Runtime Verification Based on Register Automata

Dino Distefano* Radu Grigore*
Rasmus Lerchedahl Petersen† Nikos Tzevelekos*

Abstract

We introduce a new formal framework aimed at checking temporal safety properties of object-oriented programs. The underpinning theoretical foundations of our framework is an extended class of *register automata*. As registers' values can be updated, our automata are able to express highly complex properties of systems involving multiple interacting objects. Specifications are then used to instrument Java bytecode so that their violations can be automatically detected at runtime. The tradeoff between the coverage and the overhead of the monitoring system is tunable by means of a number of parameters. We validated our technique by checking several safety properties on large open source projects.

1 Introduction

Runtime verification is a technique where the behaviour of programs is monitored at runtime to check whether executions can violate certain safety properties. Systems for runtime verification stand somehow in between classic verification and testing: they work on the actual system (as testing) but check violations of temporal properties as, for example, it is done in model checking. Compared to traditional formal verification techniques, runtime verification systems check only certain program executions, however, the error reports are accurate and detected violations represent real bugs in the program.

In the context of object-oriented programming leading runtime verification techniques include JavaMOP [22], QVM [4], Tracematches [2], and techniques based on typestate [29, 9, 23, 15, 6]. Although powerful, these methods present some limitations. For example, either in terms of language expressivity, or in the ability to express concisely properties of the interaction of several objects over time (typestate), or in concisely expressing properties requiring an unbounded number of different object re-bindings (sometimes called parameters in the runtime verification community).

*Queen Mary University of London
†Microsoft Research Cambridge

Register Automata [21] are automata where a number of registers are used to store and compare data. The values of registers can be updated over time, and data can be taken from unbounded domains (e.g., the set of object references). Thus register automata provide a powerful device for reasoning about temporal relations of a (possibly unbounded) number of objects in a finite manner. They have been studied extensively from the theoretical point of view and essential results for their analysis have been established (e.g., decidability of emptiness, closure under intersection, and so forth).

Thus, it seems sensible to attempt an approach to runtime verification taking register automata as the starting point. The aim is to exploit the flexibility and the power of registers to address certain properties not easily dealt with in other approaches. On the formal side, we started from register automata and have extended them driven by typical properties required in real-world object-oriented systems. The extension has been careful crafted to be able to define the precise mathematical connection with register automata, and yet, make it easy for programmers to express properties of their code. This process has resulted in the definition of two new classes of automata: a high-level and a low-level one. In the high-level automata, temporal properties about programs are naturally expressed. Low-level automata, on the other hand, simplify the formal correspondence with register automata. Finally we defined a formal language – called TOPL (Temporal Object Property Language) – mapping directly on the high-level automata.

We complement the theoretical construction with a practical tool. We have implemented TOPL in a runtime parametric verification framework that can be used by Java programmers to express rigorous temporal properties about their programs. These properties will then be automatically checked by the system. The formal correspondence between the properties and register automata defined in this paper gives a rigorous mathematical foundation to the approach and allows us to reuse many of the results for register automata on TOPL automata. For example, given a TOPL property, we can decide whether its language is empty. The formal development is completely hidden from the programmer. Our system can be tuned in terms of coverage, overhead, and trace reporting by means of a number of parameters.

Contributions. In summary, the contributions of this paper are:

- We define TOPL, a formal specification language tailored to express properties involving object interactions over time in a way that is familiar to object-oriented programmers.
- We define a precise formal semantics for TOPL, making it suitable for program analysis, both static and dynamic.
- We define a precise formal correspondence between TOPL automata and register automata. In particular we prove that they are equally expressive. As our proof is constructive, this implies that the decidability results for

register automata hold also for TOPL automata. Consequently we can decide several features of TOPL specifications (e.g., language emptiness).

- We have implemented our parametric framework in a tool that automatically check at runtime violations of TOPL properties in Java programs. A number of parameters are used to tune the precision of the system.

- We report on experiments of running our tool on large open-source projects. The results are encouraging (e.g., we have found an interesting and previously unknown concurrency bug in the DaCapo suite).

The paper is organized as follows. Section 2 gives some example properties motivating TOPL. Section 3 provides background of register automata and defines the basis of our formalism. Section 4 introduces TOPL automata and proves the equivalence with register automata. Section 5 defines the TOPL property language and its semantics. Section 6 describes several example properties using TOPL. Section 7 describes the implementation of the runtime verification system. Section 8 discusses the experimental results and Section 9 related work. Finally, Section 10 concludes the paper and describes our plans for future work.

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 existing iterators of the same collection become invalid, which means they cannot be used further. (1)

The formalization 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 by classes in Figure 1. Class `CharArray` manipulates an array of chars and implements the `Str` interface. Class `Concat` is used to concatenate two object of type `Str`. Also `Concat` implements `Str`. Consider the case where `Concat` is used to implement a *rope* data structure.[1] Rope's operations like insert, concat, delete may update the shape of the tree and references to its root. In this case we may have two (or more) collections *sharing* some elements. Consequently iterators operating on these different collections may invalidate each other. We need to modify (1) increasing its complexity:

If one iterator modifies its collection, then other existing iterators of collections sharing some elements become invalid, which means they cannot be used further. (2)

[1] A rope is a binary tree shaped data structure for efficiently storing and manipulating very long strings [11].

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

Figure 1: A first example: Java code

Re-binding of specification variables. Now let's consider the case where we want to perform *taint checking* for input coming from a web form. What we want to check is the following property:

Any value introduced by the `input()` *method should not reach the* `sink()` *method without first passing through the* `sanitizer()` *method.* (3)

This property is at first sight simple. However, it's difficulty can vary depending on the context. Consider in fact the case where the input is constructed by concatenating strings from the web form, by for example, using ropes implemented with class `Concat`. The number of user inputs, and therefore of concatenations, is not known a priori and in general unbounded. Consequently we will have a possibly unbounded number of tainted objects. In a temporal specification, we would then need either different logical variables for each different tainted objects or the ability to *re-bind* (or *update*) variables in the temporal specification in a way that, at the semantic level, we capture all the tainted objects. For an unbounded number of object, re-binding specification variables with different values during the computation helps the specification to be kept 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 using a list:

```
  while (l.next()!=null) {
    // here using the list l in some way...
    ......
}
```

If the list `l` is circular or pan-handle shaped, one of the problem could be that 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 have obtained the list by calling a third party library, we would want to check the following property:[2]

The shape of the list should be neither circular nor pan-handle. (4)

We will see in Section 6 that also the encoding of properties of this kind requires the ability to update the value of the specification variables. This is because the way to encode the property is by detecting, when using the list, that we reach an element we have seen before.

3 Overview and Background

Figure 2 shows the main concepts of concern. Register automata are well-studied devices, introduced decades ago. The gap between TOPL properties and register automata is filled by two semantic models, namely high-level and low-level TOPL automata. The high-level model (rollback TOPL automata) is better suited for implementation; the low-level model (simple TOPL automata) is easier to analyze.

The target of an arrow is at least as expressive as its source. All three automata models have the same expressivity. In general, decision problems on automata (such as membership, emptiness, language inclusion), are either decidable for all three models, or undecidable for all three models. However, decidable problems do not necessarily have the same complexity for the three models.

$$\begin{tikzcd}
\text{TOPL properties} \arrow[r] & \text{rollback TOPL automata} \arrow[d, leftrightarrow] \\
 & \text{simple TOPL automata} \arrow[d, leftrightarrow] \\
 & \text{register automata}
\end{tikzcd}$$

Figure 2: Main concepts

We review here register automata and several results about them. Together with the reductions presented later, these results imply similar results about TOPL automata.

[2]Of course, one can write a method (to call before entering the loop) which traverses the list and check whether it is circular or pan-handle. However, when the scope of the list is not clearly defined, these kind of explicit checks become cumbersome and error prone.

We begin by defining a generic automaton, of which both TOPL and register automata are instances. A *letter* ℓ is an n-tuple $(v_1, \ldots, v_n)$ of *values* from a possibly infinite set V; we denote the *alphabet* by $\Sigma = V^n$. A *store* s is an m-tuple $(u_1, \ldots, u_m)$ of values; we denote the set of stores by $S = V^m$. A *register* i is an integer that identifies a component of the store; the set of registers is $[m] = \{1, 2, \ldots, m\}$. A *guard* g is a formula in some logic interpreted over pairs of letters and stores; we write $(s, \ell) \models g$ to denote that 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 acting on stores, which has relational semantics; we denote the set of actions by $A = \Sigma \to (S \times S)$. Thus, an action applied to a letter $a(\ell)$ is a relation on stores. We will later specialize to specific and simple logics (for guards) and programming languages (for actions). A *label* λ is a pair (g, a) of a guard g and an action a; we denote the set of labels by $\Lambda = G \times A$.

Definition 1. An *automaton* $\mathcal{A}$ with m registers over n-tuples of values from the set V consists of

- a finite set Q of states
- an initial state $q_0 \in Q$
- an initial store $s_0 \in S$
- a set $F \subseteq Q$ of final states
- a finite transition relation $\delta \subseteq Q \times \Lambda \times Q$

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 the automaton is a subset of $X \times \Sigma \times X$. We write

$$x_1 \xrightarrow{\ell}_{\mathcal{A}} x_2$$

to mean that (x_1, ℓ, x_2) is in the configuration graph of $\mathcal{A}$. If the automaton is clear from the context, we omit the subscript.

Definition 2. The *configuration graph* of an automaton consists of exactly those configuration transitions

$$(q_1, s_1) \xrightarrow{\ell} (q_2, s_2)$$

for which there exists an automaton transition $(q_1, (g, a), q_2) \in \delta$ such that $(s_1, \ell) \models g$ and $(s_1, s_2) \in a(\ell)$.

An automaton is *nondeterministic* when its configuration graph contains two distinct transitions that have the same source x_1 and are labeled by the same letter ℓ:

$$x_1 \xrightarrow{\ell} x_2 \text{ and } x_1 \xrightarrow{\ell} x_3, \quad \text{with } x_2 \neq x_3$$

Otherwise, the automaton is *deterministic*.

The *language* $\mathcal{L}(\mathcal{A})$ of an automaton $\mathcal{A}$ is the set of words that label walks from the initial configuration to some final configuration:

$$\mathcal{L}(\mathcal{A}) = \{\, \ell_1 \ldots \ell_k \mid x_0 \text{ initial, } x_{i-1} \xrightarrow{\ell_i}_{\mathcal{A}} x_i \text{ for } i \in [k],\ x_k \text{ final} \,\}$$

Being a walk rather than a path, the configurations x_0, x_1, …, x_k need not be distinct.

Given a number m of registers, an arity n for letters, and a countable set V of values, the set $A = \Sigma \to (S \times S)$ of actions is uncountable. Therefore we must focus on a subset of A if we want our automata to be finitely representable. We will only consider actions that are sequences of assignments. The assignment $(\mathsf{set}\, i := j)$ is defined as follows:

$$\big((v_1, \ldots, v_m), (v'_1, \ldots, v'_m)\big) \in (\mathsf{set}\, i := j)\big((u_1, \ldots, u_n)\big)$$

if and only if $v'_k = v_k$ for all k, except that $v'_i = u_j$. The sequence operator ; is defined by saying that the relation $(a_1; a_2)(\ell)$ is the composition of the relations $a_1(\ell)$ and $a_2(\ell)$, for all ℓ. The identity action nop satisfies $\mathsf{nop}(\ell) = \{\, (s, s) \mid s \in S \,\}$ for all ℓ.

Register automata work with arity $n = 1$, that is $\Sigma = V$. In this case we write $(\mathsf{set}\, i)$ as a shorthand for $(\mathsf{set}\, i := 1)$. The syntax of guards for register automata is the following:

$$G_{\mathrm{RA}} \ ::= \ \mathsf{fresh} \mid \mathsf{eq}\, i$$

They are interpreted as follows:

$$\begin{aligned} \big((u_1, \ldots, u_m), v\big) &\models \mathsf{fresh} && \text{iff } u_i \neq v \text{ for all } i \\ \big((u_1, \ldots, u_m), v\big) &\models \mathsf{eq}\, i && \text{iff } u_i = v \end{aligned}$$

With these notations we can fix a definition for register automata.

Definition 3. A *register automaton* with m registers is an automaton over the alphabet $\Sigma = V$ whose labels are given by $\Lambda = \{\, (\mathsf{fresh}, \mathsf{set}\, i) \mid i \in [m] \,\} \cup \{\, (\mathsf{eq}\, i, \mathsf{nop}) \mid i \in [m] \,\}$.

There are several variants of definitions for register automata in the literature. Our variant corresponds directly to what Neven et al [24] call 1N-RA (for 'one-way non-deterministic register automata').

Example 1. The language $\{\, abc \mid a \neq c \text{ and } b \neq c \,\}$ is recognized by the following register automaton with 3 registers over the alphabet $\Sigma = V \cup \{\bot\}$:

- $Q = \{1, 2, 3, 4\}$ and $q_0 = 1$ and $s_0 = (\bot, \bot, \bot)$ and $F = \{4\}$
- the set of transitions is

$$\begin{aligned} \delta = \{ &\big(1, (\mathsf{fresh}, \mathsf{set}\, 1), 2\big),\ \big(2, (\mathsf{fresh}, \mathsf{set}\, 2), 3\big), \\ &\big(2, (\mathsf{eq}\, 1, \mathsf{nop}), 3\big), \big(3, (\mathsf{fresh}, \mathsf{set}\, 1), 4\big) \}. \end{aligned}$$

Observe that: (1) The two parallel transitions from 2 to 3 are needed because labels cannot have the shape $(\mathsf{true}, \mathsf{set}\, i)$; in fact, we have not even defined yet the guard true. (2) Only values from s_0 may appear multiple times in a store; in this case, only $\perp$. (3) Register 3 ensures that $\perp$ will not appear in the third position of words, which is a risk because we extended the alphabet. (4) The action of the last transition is superfluous, but required by the definition. ∎

Theorem 1. *The following properties of register automata are known:*

1. *The emptiness problem is coNP-complete [27].*
2. *The membership problem is NL-complete [24] and NP-complete [27].*
3. *The inclusion, equivalence, and universality problems are decidable for* 2 *registers [21, 24].*
4. *The inclusion, equivalence, and universality problems are undecidable in general [24].*
5. *Register automata are closed under union, intersection, concatenation, and Kleene star [21].*
6. *Register automata are* not *closed under complementation and reversal [21].*
7. *Nondeterministic register automata are strictly more expressive than deterministic register automata [21].*
8. *The language $\mathcal{L}(\mathcal{A}) \cap C^*$ is regular for any register automaton $\mathcal{A}$ and any finite subset $C \subset V$ of values [21].*

Some of these properties contrast with analogues for finite alphabets, which are better known. For example with finite alphabets automata are closed under complementation and reversal, and nondeterminism does not increase expressivity. These results differ for infinite alphabets because registers can be re-bound to different values an unbounded number of times. On the other hand, once the alphabet is 'sliced' to a finite subset, the resulting language is regular in the standard sense.

These results have analogues for TOPL automata. The details depend on what exactly the arrows from Figure 2 are. The gist will generally be a transformation of the following kind: From an automaton $\mathcal{A}$ in one model, construct an automaton $\mathcal{A}'$ in another model such that

$$\ell_1 \ldots \ell_k \in \mathcal{L}(\mathcal{A}) \iff f(\ell_1) \ldots f(\ell_k) \in \mathcal{L}(\mathcal{A}')$$

for some fixed f that is easily computable. The interesting ideas will be in how $\mathcal{A}'$ is built; f handles technicalities.

4 TOPL Automata

We now gradually move towards a model that is better suited for runtime verification of Java programs.

4.1 Simple TOPL Automata

TOPL automata work with the alphabet $\Sigma = V^n$ for some $n \in \mathbb{N}$, unlike register automata, which force $n = 1$. It is then natural to consider different guards and actions, which handle specific components of the letters. TOPL guards are the following:

$$G_{\mathrm{TOPL}} \ ::= \ \mathsf{eq}\, i = j \mid \mathsf{neq}\, i \neq j \mid \mathsf{true} \mid G_{\mathrm{TOPL}} \ \mathsf{and}\ G_{\mathrm{TOPL}}$$

If $n = 1$, we write $(\mathsf{eq}\, i)$ instead of $(\mathsf{eq}\, i = 1)$, and $(\mathsf{neq}\, i)$ instead of $(\mathsf{neq}\, i \neq 1)$. Recall that $(\mathsf{set}\, i)$ abbreviates $(\mathsf{set}\, i := 1)$. The guards are interpreted as follows:

$$\begin{aligned} \big((u_1,\ldots,u_m),(v_1,\ldots,v_n)\big) &\models \mathsf{eq}\, i = j && \text{iff } u_i = v_j \\ \big((u_1,\ldots,u_m),(v_1,\ldots,v_n)\big) &\models \mathsf{neq}\, i \neq j && \text{iff } u_i \neq v_j \end{aligned}$$

and

$$\begin{aligned} (s,\ell) &\models \mathsf{true} && \text{always} \\ (s,\ell) &\models g_1 \ \mathsf{and}\ g_2 && \text{iff } (s,\ell) \models g_1 \text{ and } (s,\ell) \models g_2 \end{aligned}$$

We can now define TOPL automata.

Definition 4. A *TOPL automaton* is an automaton whose guards are from the set G_{TOPL} (defined above) and whose actions are sequences of assignments (defined in Section 3).

Example 2. Consider again the language $\{\, abc \mid a \neq c \text{ and } b \neq c \,\}$. It is recognized by the following TOPL automaton with 2 registers over the alphabet $\Sigma = V$:

- $Q = \{1,2,3,4\}$ and $q_0 = 1$ and $F = \{4\}$
- δ consists of $\big(1, (\mathsf{true}, \mathsf{set}\, 1), 2\big)$ and $\big(2, (\mathsf{true}, \mathsf{set}\, 2), 3\big)$ and $\big(3, (\mathsf{neq}\, 1 \ \mathsf{and}\ \mathsf{neq}\, 2, \mathsf{nop}), 4\big)$.

Any initial store s_0 works. ∎

It is not difficult to see that TOPL automata are at least as expressive as register automata.

Lemma 1 (RA to TOPL)**.** *There exists an algorithm that, given a register automaton $\mathcal{A}$ over the alphabet V, with m registers, $|Q|$ states, and $|\delta|$ transitions, builds a TOPL automaton $\mathcal{A}'$ over the alphabet V, with m registers, $|Q|$ states, and $|\delta|$ transitions such that*

$$w \in \mathcal{L}(\mathcal{A}) \iff w \in \mathcal{L}(\mathcal{A}')$$

Proof. The guard fresh is encoded as the guard $\mathsf{neq}\, 1 \ \mathsf{and}\ \ldots \ \mathsf{and}\ \mathsf{neq}\, m$. All other guards and actions remain unchanged. □

It is relatively harder to establish that TOPL automata are at most as expressive as register automata.

Lemma 2 (TOPL to RA)**.** *There exists an algorithm that, given a TOPL automaton $\mathcal{A}$ over the alphabet V^n, with m registers, $|Q|$ states, and $|\delta|$ transitions, builds a register automaton $\mathcal{A}'$ over the alphabet V', with $m+1$ registers, $O(n\,m^{m+1}|Q|)$ states, and $O(n\,m^{m+2}|\delta|)$ transitions such that*

$$\ell_1 \ldots \ell_k \in \mathcal{L}(\mathcal{A}) \iff f(\ell_1)\ldots f(\ell_k) \in \mathcal{L}(\mathcal{A}')$$

where $f\big((v_1,\ldots,v_n)\big)$ is $v_1 \ldots v_n$ repeated $m+1$ times, and V' is V with up to $m-1$ extra dummy values.

Proof. We describe an encoding chain $\mathcal{A}_1 \to \mathcal{A}_2 \to \mathcal{A}_3 \to \mathcal{A}_4$ that goes from a TOPL automaton $\mathcal{A}_1$ to a register automaton $\mathcal{A}_4$. The encoding $\mathcal{A}_2 \to \mathcal{A}_3$ is more difficult than the other two.

Step $\mathcal{A}_1 \to \mathcal{A}_2$. The automaton $\mathcal{A}_2$ works over the alphabet V, while $\mathcal{A}_1$ may work on an alphabet V^n with $n > 1$. The number of registers is preserved. Each letter $(v_1,\ldots,v_n)$ is transformed into the word $v_1 \ldots v_n$ repeated $m+1$ times. Each transition $\big(q_1,(g,a),q_2\big)$ is encoded with a path from q_1 to q_2 of length $(m+1)n$. The first n transitions encode the guard g – they are labeled (g_1,nop), …, (g_n,nop). The guard g_j contains the conjunct $\mathsf{eq}\,i$ if and only if g contains the conjunct $\mathsf{eq}\,i = j$; similarly for neq. The following mn transitions encode the action a – they are labeled (true, a_1), …, (true, a_{mn}). If the last assignment of a to register i is $\mathsf{set}\,i := j$, then $a_{(i-1)m+j}$ is $\mathsf{set}\,i$. Otherwise, if a does not assign at all to register i, then $a_{(i-1)m+1}$ is nop.

The labels of $\mathcal{A}_2$ have one of the following shapes:

- $(\mathsf{eq}\,i_1$ and $\mathsf{eq}\,i_2 \ldots$ and $\mathsf{neq}\,i'_1$ and $\mathsf{neq}\,i'_2 \ldots,\ \mathsf{nop})$
- $(\mathsf{true},\ \mathsf{set}\,i)$
- $(\mathsf{true},\ \mathsf{nop})$

If automaton $\mathcal{A}_1$ has $|Q|$ states and $|\delta|$ transitions, then $\mathcal{A}_2$ has $O(mn|Q|)$ states and $O(mn|\delta|)$ transitions.

Step $\mathcal{A}_2 \to \mathcal{A}_3$. Again, the number of registers is preserved. For this step, the transformation of letters is the identity. The automaton $\mathcal{A}_3$ uses guards from G_{RA}, while $\mathcal{A}_2$ uses guards from G_{TOPL}. Observe that a register automaton never stores a value twice, because assignments are guarded by fresh. This is the main obstacle. The key idea of the encoding $\mathcal{A}_2 \to \mathcal{A}_3$ is the following:

> *Encode a function $r \in [m] \to [m]$ in the state itself such that $r(i)$ is the register of $\mathcal{A}_3$ that currently simulates the register i of $\mathcal{A}_2$. If two registers $i \neq j$ of $\mathcal{A}_2$ hold the same value, they are simulated by the same register $r(i) = r(j)$ of $\mathcal{A}_3$.*

Let us develop the idea in some detail. Let m and Q denote the count of registers and, respectively, the set of states of $\mathcal{A}_2$. The set of states of $\mathcal{A}_3$ is $Q \times ([m] \to [m])$. Each state q of $\mathcal{A}_2$ is encoded with m^m states (q, r) of $\mathcal{A}_3$. The configuration $\big((q, r), (u_1, \ldots, u_m)\big)$ of $\mathcal{A}_3$ simulates the configuration $\big(q, (v_1, \ldots, v_m)\big)$ of $\mathcal{A}_2$ with $v_i = u_{r(i)}$. Moreover, we ensure that no two registers of $\mathcal{A}_3$ hold the same value; that is, the stores of $\mathcal{A}_3$ are injective, $u_i \neq u_j$ if $i \neq j$. To ensure that the initial store of $\mathcal{A}_3$ is injective we may need to use up to $m - 1$ dummy values.

Each transition $\big(q_1, (g, a), q_2\big)$ of $\mathcal{A}_2$ is encoded into several transitions $\big((q_1, r), (g', a'), (q_2, r')\big)$ of $\mathcal{A}_3$. Here r takes all m^m possible values, and g', a', and r' are chosen depending on g, a, and r, as described below.

Consider a transition $\big(q_1, (\mathsf{eq}\, i_1 \text{ and } \mathsf{eq}\, i_2 \text{ and } \ldots \text{ and } \mathsf{neq}\, i'_1 \text{ and } \mathsf{neq}\, i'_2 \text{ and } \ldots, \mathsf{nop}), q_2\big)$. Let $I = \{\, i_1, i_2, \ldots \,\}$ and $I' = \{\, i'_1, i'_2, \ldots \,\}$. Note that transitions labeled by $(\mathsf{true}, \mathsf{nop})$ are a special case. We analyze separately the cases $I = \emptyset$ and $I \neq \emptyset$.

Case $I = \emptyset$. The guard requires the current letter ℓ to be different from the values in the set I' of registers of $\mathcal{A}_2$. So, ℓ must be different from the values held in the set $r(I')$ of registers of $\mathcal{A}_3$. Because the stores of $\mathcal{A}_3$ have no repeated value, ℓ may be either fresh, or equal to a register outside of the set $r(I')$. Thus, the encoding consists of several parallel transitions from (q_1, r) to (q_2, r). One is labeled $(\mathsf{fresh}, \mathsf{nop})$, and several are labeled $(\mathsf{eq}\, j, \mathsf{nop})$ for each $j \in [m] - r(I')$.

Case $I \neq \emptyset$. The guard also requires that the current letter equals the value in the set I of registers of $\mathcal{A}_2$. This cannot be true if $|r(I)| > 1$ or $r(I) \cap r(I') \neq \emptyset$: The encoding in this case is no transition. Otherwise, $r(I) = \{i\}$ for some i, and the encoding is a transition from (q_1, r) to (q_2, r) labeled by $(\mathsf{eq}\, i, \mathsf{nop})$.

Consider a transition $\big(q_1, (\mathsf{true}, \mathsf{set}\, i), q_2\big)$. Either the current letter is in some register or it is in none. To account for the first case, the encoding contains parallel transitions from (q_1, r) to (q_2, r'_j) labeled by $(\mathsf{eq}\, j, \mathsf{nop})$, for all $j \in [m]$. Here r'_j is the same as r except that $r'_j(i) = j$. To account for the second case, the encoding contains a transition from (q_1, r) to (q_2, r'_j) labeled by $(\mathsf{fresh}, \mathsf{set}\, j)$, for one j chosen as follows. If r is a permutation then $j = r(i)$; otherwise j is any element of $[m] - r([m])$.

If automaton $\mathcal{A}_2$ has $|Q|$ states and $|\delta|$ transitions, then $\mathcal{A}_3$ has $O(m^m |Q|)$ states and $O(m^{m+1} |\delta|)$ transitions.

Step $\mathcal{A}_3 \to \mathcal{A}_4$. Transitions labeled by $(\mathsf{fresh}, \mathsf{nop})$, which are produced by the encoding described so far, are not allowed in register automata. They can be handled by adding one register, without significantly increasing the number of transitions. First, each label $(\mathsf{fresh}, \mathsf{nop})$ is transformed into $(\mathsf{fresh}, \mathsf{set}\, m + 1)$. Second, for each transition labeled (fresh, a), we add a parallel transition labeled $(\mathsf{eq}\, m + 1, a)$. Again, the transformation of letters is the identity.

If the automaton $\mathcal{A}_3$ has m registers and $|\delta|$ transitions, then $\mathcal{A}_4$ has $m + 1$ registers and $O(|\delta|)$ transitions.

This completes the proof. $\square$

Remark 1. It follows from Lemma 1 and Lemma 2 that simple TOPL automata and register automata are equally expressive.

4.2 Automata with Rollback and Skip

TOPL automata are almost convenient for runtime verification of Java. The letters will be events like method calls and returns. Calls and returns happen, strictly speaking, at different moments. Moreover, some properties might explicitly mention other events that must happen between a call and its corresponding return. But, consider the constraint that iterators should advance only if `hasNext` returned `true`. The event '`hasNext` was called on the iterator that is being tracked' and the event '`hasNext` returned `true`' are distinct. Yet, it is convenient to pretend that two such consecutive events make one composed event. This motivates rollback, a mechanism by which transitions consume multiple consecutive letters.

Definition 5. A *rollback automaton* $\mathcal{A}$ consists of:

- a finite set Q of states; an initial state $q_0 \in Q$; an initial store $s_0 \in S$; a set $F \subseteq Q$ of final states
- a finite transition relation $\delta \subseteq Q \times \Lambda^* \times Q$

where Λ is a set of labels as defined in Section 3.

The difference between rollback automata and (simple) automata, according to definitions 1 and 5, is that rollback transitions have a list of labels $\bar{\lambda} = \lambda_1 \ldots \lambda_d$, while simple transitions have exactly one label λ. A transition with exactly one label is said to be a *unit transition*. If a rollback automaton has only unit transitions, then it is also a (simple) automaton.

The semantics of a rollback automaton cannot be described by the configuration graph from Definition 2, because that definition does not handle non-unit transitions. Instead, we define a rollback configuration graph. Perhaps surprisingly, in the case of simple automata, the rollback configuration graph does not coincide with the configuration graph. This discrepancy is explained by the different goals of the two models: automata without rollback are meant to be easy to analyze, while rollback automata are meant to be convenient for specifying properties of object-oriented programs. Often the desired behavior for a letter is to remain in the same configuration. Without rollback, this behavior is achieved by explicit loops in the automaton. The definition of the rollback configuration graph reduces the need to explicitly list such skip loops.

A *rollback configuration* is a pair (x, w) of a configuration x and a word w; we denote the set of rollback configurations by $Y = X \times \Sigma^*$. We think of w as yet to be processed. A rollback configuration is *initial* when its configuration is the initial configuration; that is, it has the shape $\big((q_0, s_0), w\big)$, where q_0 is the initial state of the automaton, and s_0 is the initial store of the automaton. A rollback configuration is *final* when its state is final and its word is the empty word; that is, it has the shape $\big((q, s), \epsilon\big)$, where q is one of the final states F of the automaton, and ϵ is the empty word. The rollback configuration graph is a subset of $Y \times \Sigma^* \times Y$. We write

$$y_1 \overset{w}{\hookrightarrow}_{\mathcal{A}} y_2$$

to mean that (y_1, w, y_2) is in the rollback configuration graph of $\mathcal{A}$. If the automaton is clear from the context, we omit the subscript.

The following concepts simplify the definition of the rollback configuration graph. The (simple) automaton

$$\mathcal{T}\big(s_0; (g_1, a_1), \ldots, (g_d, a_d)\big)$$

is defined to have states 0, 1, ..., d, out of which 0 is initial and d is final, transitions $\big(i-1, (g_i, a_i), i\big)$ for all $i \in [d]$, and initial store s_0. Recall that $\ell_1 \ldots \ell_k \in \mathcal{L}(\mathcal{A})$ when there exist (not necessarily distinct) configurations x_0, x_1, ..., x_k such that x_0 is the initial configuration, x_k is a final configuration, and

$$x_{i-1} \overset{\ell_i}{\hookrightarrow}_{\mathcal{A}} x_i \qquad \text{for } i \in [k]$$

When the store of x_k is s_k we say that $\mathcal{A}$ *can accept* $\ell_1 \ldots \ell_k$ *with store* s_k. A word may be accepted with multiple stores.

Definition 6. The *rollback configuration graph* of an automaton consists of two types of transitions:

1. *rollback transitions*

   $$\big((q_1, s_1), ww'\big) \overset{w}{\hookrightarrow} \big((q_2, s_2), w'\big)$$

   when there exists a transition $(q_1, \bar{\lambda}, q_2) \in \delta$ such that $\mathcal{T}(s_1; \bar{\lambda})$ can accept w with store s_2

2. *skip transitions*

   $$(x, \ell w) \overset{\ell}{\hookrightarrow} (x, w)$$

   when no rollback transition starts from $(x, \ell w)$

The *rollback language* $\mathcal{L}_\rho(\mathcal{A})$ of an automaton $\mathcal{A}$ is the set of words that label walks from some initial rollback configuration to some final rollback configuration:

$$\mathcal{L}_\rho(\mathcal{A}) = \{\, w_1 \ldots w_k \mid y_0 \text{ initial}, \; y_{i-1} \overset{w_i}{\hookrightarrow} y_i \text{ for } i \in [k], \; y_k \text{ final} \,\}$$

Since a simple automaton $\mathcal{A}$ is also a rollback automaton, is has both a simple language $\mathcal{L}(\mathcal{A})$ and a rollback language $\mathcal{L}_\rho(\mathcal{A})$. The treatment of skip transitions implies the aforementioned discrepancy, that the languages $\mathcal{L}(\mathcal{A})$ and $\mathcal{L}_\rho(\mathcal{A})$ are usually distinct.

Example 3. Consider the automaton consisting of one transition labeled with the guard $\mathsf{eq}\,1$, from the initial state to the final state. The alphabet is $\Sigma = V$ and the initial store has one register containing value v. The simple language of this automaton consists of one word made of one letter, namely v. The rollback language, on the other hand, consists of all words that contain the letter v. ∎

It is possible, however, to transform TOPL automata such that their rollback languages coincide with their simple languages.

Lemma 3 (TOPL to rTOPL)**.** *There exists an algorithm that, given a TOPL automaton $\mathcal{A}$ with $|Q|$ states, at most d transitions outgoing of each state, and guards with at most k conjuncts, builds a TOPL automaton $\mathcal{A}'$ with $|Q|+1$ states and at most $(d+k^d)|Q|$ transitions such that*

$$w \in \mathcal{L}(\mathcal{A}) \iff w \in \mathcal{L}_\rho(\mathcal{A}')$$

Proof. For the simple automaton $\mathcal{A}$ compare the simple configuration graph with the rollback configuration graph. Each simple transition $x_1 \to^\ell x_2$ corresponds to several rollback transitions $(x_1, \ell w) \hookrightarrow^\ell (x_2, w)$, for all w. The rollback graph, however, also has skip transitions $(x, \ell w) \hookrightarrow^\ell (x, w)$ for configurations x that have no outgoing rollback transitions. Thus, the simple language and the rollback language would agree if all simple configurations would have at least one outgoing simple transition.

We obtain $\mathcal{A}'$ from $\mathcal{A}$ by adding unit transitions that do not change the simple language, but ensure that all simple configurations have an outgoing transition. First we add a special stuck state q_{stuck}. Then, for each original state q, we list the guards $g_1, g_2, \ldots, g_d$ of the outgoing transitions. The configuration (q, s) has no outgoing transition for some ℓ when $(s, \ell) \not\models g_i$ for $i \in [k]$. So, we synthesize a guard g that holds exactly in this situation. Informally, we want to add a transition from q to q_{stuck} with the guard $g = \neg g_1 \wedge \ldots \wedge \neg g_d$ and the action nop. There is one small complication: G_{TOPL} does not have a negation operator. However, we can negate the simple guards eq and neq, we can use distributivity to put g in disjunctive normal form, and we can simulate disjunction by parallel transitions. Thus, if each g_i contains up to k simple conjuncts, we add at most k^d transitions from state q. □

The converse transformation is more difficult. There are many obvious ways to simulate a rollback TOPL automaton with a simple TOPL automaton, and many of them are wrong. Note, for example, that when the set V of values is countable, the simple configurations are countable, but the rollback configurations are uncountable. Rollback, which ensures that several letters occur atomically, and skip, which filters irrelevant letters, usually lead to intuitive behaviors, but not always, as the following example illustrates.

Example 4. Consider the following rollback TOPL automaton with 2 registers over the alphabet $\Sigma = V = \{A, B\}$:

- $Q = \{1, 2, 3\}$ and $q_0 = 1$ and $s_0 = (A, B)$ and $F = \{3\}$

- δ consists of $\big(1, \big[(\mathsf{eq}\,2, \mathsf{nop}), (\mathsf{eq}\,1, \mathsf{nop}), (\mathsf{eq}\,2, \mathsf{nop})\big], 2\big)$ and $\big(1, \big[(\mathsf{eq}\,1, \mathsf{nop})\big], 3\big)$

The rollback language of this is automaton consists of those words in which the first A is not surrounded by two Bs. For example it contains A, BA, BAA, and $BAAB$, but not BAB or $BBAB$. ∎

The key idea for encoding rollback TOPL automata into simple TOPL automata is the following:

> *It is sufficient to simulate rollback configurations with words shorter than the length d of the longest transition, because both rollback and skip transitions depend on $\leq d$ letters.*

If $|w| < d$, it should be possible to encode a rollback configuration $((q, s), w)$ using the finite memory of a simple TOPL automaton. To decide which transitions to take we have $|w|+1$ letters, the ones stored in the current configuration plus the current letter, and m registers to consider. If $|w|+1 < d$, then the current letter ℓ is simply stored in the current configuration, and we move to (the encoding of) $((q, s), w\ell)$. Eventually, d letters are available and all transitions outgoing from q can be evaluated. The original guards involve comparing letter components with register values. Now some of the letter components are themselves in registers, and TOPL automata do not have guards that compare registers. But, we can ensure that the stores are injective (as in the translation from TOPL automata to register automata) and then the information about which registers equal which registers is statically available. Actions simply translate to substitutions into latter guards on the same transition.

Lemma 4 (rTOPL to TOPL). *There exists an algorithm that, given a rollback TOPL automaton $\mathcal{A}_\rho$ over the alphabet V_ρ^n with m_ρ registers, $|Q_\rho|$ states, and $|\delta_\rho|$ transitions of length $\leq d$, builds a simple TOPL automaton $\mathcal{A}$ with $m = m_\rho + dn$ registers, $O(d(d{+}1)(m{+}1)^m|Q_\rho|)$ states, and $O(d(d{+}1)(m{+}1)^{(m+n)}|\delta_\rho|)$ transitions such that*

$$w \in \mathcal{L}_\rho(\mathcal{A}_\rho) \iff w\,\underbrace{\ell_\perp \ldots \ell_\perp}_{d+1\ times} \in \mathcal{L}(\mathcal{A})$$

where $\ell_\perp$ is an extra dummy letter.

Proof. The set of states of $\mathcal{A}$ is

$$Q = Q_\rho \times \{0, \ldots, d\} \times \{0, \ldots, d-1\} \times \big([m] \rightharpoonup [m]\big)$$

When the rollback configuration

$$\big(q_\rho, (v_1, \ldots, v_{m_\rho})\big), \ell_0 \ldots \ell_{h-1}$$

of $\mathcal{A}_\rho$ is reached, one of the simple configurations

$$(q_\rho, h, k, r),\ (u_1, \ldots, u_m)$$

of $\mathcal{A}$ is reached. Here h is the length of the remembered history, while k, r, u_1, $\ldots$, u_m vary subject to the following constraining invariants:

- the stores of $\mathcal{A}$ are injective: $u_i \neq u_j$ if $i \neq j$
- for $i \in [m_\rho]$, the value of register i of $\mathcal{A}_\rho$ is stored in register $r(i)$ of $\mathcal{A}$; that is, $u_{r(i)} = v_i$
- for $0 \le k' < h$ and $\ell_{k'} = (v'_1, \ldots, v'_n)$ and $1 \le i' \le n$, the i'th component $v'_{i'}$ of letter $\ell_{k'}$ is stored in register $r(\iota(k+k', i'))$ of $\mathcal{A}$, where $\iota(k'', i') = m_\rho + (k'' \bmod d)n + i'$; that is,
$$u_{r(\iota(k+k',i'))} = v'_{i'}$$
- r is undefined for register slots that are reserved for storing letters but are currently unused; namely, $r(\iota(k+k', i'))$ is undefined for all i' and k' such that $1 \le i' \le n$ and $h \le k' < d$

We consider in turn each state $q = (q_\rho, h, k, r)$. Its outgoing transitions are determined by the outgoing transitions of q_ρ.

Case $h < d$. In this case all outgoing transitions of q simply record the current letter $\ell = (v_1, \ldots, v_n)$. To maintain the injectivity of stores, only those components of ℓ that are not already in some register must be stored. We consider $(m+1)^n$ distinct situations: each of the n components may be in one the m registers, or it may be fresh. Such a situation is described by a function $p \in [n] \rightharpoonup [m]$. We add to $\mathcal{A}$ a transition
$$(q_\rho, h, k, r),\ (g_p, a_p),\ (q_\rho, h+1, k, r_p)$$
The guard g_p is constructed such that it ensures we are indeed in a situation described by p; the action a_p stores the fresh values of ℓ somewhere outside of $r([m])$; the function r_p records where the fresh values were stored and where the existing values already were.

The guard g_p is constructed as follows. If $p(j)$ is undefined, which means that v_j should be fresh, then g_p contains conjuncts $(\mathsf{neq}\, i \neq j)$ for $i \in [m]$. If $p(j)$ is defined, which means that $u_{p(j)} = v_j$, then g_p contains the conjunct $(\mathsf{eq}\, p(j) = j)$. These are all the conjuncts of g_p.

We now fix some injection σ from the set $\{v_j \mid p(j) \text{ undefined}\}$ of fresh values to the set $[m] - r([m])$ of unused registers. The action a_p contains an assignment $(\mathsf{set}\, \sigma(v_j) := j)$ for each j where $p(j)$ is undefined. Also
$$r_p(\iota(k', i')) = \begin{cases} p(i') & \text{if } k' = k+h \text{ and } p(i') \text{ defined} \\ \sigma(v_{i'}) & \text{if } k' = k+h \text{ and } p(i') \text{ undefined} \\ r(\iota(k', i')) & \text{otherwise} \end{cases}$$

Case $h = d$. At this point the values in the m registers of $\mathcal{A}$ are enough to decide whether to simulate a rollback or a skip transition of $\mathcal{A}_\rho$. (Note that we could use the current letter and decide one step earlier, when $h = d - 1$, which transitions to take. But, that alternative is slightly more complex.) First we encode the transitions outgoing from q_ρ,
$$(q_\rho, [(g_0, a_0), \ldots, (g_{d'-1}, a_{d'-1})], q'_\rho) \qquad \text{with } d' < d$$

and later we will encode the skip configuration transition.

Consider an assignment ($\mathsf{set}\, i := j$) that appears in $a_{k'}$. Suppose that the distribution of values just before this assignment is given by r'. This means that the register i of $\mathcal{A}_\rho$ is simulated by $r'(i)$, and that the jth component of letter $\ell_{k'}$ is simulated by $r'(\iota(k+k',j))$. After the assignment is executed, the distribution of values is given by

$$r''(i') = \begin{cases} r'(\iota(k+k',j)) & \text{if } i' = i \\ r'(i') & \text{otherwise} \end{cases}$$

Thus, it is possible to find the register distribution function after each of the d' steps. Let us write $r_{k'}$ for the distribution function just before step k'; in particular, $r_0 = r$, where r is given by the state q of $\mathcal{A}$. Suppose now that $g_{k'}$ contains the conjunct ($\mathsf{eq}\, i = j$). We can evaluate this conjunct statically by checking whether $r_{k'}(i) = r_{k'}(\iota(k+k',j))$. Similarly, we can evaluate all guards g_0, g_1, $\ldots$, $g_{d'-1}$. If some guard does not hold, then the encoding is nothing. If all guards hold, then we add outgoing transitions. These transitions must also save the current letter, so the whole process from *Case* $h < d$ is repeated. However, that process is applied as if the starting state has the value distribution function $r_{d'}$, rather than r; as if the history is $h - d'$, rather than h; and as if the reference point is $(k + d') \bmod d$, rather than k.

If the state $q = (q_\rho, d, k, r)$ still has no outgoing transitions at this point, it is because none of the outgoing transitions of q_ρ have guards that all hold. In this case, we must simulate a skip transition. This is done as in *Case* $h < d$, except we first simulate removing the letter to which k points. We do so by incrementing k and decrementing h and replacing the distribution function r by one that is undefined at $\iota(k, i')$ for $i' \in [n]$.

Finally, we add one final state, and transitions to it from the states with $h = d$. These transitions have a guard that requires all d saved letters to be $\ell_\perp$. □

Remark 2. It follows from Lemma 3 and Lemma 4 that rollback TOPL automata and simple TOPL automata are equally expressive.

4.3 Properties

The following are consequences of Theorem 1 and Lemmas 1, 2, 3, and 4.

Theorem 2. *Register automata, simple TOPL automata, and rollback 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 decidable for* 1 *register.*
3. *The language inclusion, the language equivalence, and the universality problems are undecidable in general.*

4. *The languages of these automata are closed under union, intersection, concatenation, and Kleene star.*

5. *The languages of these automata are not closed under complementation.*

5 TOPL Properties

In this section, we describe the user level language of TOPL properties, as well as the translation from TOPL properties to rollback TOPL automata in Figure 2. This language provides a user-friendly way to write down certain rollback TOPL automata. As such, TOPL properties form a subset of rollback TOPL automata, and the translation in Figure 2 is morally an injection. There is, however, in the interest of user-friendliness, some encoding happening.

Syntactically, a TOPL property has a set of transitions.[3] Each transition is a labelled arc (directed edge) with a source and a target vertex identified by their name. There must be a vertex named `start` and one named `error`. Labels look like method calls.

In the following, we will describe TOPL labels and how they translate to guards and actions. But first we will describe the events that a TOPL property considers and hence the alphabet that the corresponding rollback TOPL automaton works over.

5.1 The Alphabet

We model the semantics of programs and properties as sets of event traces. We say that a program *violates* a property when their sets of traces intersect. In other words, properties encode bad executions, rather than good executions.

The traces of events that a property P deems lead to a bad execution is thus the language of the corresponding automaton $\mathcal{A}_P$. Observable events for TOPL properties are method calls with parameter values and a time-tag. The method name is taken to be a string literal and an arity (for simplicity we do not distinguish overloaded methods with the same arity). The time-tag is either *call* or *ret*, corresponding to observing the method being called or returned from. If the time-tag is *ret*, the event carries the return value rather than any parameter values.

Thus, for $\mathcal{A}_P$, the values of interest come from two sources; event ids and parameter values: The set E of event ids is given by the grammar

$$\mathit{call}\ m^k \mid \mathit{return}\ m^k$$

where m^k is one of the method names mentioned in P (with arity k).

The set V_L of possible parameter values is the set of values from the programming language (e.g., Java) plus a dummy value $\perp$.

[3]For the complete BNF grammar for TOPL we refer to [18].

The values are then $V = E + V_L$, and the alphabet is V^{n+2}, where n is the maximal arity of a method mentioned in P. (We need one extra register for the return value and one for the event id.)

We are only concerned with the subset $E \times V_L^{n+1}$, ie the tuples where exactly the first component is an event id. If P has a maximal arity of 5, the event *call* $m^3(a, b, c)$ would be understood as $(\mathit{call}\ m^3, \perp, a, b, c, \perp, \perp)$ 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 fillers which allows all tuples to have the same lenght. The event *ret* $r = m^3$ would be understood as $(\mathit{ret}\ m^3, r, \perp, \perp, \perp, \perp, \perp)$ by $\mathcal{A}_P$.

We can now describe the labels of P and explain how they are transformed into guards and actions of $\mathcal{A}_P$.

5.2 Guards and Actions from Labels

Labels refer to the registers via *patterns*. To retain the flavour of a programming language, we call the registers of P *property variables*. For each property variable `v` there are 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. The negated lowercase pattern `!v` reads the value of the property variable `v` and only matches different values.

Further, every element of V acts as a pattern that matches only the value it denotes, and a wildcard * pattern matches any value.

A TOPL property is *well-formed* when it satisfies the following two conditions: *(i)* labels must contain uppercase value patterns at most once; *(ii)* any use of a lowercase patterns must be preceded by a use of the corresponding uppercase pattern on all paths from `start` (i.e., property variables must be written before being read). From now on we assume TOPL properties to be well-formed.

A label is of one of three forms:

$$l ::= \mathit{call}\ m^k(x_1, \ldots, x_k) \mid \mathit{ret}\ x := m^k \mid x := m^k(x_1, \ldots, x_k)$$

where x and $x_1, \ldots, x_k$ are all patterns. The first two forms describe observable events and will be translated into unit transitions. We describe that translation first and treat the last form at the end.

In order to match elements from V, there is an extra register in $\mathcal{A}_P$ for each unique element mentioned by P (note that this includes all the method names). This register contains that value in the initial state of $\mathcal{A}_P$ and is never overwritten.

For each pattern x which is not *, we define three functions: $\mathit{reg}(x) : \mathbb{N}$ gives the associated register, $\mathit{pred}(x) : \mathbb{N} \to G$ gives the corresponding predicate and,

$act(x) : \mathbb{N} \to A$ gives the action, i.e.

x	$pred(x)(j)$	$act(x)(j)$
$\mathtt{V}$	true	$\mathsf{set}\, reg(\mathtt{v}) := j$
$\mathtt{v}$	$\mathsf{eq}\, reg(\mathtt{v}) = j$	nop
$!\mathtt{v}$	$\mathsf{neq}\, reg(\mathtt{v}) = j$	nop
$a \in V$	$\mathsf{eq}\, reg(a) = j$	nop
$*$	true	nop

We can now interpret labels. For a label l, we define

$$[\![l]\!] = [([\![l]\!]_G, [\![l]\!]_A)]$$

where $[\![-]\!]_G$ is given by

l	$[\![l]\!]_G$
$call\ m^k(x_1, \ldots, x_k)$	$pred(m^k)(1)$
	$\wedge\, pred(x_1)(3) \wedge \ldots \wedge pred(x_k)(k+2)$
$ret\ x := m^k$	$pred(m^k)(1) \wedge \mathsf{eq}\, reg(x) = 2$

and $[\![-]\!]_A$ is given by

l	$[\![l]\!]_A$
$call\ m^k(x_1, \ldots, x_k)$	$act(x_1)(3); \ldots ; act(x_k)(k+2)$
$ret\ x := m^k$	$act(x)(2)$

If P is well-formed all the writes will be to different registers and thus commute.

Let us now consider the third label

$$x := m^k(x_1, \ldots, x_k)$$

Notice how the right hand side refers to call time, while the left hand side refers to return time. We decided that this label should mean that m^k was called with parameters matching $x_1, \ldots, x_k$ and returned a value matching x, *and no event was observed in the meantime*. This is because an intermediate call, for instance a recursive call, could disconnect the method call and the return value. This label makes use of the rollback feature and translates to a transition of depth two:

$$[\![x := m^k(x_1, \ldots, x_k)]\!] = [[\![call\ m^k(x_1, \ldots, x_k)]\!], [\![ret\ x := m^k]\!]]$$

Initial State. All that remains is to describe the initial state of $\mathcal{A}_P$. As mentioned, the registers which corresponds to values in V contain those values in the initial state. Note that the generated actions never overwrite these registers. Registers which correspond to property variables contain $\perp$. If P is well-formed, $\perp$ will never be read from those registers.

Directives. To ease the task of writing TOPL properties and to increase their usefulness, two directives were added to the language. A `prefix` directive for convenience when specifying method names and an `observe` directive for pre-filtering events. The prefix directive `prefix` p produces from every method name pattern m^k, an extra method pattern pm^k. It further produces, from any transition involving m^k, a similar transition involving pm^k. The prefix directive `observe` p has the effect that only method calls which matches p are considered part of the event trace.

6 Examples

In this section, we formalize some temporal properties in TOPL. We refer the reader to [18] for a more comprehensive exemplification of properties.

Iterators. Property (1) in Section 2 can be formalized as follows:

```
property InvalidateOtherIterators
  observe <java.util.{Collection,Iterator}.*>
  prefix <java.util.{Collection,Iterator}>
  start -> start: *
  start -> one:  X := C.iterator()
  one -> one:    *
  one -> two:    Y := c.iterator()
  two -> yBad:   x.remove()
  two -> xBad:   y.remove()
  yBad -> error: call y.*[*]
  xBad -> error: call x.*[*]
```

This property describes the interaction of three objects. The first line is an `observe` directive declaring that we are interested in all method calls (denoted by the wild card *) on instances of `java.util.Collection` and `java.util.Iterator` *and* all those that override them. Figure 3 shows a graphical representation of the property.

Heap Shape and Values Sensitive Properties The following is a TOPL encoding of the property (4) in Section 2. It reports an error if a list is cyclic or it has the pan-handle shape (i.e., there is a lasso at some point). Directives are omitted.

```
property ListNotCyclic
  start -> start: *
  start -> a: X := *.getList()
  a -> a:     X := x.next()
  a -> b:     Y := x.next()
  b -> b:     Y := y.next()
  b -> error: x := y.next()
  a -> error: x := x.next()
```

The idea is that this property will bind the property variable x with any possible object in the list, and the y with any possible successors (via the next field) of

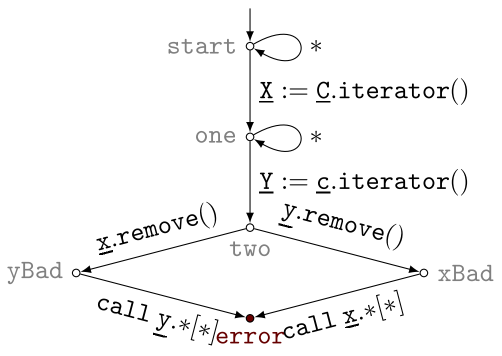

Figure 3: Property `InvalidateOtherIterators`

the current binding of x. Therefore, if there is a lasso, in the list, this will be detected when a new binding of y via a `b -> b` becomes equal to the binding of x. The transition `a ->error` detects the case where there is an object pointing to itself. This property shows the importance of re-binding for specification variables. This phenomenon is also shown by the following property which detects when a dictionary overwrite one of its bindings.

```
property BadDictionary
  message "dictionary overwrites its bindings"
  observe <Dictionary.*>
  prefix <Dictionary>
  start -> start: *
  start -> written:   D.put(K, V)
  written -> written: d.put(k, V)
  written -> error:   !v := d.get(k)
```

The overwrite is detected by the guard which checks if the value associated with a key k is the same as the original binding recorded in the property variable v.

Iterating Ropes Data Structures. The property (2) from Section 2 can be encoded in TOPL as follows:

```
property IterateRopes
  observe <Itr.next,Concat.make,Str.itr>
  prefix <Itr,Concat,Str>
  start -> start: *
  start -> a:  I := S.itr()
  a -> a: *, S := make(s, *), S := make(*, s)
  a -> b: J := s.itr()
  b -> c: i.set(*)
  c -> error: j.next()
  b -> d: j.set(*)
  d -> error: i.next()
```

The re-binding in the self-loop `a -> a` combined with `a->a:  *` ensure that iterators of sub-collections in the rope are tracked. Therefore, of two different iterators of two collections sharing some object invalidating ‘h the error state. ddare there other permutation missed by the property?

Property for Taint Info. To check taint information as described by property (3) in Section 2 we can use the following TOPL specification:

```
property TaintCheck
  observe <Itr.next,Concat.make,Str.itr>
  prefix <Itr,Concat,Str>
  start -> start: *
  start -> a:  X := input()
  a -> a: X := make(x, *), X := make(*, x), (!sani)(*)
  a -> b: sani(x)
  a -> error: sink(x)
```

The specification takes care to track any tainted substring x used in the rope and report an error if x reaches the sink method.

Singleton Pattern. The last property we present concerns the singleton pattern where the instantiation of a class is restricted to one object. It is used when exactly one object is needed to coordinate actions across the system. It can be enforced by the following.

```
property SingletonPattern
  start -> a:  I := getInstance()
  a -> error:  !i:= getInstance()
```

The property produce an error is the object obtained by the method `getInstance()` is not the same as the one already bound to the automata variable i.

7 Implementation

We have implemented the formal framework in a tool[4] for checking whether a Java program violates TOPL properties. The tool consists of two parts: a compiler and a checker. The TOPL compiler (`toplc`) instruments Java bytecode; The TOPL checker (class `topl.Checker`) monitors the execution and reports violations. Figure 4 depicts how the compiler and the checker fit together, thus summarizing our runtime checking system.

The TOPL Checker. The checker logs the property violations it detects. Internally it maintains a set of active states. New states are produced by applying an action to an existing state (called the parent). For error reporting, each state keeps track of its parent. The implementation takes advantage of this fact by representing stores using persistent sets for bindings; more precisely, we use treaps [3].

As states point to their parent, we need some mean to control overhead. This is done by a configurable parameter for the history length h in `Property.java`.

[4] `http://rgrig.github.com/topl`

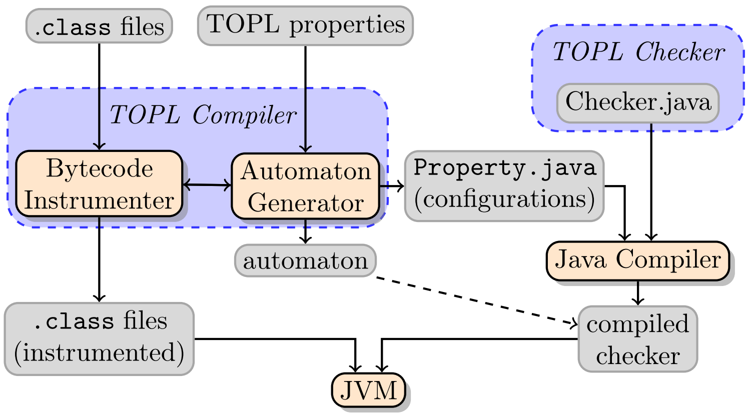

Figure 4: Architecture of the TOPL tool

We should then allow the Java VM to garbage collect states that are not reachable in h steps from some active state. To do that, after each execution of a transition we could do a BFS traversal with h steps and set to `null` the parent pointer of fringe states. However, for n active states, the traversal would take $O(nh)$ time. The procedure for executing transitions takes $O(nd)$ time, where d is the maximum out-degree of the property. So, the checker would become much slower if $h > d$, which usually is the case. One solution could be to perform the BFS once in every h/d steps. The solution we implemented tries to better address the average case. We remember the number m of states seen by a BFS, and perform the next BFS after $\sim m$ new states were created.

The set of active states in nondeterministic automata can be unbounded. Therefore, as for the case of trace lengths, we impose a bound on the number of active states. The bound can be configured by the user in the `Property.java` file. The strategy for forgetting states can also be chosen among: *random*, *newest*, and *oldest*. Newest/oldest refer to a time attached to states. The time of a state is the number of transitions taken to reach that state.

The TOPL Compiler. The compiler takes as input a Java project and a set of TOPL properties. The Java project is defined to be all files in a given directory, recursively descending into jar and zip files as well as subdirectories. The compiler produces (1) a copy of the Java project with the class files instrumented, and (2) a TOPL automaton. Only the Java methods relevant to the given TOPL properties are instrumented, but taking into consideration the inheritance tree of project. For example, suppose that a TOPL property mentions the method `next` from the interface `Iterator`. Then all the methods that implement `Iterator.next` are instrumented and matched by the automaton. More in details the compiler execute five phases. First, TOPL properties are converted into a simpler, intermediate format. Second, the compiler *statici-*

cally checks that the property is well-formed (e.g., variables writes must precede reads). We plan to also statically check whether the automaton can never fail. Third, all properties are merged into one automaton that is dumped into the file `Property.text`. Fourth, the compiler generates the checker configurations in the file `Property.java`. This file contains info used by the TOPL property and code configuring the checker. Configuration parameters include:

- *History length* – how many events to remember. A high value reports more events that led to `error`. A low value reduces the space and time overhead.
- *Maximum number of active states*. A high value increases the chances of detecting violations at the cost of higher overhead.
- *Collect call stacks* is a flag for including/excluding call stacks in violation reports.
- *Approximantion method* gives different ways in which the history informations are forgotten when the max amount of info allowed is reached.

Fifth, the compiler instruments methods' bytecode.[5]

8 Experimental Results

We measure the overhead on the test suite DaCapo [10], version 9.12. DaCapo is a collection of automated tests that exercise large portions of code from open-source projects and from the Java standard libraries. DaCapo itself was used for many experiments by the research community. Hence, we do not expect to find any bugs, but aim to measure 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, all of which are included in DaCapo. The latter properties are more complex and interesting.

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. So, we only report time overhead, in Tables 1 and 2. All experiments are performed on an Intel i5 machine with 4 cores at 3.33 GHz with 4 GiB of memory and running Linux 2.6.32 and Java VM 1.6.0_20.

8.1 JDK Properties

These properties are taken from the JavaMOP project [22]. *HasNext* was briefly mentioned in Section 4. It checks that no iterator is advanced without first calling `hasNext` and have it return `true`. *UnsafeIterator* checks that no iterator is advanced after the iterated collection has been modified. *UnsafeMapIterator*

[5]To instrument Java bytecode we use a fork of the library Barista [13].

	original	HasNext		UnsafeIterator		UnsafeMapIterator		UnsafeFileWriter	
		st=3	st=10	st=3	st=10	st=3	st=10	st=3	st=10
avrora	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	15.6	22.4	12.2X	27.0	149.9X	27.56	12.8X	21.6	23.4
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
jython	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 run on the DaCapo benchmarks (run in convergence mode). Times are in seconds.

checks that no iterator on the keys or values of a map is advanced after the map has been updated. *UnsafeFileWriter* checks that no file is written to after it has been closed.

Table 1 presents the running times for DaCapo's projects with and without instrumentation of Java's standard library. For each project there are two instrumented runs, one with a maximum of 3 active configurations and one with a maximum of 10 active configurations. For properties involving only one object, such as `HasNext`, a limit of 10 means that at most that many objects are tracked *at the same time*.

8.2 DaCapo Properties

We focus on 3 projects that are still widely developed and used: Tomcat, H2, and PMD.

All three benchmarks are large open-source projects with many users and active developers. The versions were current in 2009 when DaCapo-9.12 was released.

Tomcat 6.0.20. Tomcat is a highly concurrent servlet server. Servlets are Java programs running in a webserver, extracting data from `ServletRequests` , and sending data to `ServletResponse`s. A response has two associated incoming channels: a stream and a writer. Both should not be used concurrently. A response could be forwarded once the servlet sent data to it.

But the servlet, before forwarding the response, must call `flush` on the stream, on ther writer, or on the response itself. This is one of two properties

		Number of tracked active states		
	reference	≤ 0	$\leq 10^1$	$\leq 10^2$
tomcat	5.3±0.1	5.4±0.1	5.6±0.2	9.0±0.3
pmd	5.2±0.4	5.4±0.2	12.2±0.3	47.7±10.7
h2	6.6±0.2	9.5±0.2	130.1±12.2	timeout

Table 2: Experimental Results. Times are in seconds, averaged over 10 runs.

we checked. The other is given an anecdotal description at the end of the section.

H2 1.2.121. H2 is a database server. We checked four properties for it. Three properties express constraints on the order in which the methods of an interface may be called. For example, a client should not attempt to ask for a row from a cursor that did not advance; that is, "*for the interface* `Cursor`*, the method* `next` *must be called before calling method* `getSearchRow`". The fourth property describes the contract of the class `SysConstants`. This class reads environment variables when it is loaded. Its comments say that *"it is legal for a program to set these environment variables using the method* `System.setProperty`*, provided that this is done before the Java VM loads the class* `SysConstants`*"*. It is easy to specify this property because TOPL intentionally treats the special methods `<init>` and `<clinit>` as if they are normal methods.

PMD 4.2.5. PMD looks for bugs, dead code, and a few other problems in Java code. We checked five properties for it. One of the five involves two objects: *"A scope should be asked what is the definition of a name only if it already replied that it knows the name"*.

Table 2 summarizes the experimental results. The reference column shows the running time without any instrumentation. A timeout signifies a time greater than the reference time ×100. All reported times are averaged over 10 runs, and are accompanied by the unbiased estimate of standard deviation. In theory the set of active states is unbounded, but we impose a size limit on it. The table reports the running times for the instrumented bytecode when the size limit is 0, 10^1, 10^2. For h2 the running time is more than ×1.4 the reference time even when the TOPL checker is not tracking any active state. This indicates that we instrumented a simple method that runs often.

A Bug's Death. The interface `ServletResponse` from Tomcat contains methods `getWriter` and `getOutputStream`. The documentation of `getWriter` states that "either this method or `getOutputStream` may be called to write the body, not both". We initially interpreted this comment as follows.

> *It is illegal to call both the method* `getWriter` *and the method* `getOutputStream` *on the same instance of* `ServletResponse`*.* (5)

The TOPL checker reported many violations of (5). We were not too surprised because we were unsure of the intended meaning of the comment. We were surprised, however, that sometimes a `null` dereference occurred in DaCapo itself. We investigated. The statement that threw a `null`-pointer exception was the following:

```
if (log != null) log.write(b);                    (6)
```

It must be that a concurrent thread sets `log` to `null`. The statement is located in the method `write` of the class `TeeOutputStream`. The only method that sets `log` to `null` is `closeLog`, from the same class. So, we wrote a TOPL property saying that *executions of* `write` *and* `closeLog` *must not overlap in time*, and we asked for error reports that include call stacks. Indeed, DaCapo's main loop sometimes calls `closeLog` while the Tomcat server is printing.

TOPL helped us find a concurrency bug in the infrastructure of DaCapo. First, the bytecode instrumented by TOPL helped us notice the problem. The reason is *not* that a certain interleaving became more likely. Bug 2934521 of DaCapo mentions that exceptions of the Tomcat benchmark are silently ignored. This is why the `null`-dereference was not noticed before. The TOPL checker tried to report an unrelated issue to `System.err`. DaCapo had redirected the standard error stream to its own class, which faulted. TOPL caught the error and reported it as an internal issue, because no exceptions should be thrown while running the TOPL checker. Second, TOPL helped us identify the data race. It would have been more difficult to figure out which threads exactly used the stream simultaneously.

We then tried a weaker version of property (5):

Stream and writer of a response cannot both be used. (7)

This property[6] involves three related objects. It turns out this property is also violated. After reading the code to which the error traces pointed at, we noticed that the comments of some implementing classes weaken the contract: They say that the stream and the response cannot be used without an intervening `reset`. In this case the comments on the interface `ServletResponse` are arguably wrong, rather than the code. This brings out another advantage of TOPL: Since TOPL properties are processed by tools, they are less likely to become out-of-sync with the code.

9 Related Work

TOPL automata are a variant of *register automata* [21]. We proved that several decidability results known for register automata also hold for TOPL automata. Register automata work with infinite alphabets, as do a few other types of automata [28]. Until now, such automata found applications mostly related to databases and XML. Intuitively, however, such models are a good fit for expressing properties of programs because the set of program values is infinite.

[6]`http://goo.gl/V008y`

For example objects could be instantiated inside non-terminating loops. Our paper exhibits a working runtime verification system based on register automata, thus validating the intuition.

QVM [4] is a system for runtime monitoring of applications that are deployed. As such, QVM's main goal is efficiency, and it achieves it because (1) it is carefully implemented inside a Java virtual machine, (2) it checks properties involving a single object, (3) it adapts on-the-fly the rate of sampling objects to monitor, such that the overhead remains within a given budget. On the other hand, TOPL aims to help during development and testing. Thus, having a precise and expressive language for specifying temporal properties takes precedence over efficiency. For example TOPL can express properties involving many objects, even if monitoring such properties typically imposes a bigger overhead. Both QVM and TOPL let the programmer tune the overhead/coverage balance. In QVM's case the programmer directly chooses the percentage of time spent monitoring, which is intuitive; in TOPL's case the programmer chooses a more abstract parameter, the number of active configurations, which correlates with the overhead.

JavaMOP [22] and *tracematches* [2] are the main runtime verification systems that handle multiple objects. They differ in many details, but the essential ideas fit a common intuition. To communicate this intuition we depart from the terminologies of [22] and [2]. The alphabet is $\Sigma = C \times V$, where C is a finite set of constants and V is an infinite set of values. The task is to build a recognizer $\Sigma^* \to \mathbb{B}$. Both approaches build upon a basic recognizer $C^* \to \mathbb{B}$ over finite alphabets. Tracematches specify this recognizer by regular expressions, while JavaMOP supports several other logics. Both approaches make further assumptions on the structure of V, and then define a way to specify a family $P \subset \Sigma \to \mathbb{B}$ of predicates over letters. Each such predicate $p \in P$ determines a slice $\tau_p \in C^*$ of the trace $\tau \in \Sigma^*$ by retaining exactly those constants from letters satisfying p. The trace τ is recognized when any of its slices satisfy the basic recognizer. Within such a framework, it seems difficult to express some of our examples, which naturally use an unbounded number of assignments to registers.

Any algorithm that analyzes or transforms JavaMOP properties must be specific to a certain plugin because the formalization of the framework [26] imposes no constraint on recognizers $C^* \to \mathbb{B}$. In contrast, any algorithmic result discovered for register automata implies one for TOPL properties. For example we know that the emptiness problem is decidable, and we know that it is impossible to compute the negation of a property.

Tracematches [2] are likely to have the same expressivity as JavaMOP with the plugin for regular expressions. But, we are unaware of a fully formal comparison. Tracematches evolved from the work on aspect-oriented programming. Unsurprisingly, the user may specify arbitrary code to run when a match is found. The formal semantics ignore this arbitrary code and how it could influence further matches.

ConSpec [1] is a language used to describe security policies. Because ConSpec automatons are deterministic and have only a countable number of states, they cannot in principle express the property `InvalidateOtherIterators` (see Figure 3).

From the techniques used mostly for static verification of object-oriented programs, *typestates* [29] are probably most similar to TOPL. A modular static verification method for typestate protocols is introduced in [9]. The specification method is based on linear logic and relations among objects in the protocol are monitored by a tailored system of permissions. The specification of the interactions among objects by means of permissions adds an extra level of machinery which increases the gap between the intuitive protocol description and its formalization. Similarly [14, 8] provide a mean to specify typestate properties that belong to a single object. The specified properties are reminiscent of contracts or pre/post-conditions for methods and can deal with inheritance. In [17] the authors present sound verification techniques for typestate properties of Java programs. Their approach is divided in several stages with different verifiers varying for cost and precision. In the early stages efficient but imprecise analyses are employed whereas more expensive and precise techniques are then progressively employed in later stages. Every stage focuses on verifying only the parts of the code that previous stages failed to verify. It is likely that TOPL could be fruitfully combined with their analysis technique.

A specification language for interface checking aimed at C programs (called *SLIC*) is introduced in [6]. Differences between SLIC and TOPL include: the use (in SLIC) of non-determinism to encode universal quantification of dynamically allocated data, and the ability to have complex code in the automaton transitions. TOPL specifications naturally express universally quantified properties over data structures and for computability reasons, we have chosen to limit the actions performed during automaton transitions. Simple SLIC specifications are verified by the SLAM verifier [5]. While SLAM specialises on device drivers and checks client conformance rather than full protocols, very general specifications of object-oriented program behaviour can be given in JML [20] and Spec$^\sharp$ [7]. However the latter two languages focus on class specifications and do not have temporal features.

Finally, there are similar investigations going on in the functional programming community. In [15] contracts are used to express legal traces of programs in a functional language with references. The contracts specify traces as regular expressions over calls and returns, and hence, they resemble our automata, in a quite different setting. Here, the specifications are function-centered, though, and again, capturing inter-object relations seems somewhat tricky.

10 Conclusions and Future Work

In this paper we have introduced TOPL a new runtime verification system aimed at checking temporal safety properties of object-oriented programs. For TOPL we have defined solid formal foundations based on register automata. Moreover we have studied the formal correspondence between TOPL automata and register automata proving that they are equally expressive. TOPL has been implemented in a practical tool that we have applied to a variety of programs including large open source Java projects.

In the future, we intend to combine TOPL with separation logic [25] in order to deal with the heavy use of the heap and aliasing in object-oriented software. Moreover, we aim at developing static analysis techniques for TOPL properties using the jStar framework [16]. This will require investigating suitable abstraction for obtaining meaningful and precise over-approximations of the state space of the programs. Finally, we intend to develop a tailored bi-abduction inference technology [12] which would help with scalability of the analysis.

References

[1] I. Aktug and K. Naliuka. ConSpec — a formal language for policy specification. *Electr. Notes Theor. Comput. Sci.*, 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 R. E. Johnson and R. P. Gabriel, editors, *OOPSLA*, pages 345–364. ACM, 2005.

[3] C. R. Aragon and R. Seidel. Randomized search trees. In *FOCS*, pages 540–545. IEEE Computer Society, 1989.

[4] M. Arnold, M. Vechev, and E. Yahav. Qvm: an efficient runtime for detecting defects in deployed systems. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA '08, pages 143–162, New York, NY, USA, 2008. ACM.

[5] T. Ball and S. K. Rajamani. The SLAM toolkit. In G. Berry, H. Comon, and A. Finkel, editors, *CAV*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264. Springer, 2001.

[6] T. Ball and S. K. Rajamani. Slic: a specification language for interface checking (of C). Technical Report MSR-TR-2001-21, Microsoft Research, 2002.

[7] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[8] K. Bierhoff and J. Aldrich. Lightweight object specification with typestates. In M. Wermelinger and H. Gall, editors, *ESEC/SIGSOFT FSE*, pages 217–226. ACM, 2005.

[9] K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr., editors, *OOPSLA*, pages 301–320. ACM, 2007.

[10] 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 P. L. Tarr and W. R. Cook, editors, *OOPSLA*, pages 169–190. ACM, 2006.

[11] H.-J. Boehm, R. R. Atkinson, and M. F. Plass. Ropes: An alternative to strings. *Softw., Pract. Exper.*, 25(12):1315–1330, 1995.

[12] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In Z. Shao and B. C. Pierce, editors, *POPL*, pages 289–300. ACM, 2009.

[13] X. Clerc. Barista. `http://barista.x9c.fr/`.

[14] R. DeLine and M. Fähndrich. Typestates for objects. In M. Odersky, editor, *ECOOP*, volume 3086 of *Lecture Notes in Computer Science*, pages 465–490. Springer, 2004.

[15] T. Disney, C. Flanagan, and J. McCarthy. Temporal higher-order contracts. In *ICFP*, 2011.

[16] D. Distefano and M. J. Parkinson. jStar: towards practical verification for Java. In Harris [19], pages 213–226.

[17] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. In L. L. Pollock and M. Pezzè, editors, *ISSTA*, pages 133–144. ACM, 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] G. E. Harris, editor. *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*. ACM, 2008.

[20] `http://www.eecs.ucf.edu/~leavens/JML`.

[21] M. Kaminski and N. Francez. Finite-memory automata (extended abstract). In *FOCS*, pages 683–688. IEEE Computer Society, 1990.

[22] 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.

[23] N. A. Naeem and O. Lhoták. Typestate-like analysis of multiple interacting objects. In Harris [19], pages 347–366.

[24] F. Neven, T. Schwentick, and V. Vianu. Towards regular languages over infinite alphabets. In J. Sgall, A. Pultr, and P. Kolman, editors, *MFCS*, volume 2136 of *Lecture Notes in Computer Science*, pages 560–572. Springer, 2001.

[25] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

[26] G. Rosu and F. Chen. Semantics and algorithms for parametric monitoring. *Logical Methods in Computer Science*, 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 Z. Ésik, editor, *CSL*, volume 4207 of *Lecture Notes in Computer Science*, pages 41–57. Springer, 2006.

[29] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986.