency by being carefully implemented inside a Java virtual machine, checking properties involving a single object, and being able to tune its overhead on-the-fly. On the other hand, TOPL is designed for aiding the programmer during development and testing, and therefore focusses instead on providing a precise and expressive language for specifying temporal properties. For instance, TOPL can express properties involving many objects. Both QVM and TOPL let the programmer tune the overhead/coverage balance. *ConSpec* [1] is a language used to describe security policies. Although ConSpec automata have a countable number of states, they are deterministic and therefore cannot express the full range of TOPL properties.

From the techniques used mostly for static verification of object-oriented programs, *typestates* [29] are probably the most similar to TOPL. A modular static verification method for typestate protocols is introduced in [11]. The specification method is based on linear logic, and relations among objects in the protocol are checked by a tailored system of permissions. Similarly, [15,10] provide a means to specify typestate properties that belong to a single object. The specified properties are reminiscent of contracts or method pre/post-conditions and can deal with inheritance. In [17] the authors present sound verification techniques for typestate properties of Java programs, which we envisage that can be fruitfully combined with the TOPL paradigm. Their approach is divided into several stages each employing its own verifier, with progressively higher costs and precisions. Every stage focuses on verifying only the parts of the code that previous stages failed to verify.

A specification language for interface checking aimed at C programs, called *SLIC*, is introduced in [5]. SLIC uses non-determinism to encode universal quantification of dynamically allocated data and allows for complex code in the automaton transitions; while TOPL specifications naturally express universally quantified properties over data structures and, for effectiveness reasons, there is a limit on the actions performed during automaton transitions. Simple SLIC specifications are verified by the SLAM verifier [4].

Similar investigations have been pursued by the functional programming community. In [16] contracts are used for expressing legal traces of programs in a functional language with references. The contracts specify traces as regular expressions over calls and returns,

thus resembling our automata, albeit in quite a different setting. The specifications are function-centered and, again, capturing inter-object relations seems somewhat tricky.

Finally, as demonstrated in previous sections, TOPL automata are a variant of register automata [21,24], themselves a thread in an extensive body of work on automata over infinite alphabets (see e.g. [28]). RAs form one of the most well-studied paradigms in the field, with numerous extensions and variations (e.g. [14,12,28]).

*Acknowledgements.* We acknowledge funding from EPSRC H011749 (Distefano and Grigore) and RAEng research fellowships (Distefano and Tzevelekos). We thank the anonymous reviewers, for their suggestions significantly improved the paper.

## References

1. I. Aktug and K. Naliuka. ConSpec – a formal language for policy specification. *ENTCS*, 197(1):45–58, 2008.
2. C. Allan, P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA*, pp. 345–364, 2005.
3. M. Arnold, M. Vechev, and E. Yahav. QVM: an efficient runtime for detecting defects in deployed systems. In *OOPSLA*, pp. 143–162, 2008.
4. T. Ball and S. K. Rajamani. The SLAM toolkit. In *CAV*, pp. 260–264, 2001.
5. T. Ball and S. K. Rajamani. SLIC: a specification language for interface checking (of C). Technical Report MSR-TR-2001-21, Microsoft Research, 2002.
6. H. Barringer, Y. Falcone, K. Havelund, G. Reger, and D. E. Rydeheard. Quantified Event Automata: towards expressive and efficient runtime monitors. In *FM*, pp. 68–84, 2012.
7. H. Barringer, A. Groce, K. Havelund, and M. Smith. Formal Analysis of Log Files. *Journal of Aerospace Computing, Information, and Communication*, 7(11), 2010.
8. H. Barringer and K. Havelund. TraceContract: a scala DSL for trace analysis. In *FM*, pp. 57–72, 2011.
9. H. Barringer, D. E. Rydeheard, and K. Havelund. Rule Systems for Run-time Monitoring: from Eagle to RuleR. *J. Log. Comput.* 20(3): 675–706, 2010.
10. K. Bierhoff and J. Aldrich. Lightweight object specification with tpestates. In *ESEC/SIGSOFT FSE*, pp.217–226, 2005.
11. K. Bierhoff and J. Aldrich. Modular tpestate checking of aliased objects. In *OOPSLA*, pp. 301–320, 2007.
12. H. Björklund and T. Schwentick. On notions of regularity for data languages. *Theor. Comput. Sci.*, 411(4-5):702–715, 2010.
13. S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. L. Hosking, M. Jump, H. B. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, pp. 169–190, 2006.
14. M. Bojanczyk, A. Muscholl, T. Schwentick, L. Segoufin, and C. David. Two-Variable Logic on Words with Data. In *LICS*, pp. 7–16, 2006.
15. R. DeLine and M. Fähndrich. Tpestates for objects. In *ECOOP*, pp. 465–490, 2004.
16. T. Disney, C. Flanagan, and J. McCarthy. Temporal higher-order contracts. In *ICFP*, pp. 176–188, 2011.
17. S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective tpestate verification in the presence of aliasing. In *ISSTA*, pp. 133–144, 2006.
18. R. Grigore, R. L. Petersen, and D. Distefano. TOPL: A language for specifying safety temporal properties of object-oriented programs. In *FOOL*, 2011.
19. K. Havelund and G. Rosu. Monitoring Programs Using Rewriting In *ASE*, pp. 135–143, 2001.
20. D. Jin, P. Meredith, C. Lee, and G. Rosu. JavaMOP: Efficient parametric runtime monitoring framework. In *ICSE*, pp. 1427–1430, 2012.
21. M. Kaminski and N. Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2), 1994.
22. M. Leucker and C. Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009.
23. P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Rosu. An overview of the MOP runtime verification framework. *STTT*, 14(3):249–289, 2012.
24. F. Neven, T. Schwentick, and V. Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Logic*, 5(3):403–435, 2004.
25. J. Newsome and D. X. Song. Dynamic Taint Analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.
26. G. Rosu and F. Chen. Semantics and algorithms for parametric monitoring. *LMCS*, 8(1), 2012.
27. H. Sakamoto and D. Ikeda. Intractability of decision problems for finite-memory automata. *Theor. Comput. Sci.*, 231(2):297–308, 2000.
28. L. Segoufin. Automata and logics for words and trees over an infinite alphabet. In *CSL*, pp. 41–57, 2006.
29. R. E. Strom and S. Yemini. Tpestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986.