TOPL: A Language for Specifying Safety Temporal Properties of Object-Oriented Programs

Radu Grigore Rasmus Lerchedahl Petersen Dino Distefano
Queen Mary, University of London
{rgrig,rusmus,ddino}@eecs.qmul.ac.uk

Abstract

In this paper we present ongoing work related to a new specification language for temporal safety properties aimed at object-oriented software. The language naturally captures relationships between objects and it is designed with the goal of performing dynamic and static analysis. We present its formal semantics as well as several examples showing its expressivity.

Categories and Subject Descriptors D.2.1 [*Software Engineering*]: Requirements/Specifications

General Terms Languages, Verification

Keywords Safety, Temporal Properties, Object-Oriented

1. Introduction

The verification community showed interest in *temporal safety properties* for a long time. Manna and Pnueli [24] provide a theoretical foundation and clearly argue why such properties are crucial. In this article, we focus on an object-oriented setting. The long term aim of our project is the automatic verification of temporal safety properties for Java programs of realistic size. To achieve our aim we need

- a language to formally specify temporal safety properties, and
- tools that automatically analyze Java code, dynamically and statically, with respect to given temporal safety properties.

This paper addresses the first point by introducing TOPL (**t**emporal **o**bject-oriented **p**roperty **l**anguage, pronounced like 'topple'). The development of tools remains future work.

We draw inspiration from existing specification languages, each of which we found not entirely suitable for our goal. Bierhoff and Aldrich [8] as well as Naeem and Lhoták [25] use specification languages inspired by typestates [29]. Specifically, Bierhoff and Aldrich [8] use a combination of linear logic [18] and access permissions, while Naeem and Lhoták [25] use tracematches. Disney et al. [14] use a language based on regular grammars to specify higher-order temporal contracts. Finally, Ball and Rajamani [5] essentially use nondeterministic aspect-oriented programming.

TOPL has the following characteristics:

1. It expresses easily relationships between several objects.
2. It is very high-level and similar to diagrams used in informal explanations.
3. It has a well-defined formal semantics in terms of a specific type of automatons.
4. It is designed to be used in program analysis (both static and dynamic).

The ability to express relations of several objects makes TOPL quite expressive. For example, for 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 and cannot be used later.

It is apparent that the formalization of the above constraint needs to keep track of several objects (at least two iterators, and one collection) and their interaction. Most other techniques aim at decomposing properties involving several objects into specifications that reflect the point of view of a single object. In contrast, TOPL does not try to achieve such decomposition. Parkinson [26] argues that invariants involving several objects are sometimes better than one-object invariants. Similarly, we believe that temporal properties that naturally involve several objects are easier to reason about if they are *not* decomposed.

Because TOPL is a high-level language with formal semantics (points 2 and 3 above), it reduces the semantic gap between the programmers' intuition of various temporal constraints on their code and the precise formal description needed by verification tools for automatic checking of these constraints.

In this paper we focus on TOPL's formal semantics and on TOPL's expressivity.

The paper is organized as follows. In Section 2 we start with motivating examples. Section 3 gives the syntax of TOPL and Section 4 introduces its semantics. Section 5 discusses related work. Finally, Section 6 concludes the paper and describes our plans for future work.

2. Examples

The first example (Section 2.1) uses a large part of TOPL. The other examples (Sections 2.2–2.5) illustrate TOPL's expressiveness.

2.1 Iterators Step by Step

The last statement in Figure 1 throws an exception. There are two iterators on the same collection, one of them modifies the collection, and this invalidates the other iterator. Such properties are often explained using diagrams [8–10, 16, 17, 25]. The diagrams sometimes have semantics, but are not expressive enough [16, 17]; the diagrams are sometimes expressive, but do not have formal

FOOL'11 October 23, 2011, Portland, Oregon, USA

```java
import java.util.*;
public class IncorrectIteratorUse {
  public static void main(String[] args) {
    List<Integer> c = new ArrayList<Integer>();
    c.add(1); c.add(2);
    Iterator<Integer> i = c.iterator();
    Iterator<Integer> j = c.iterator();
    i.next(); i.remove(); j.next();
  }
}
```

Figure 1. A first example: Java code

```
prefix <java.util.{Collection,Iterator}>
```

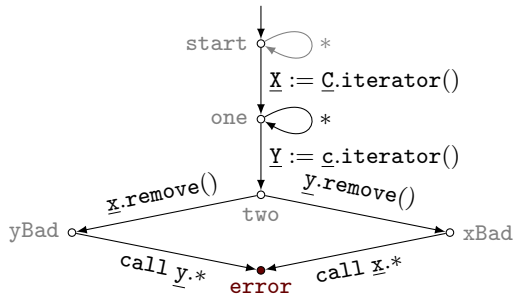

Figure 2. A first example: Diagram of safety property

```
property IteratorComodification
  prefix <java.util.{Collection,Iterator}>
  start -> start:* // implicit by convention
  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.*
```

Figure 3. A first example: TOPL property

semantics [8–10, 25]. Figure 2, on the other hand, captures a temporal property involving three interacting objects and has formal semantics. Figure 3 describes the same property in (the textual form of) TOPL, and is clearly isomorphic to the diagram of Figure 2. The rest of this section explains the semantics of Figure 2 and Figure 3: *Modifying a collection through an iterator invalidates other iterators for the same collection.*

Vertices have identifiers (`start`, `one`, `two`, ...); transitions have labels (`I := C.iterator()`, ...). There are two special vertices: `start`, from where the execution begins; and `error`, where the execution ends. Labels capture, roughly, the shape of statements that enable the corresponding transition.

Figure 4 shows an execution of the program and of an automaton for the property `IteratorComodification`. The automaton is nondeterministic. The lines {in curly brackets} describe the state of the automaton; the lines in `monotype` show the statements that execute; the other lines are comments. At a given moment, the automaton has a set of active states. A state is a pair of a vertex and a store. The store is a memory that holds automaton variables. Technically, it is a finite partial map from variables to values. (A partial finite map is sometimes called a *dictionary*.)

{ (`start`, []) }
`Iterator<Integer> i = c.iterator();`
// assume `c` $== \gamma$ and `i` $== \alpha$
{ (`start`, []),
  (`one`, $[c : \gamma, x : \alpha]$) }
`Iterator<Integer> j = c.iterator();`
// assume `j` $== \beta$
{ (`start`, []),
  (`one`, $[c : \gamma, x : \alpha]$),
  (`one`, $[c : \gamma, x : \beta]$),
  (`two`, $[c : \gamma, x : \alpha, y : \beta]$) }
`i.next();`
{ (`start`, []),
  (`one`, $[c : \gamma, x : \alpha]$),
  (`one`, $[c : \gamma, x : \beta]$),
  (`two`, $[c : \gamma, x : \alpha, y : \beta]$) }
`i.remove();`
{ (`start`, []),
  (`one`, $[c : \gamma, x : \alpha]$),
  (`one`, $[c : \gamma, x : \beta]$),
  (`yBad`, $[c : \gamma, x : \alpha, y : \beta]$) }
`j.next()`
{ (`start`, []),
  (`one`, $[c : \gamma, x : \alpha]$),
  (`one`, $[c : \gamma, x : \beta]$),
  (`error`, $[c : \gamma, x : \alpha, y : \beta]$) }

Figure 4. A first example: Running step by step

Notation 1. We write $[k_1 : v_1, k_2 : v_2]$ for the finite partial map that maps key k_1 to value v_1, and key k_2 to value v_2. The empty map is denoted by [].

The automaton has variables x, y, and c. At vertex `one` the variables x and c are initialized, and the variable y is not initialized; at vertex `two` all the variables x, y, and c are initialized. Being at vertex `one` means that x is an iterator for c; being at vertex `two` means that x and y are two iterators for the same collection c. There is a program variable `c` and an automaton variable c. The same name was chosen because the two variables always hold the same value in this example. In general, however, program variables and automaton variables live in different namespaces, and may hold different values.

Notation 2. Program variables are typeset in `monotype` (`c`, `i`, `j`); automaton variables are typeset in *italics* (c, x, y). Program variables appear in the program; automaton variables do *not* appear in the property. Instead, automaton variable *patterns* appear in the property, and they are typeset in `underlined monotype` (`c`, `C`, `x`, `X`, `y`, `Y`). We say *uppercase pattern* for an automaton variable pattern that starts with an uppercase letter (`C`, `X`, `Y`); we say *lowercase pattern* for an automaton variable pattern that starts with a lowercase letter (`c`, `x`, `y`).

As it will become apparent below, uppercase patterns write to the automaton store, while lowercase patterns read from the automaton store.

We now execute the program and the automaton step by step in parallel. Each step introduces a few new concepts.

Step 1. Initially, only the state $(\texttt{start}, [])$ is active. The outgoing transition of vertex `start` is labeled by

$$\underline{\texttt{X}} := \underline{\texttt{C}}.\texttt{iterator}()$$

and the first executed statement is

```
i = c.iterator().
```

A method call matches a label when

(a) the called method matches the method pattern, and

(b) the program values match their corresponding patterns.

By definition, any value matches an uppercase pattern. Here, the values of `i` and `c` trivially match the patterns $\underline{\texttt{X}}$ and $\underline{\texttt{C}}$. The method itself also matches the method pattern, but for a slightly more complicated reason than it appears. In Java, methods are identified by their fully qualified names plus the static types of the arguments. For simplicity, we ignore argument types and identify Java methods only by their fully qualified name and their arity. The called method `iterator` is in the class `ArrayList` and has arity 1. We identify it as follows.

```
java.util.ArrayList.iterator[1]
```

(Only static methods have arity 0.) Without the `prefix` directive, the method pattern would be `iterator[1]`. With the directive, however, the pattern is the following.

```
java.util.Collection.iterator[1]
java.util.Iterator.iterator[1]
```

The meaning is that these two methods *and* all those that override them match. Here, `ArrayList` implements `Collection`.

All conditions are met to enable the transition from `start` to `one`. When the transition is performed the values that matched $\underline{\texttt{X}}$ and $\underline{\texttt{C}}$ are written in the automaton variables x and c. For concreteness, let us assume these values are, respectively, the references α and γ. (Note that the program variables x and c may change their values, which is why we need a name for their values α and γ at this point.) After the transition is performed, the state $(\texttt{one}, [c : \gamma, x : \alpha])$ is active. The state $(\texttt{start}, [])$ remains active because the implicit transition

```
start -> start: *
```

is also enabled and performed. For convenience, TOPL assumes that there is a loop on `start` even if the user did not explicitly write it.

Step 2. For the second step, the statement to be executed is

```
j = c.iterator().
```

Now we need to consider the two active states

$$(\texttt{start}, []) \quad \text{and} \quad (\texttt{one}, [c : \gamma, x : \alpha])$$

in turn. For $(\texttt{start}, [])$, the same reasoning as for step 1 holds, so the states $(\texttt{start}, [])$ and $(\texttt{one}, [c : \gamma, x : \beta])$ are active after step 2. Note that now the automaton variable x remembers the value of the program variable `j`. For $(\texttt{one}, [c : \gamma, x : \alpha])$, we look at the transitions outgoing from vertex `one`.

```
one -> one: *
one -> two: Y := c.iterator()
```

The transition from `one` to `one` is always enabled, and performing it keeps states with vertex `one` active. The transition from `one` to `two` has two patterns, $\underline{\texttt{Y}}$ and $\underline{\texttt{c}}$. The uppercase pattern $\underline{\texttt{Y}}$ always matches; the lowercase pattern $\underline{\texttt{c}}$ matches only the value held by the automaton variable c. In this case, the automaton variable c and the program variable `c` both have the same value γ. Therefore, the transition from `one` to `two` is performed and the state $(\texttt{two}, [c : \gamma, x : \alpha, y : \beta])$ is activated.

Step 3. The third step involves the statement

```
i.next();
```

which enables the transitions

```
one -> one: *
start -> start: *.
```

All the active states remain active, but for different reasons. The active states with the vertex `start` and those with the vertex `one` remain active because the enabled transitions are loops. The active state with the vertex `two` remains active because no outgoing transition is enabled. Therefore the set of active states of the automaton remains unchanged.

Step 4. In the fourth step, the transition from `two` to `yBad` is performed. Notice that the label $\underline{\texttt{x}}.\texttt{remove}()$ does not have a left-hand side, which simply means that the returned value is irrelevant for this transition. The states corresponding to vertices `start` and `one` remain unchanged, because their outgoing transitions are disabled.

Step 5. For the fifth and final step, the statement to be executed is

```
j.next().
```

The label of the outgoing transition

$$\texttt{call}\ \underline{\texttt{y}}.*$$

of the active state $(\texttt{yBad}, [c : \gamma, x : \alpha, y : \beta])$, has two distinguishing features: the $*$ as a method pattern and the tag `call`. As before, before matching the method name, the prefixes are prepended.

```
java.util.Collection.*[*]
java.util.Iterator.*[*]
```

Then the $*$s are expanded, taking into account the `CLASSPATH`. We have a match because the expansion

```
java.util.Iterator.next[1]
```

is overridden by the method that is actually called.

The tag `call` is used when we want the automaton to take a transition precisely at the call-time of a method invocation. The automaton expresses that a call to one of `j`'s methods while vertex `yBad` is active constitutes an error. Notice that this is different from a label like $\underline{\texttt{X}} := \underline{\texttt{C}}.\texttt{iterator}()$ which may match only after the return value is known.

The execution we stepped through reaches the `error` vertex, so we conclude that the property is violated. Notice that in order to find a counterexample we need to keep track of the relation between several objects, in particular that iterators `i` and `j` are for the same collection `c`. One can write the TOPL property without understanding any implementation of Java's collections library.

2.2 More on Iterators

Other interesting properties of iterators [10, 21, 25] are also easy to express in TOPL.

Modifying a collection invalidates all its existent iterators:

```
property CollectionComodification
  prefix <java.util.{Collection,Iterator}>
  start -> iterating:    I := C.iterator()
  iterating -> modified: c.add(*), c.remove(*)
  modified -> error:     call i.*
```

The transition from `iterating` to `modified` should list all the methods of `Collection` that mutate it. If classes that implement `Collection` add mutating methods, then those should be included as well. This abstraction leak is intrinsic to Java, where sub-classing is not sub-typing.

Iterators should advance only if they are not exhausted:

```
property UnsafeIteratorNext
  prefix <java.util.{Collection,Iterator}>
  start -> iterating:        I := *.iterator()
  iterating -> notExhausted: <true> := i.hasNext()
  notExhausted -> iterating: i.next()
  iterating -> error:        i.next()
```

The transition from `iterating` to `notExhausted` is enabled when `hasNext` returns `true`. Java literals surrounded in angle brackets act as guards on transitions.

Method `remove` may only be called after `next`:

```
property RemoveBeforeNext
  prefix <java.util.{Collection,Iterator}>
  start -> created: I := *.iterator()
  created -> ok:    i.next()
  created -> error: i.remove()
```

The vertex `ok` is not special. The purpose of the transition going to `ok` is to deactivate the state with vertex `created`.

We have now seen four properties of iterators. The first two involve more than one object; the last two involve only one object. In all cases, TOPL properties are succinct and each corresponds to one English sentence.

2.3 Resources

Many resources impose two temporal properties: Before being used they must be acquired; after being acquired they must eventually be released. Resources include memory, files, sockets, locks. Memory must be allocated before reading or writing and should eventually be deallocated; files must be opened before reading or writing and should eventually be closed; sockets must be created before sending and receiving data and should eventually be closed; locks should be acquired before accessing the memory they protect and should eventually be released.

Resources are often represented by objects.

```
interface Resource {
  void acquire();
  void use();
  void release();
}
```

To create a resource, we use the static method `makeResource`.

A resource must be acquired when used:

```
property UseReleasedResource
  prefix <resources> // a package
  start -> released:    R := makeResource()
  released -> error:    r.use()
  released -> acquired: r.acquire()
  acquired -> released: r.release()
```

In simple cases, resources cannot repeatedly move between the states `released` and `acquired`.

```
property UseReleasedSimpleResource
  prefix <resources>  // a package
  start -> acquired:    R := makeResource()
  acquired -> released: r.release()
  released -> error:    r.use()
```

The following examples in this subsection only cover the case when `acquire` must be called explicitly.

A resource must not be released twice:

```
property DoubleRelease
  prefix <resources>  // a package
  start -> released:    R := makeResource()
  released -> acquired: r.acquire()
  acquired -> released: r.release()
  released -> error:    r.release()
```

Similarly, one can express that a resource must not be acquired twice.

A resource must eventually be released:

```
property ResourceLeak
  prefix <resources>  // a package
  start -> released:    R := makeResource()
  released -> acquired: r.acquire()
  acquired -> released: r.release()
  acquired -> error:    return Main.main[1]
```

TOPL is designed for safety properties, which have finite counter-examples. It is not possible to express the lack of resource leaks in non-terminating programs. It is possible, however, to check if resources are leaked when the program terminates.

2.4 SLIC Queue

SLIC [5] is the property language of SLAM [4]. Its authors give the following interesting example.

A queue may contain at most three zeros:

```
property TooManyZeros
  start -> cnt0: Q := makeQueue()
  cnt0 -> cnt1:  q.put(0)
  cnt1 -> cnt2:  q.put(0)
  cnt2 -> cnt3:  q.put(0)
  cnt3 -> error: q.put(0)
  cnt3 -> cnt2:  0 := q.get()
  cnt2 -> cnt1:  0 := q.get()
  cnt1 -> cnt0:  0 := q.get()
```

The vertex `cnt`x means that there are x zeros in q. A natural generalization is to allow at most n zeros. The size of the TOPL property text grows linearly with n. A better solution would be to allow incrementing and decrementing of automaton variables when transitions are performed. This feature can be easily added to TOPL. However, this is the only example we have encountered so far that would require incrementing and decrementing automaton variables. For this reason, we chose to keep TOPL simple and not include this feature.

2.5 Recursivity and Atomicity

This section is inspired by the higher-order temporal properties of Disney et al. [14].

The `sort` method is not recursive:

```
property RecursiveSort
  prefix <Sorter>
  start -> inSort: call *.sort[*]
  inSort -> ok:    return sort[*]
  inSort -> error: call *.sort[*]
```

Method `m` should be atomic:

```
property NotAtomicM
  observing <somePackage.*>
  prefix <somePackage.SomeClass>
  start -> inM: call *.m[*]
  inM -> ok:    return m[*]
  inM -> error: call *
```

The `observing` directive explicitly lists the observable events, thus choosing the proper granularity. In this case, all method calls within `somePackage` are observed. The method `m` is assumed to be in `SomeClass`.

2.6 Null Dereference

Simple non-temporal properties can also be expressed in TOPL.

No method should be called on `null`:

```
property NullDereference
  observing <*>
  start -> error: call <null>.*
```

3. Syntax

We begin the systematic presentation of TOPL with its syntax (Section 3.1). Before moving on to semantics (Section 4) we shall also introduce the syntax of SOOL, a **s**imple **o**bject-**o**riented programming **l**anguage (Section 3.3).

3.1 TOPL and iTOPL

TOPL aims to be intuitive: Labels look like method calls, there is a shorthand notation for parallel transitions, there is an implied loop on the vertex `start`, the `prefix` directive offers some extra convenience, and so on. From the point of view of semantics, however, these conveniences are in the way. For this reason we also define iTOPL (**i**nner TOPL), an even simpler language into which TOPL is desugared.

Figure 5 shows the syntax of TOPL. A property has a name, a set of `prefix` directives, a set of `observing` directives, and a set of transitions. Each transition has an arc (directed edge) and labels. Each arc has a source vertex and a target vertex. All vertices are identified by their name. Labels look roughly like method calls. Each label has a method pattern that is used to identify the set of methods to which the label refers. In the simple case, a method pattern consists of a string pattern for the name of the method and an integer that specifies the method arity. (For simplicity, TOPL does not use the static types of arguments to distinguish between overloaded methods.) The more interesting case is when there are value patterns for each argument and perhaps even for the result. Transitions may be tagged with `call` or `return` to specify exactly at what time they should be performed (see Section 4.1 for details).

There are three types of patterns used in TOPL—for strings, for integers, and for values. String patterns are POSIX globs [2] and match method names. (For simplicity, TOPL does not use full regular expressions as string patterns.) Integer patterns specify method arities. Value patterns are the most interesting. For each automaton variable *var* there are two associated patterns. The uppercase pattern `Var` matches any value and writes it in the automaton variable *var*. The lowercase pattern `var` reads the value of the automaton variable *var* and only matches that value. A Java literal surrounded by angle brackets acts as a pattern that matches only the value it denotes. A wildcard `*` pattern matches any value.

A TOPL property is *well-formed* when it satisfies the following conditions.

- For each automaton variable, a transition may prescribe at most one write. In other words, a given uppercase pattern may appear at most once within a label.

$$
\begin{array}{rcl}
\textit{Property} & ::= & \texttt{property}\ \textit{Identifier}\ \textit{Item}^* \\
\textit{Item} & ::= & \textit{Prefix} \mid \textit{Observing} \mid \textit{Transition} \\
\textit{Prefix} & ::= & \texttt{prefix}\ \texttt{<}\ \textit{StringPattern}\ \texttt{>} \\
\textit{Observing} & ::= & \texttt{observing}\ \texttt{<}\ \textit{StringPattern}\ \texttt{>} \\
\textit{Transition} & ::= & \textit{Arc}\ \texttt{:}\ \textit{Label}\ (\texttt{,}\ \textit{Label})^* \\
\textit{StringPattern} & ::= & (\textit{Letter} \mid \texttt{.} \mid \texttt{*} \mid \texttt{\{} \mid \texttt{\}} \mid \texttt{,})^+ \\
\textit{Arc} & ::= & \textit{Vertex}\ \texttt{->}\ \textit{Vertex} \\
\textit{Label} & ::= & \textit{Tag}^?\ \textit{MethodPattern} \\
\textit{Vertex} & ::= & \textit{Identifier} \\
\textit{Tag} & ::= & \texttt{call} \mid \texttt{return} \\
\textit{MethodPattern} & ::= & \textit{ResultPattern}^?\ \textit{NamePattern}\ \textit{ArgumentsPattern} \\
\textit{ResultPattern} & ::= & \textit{ValuePattern}\ \texttt{:=} \\
\textit{NamePattern} & ::= & \textit{StringPattern} \\
\textit{ArgumentsPattern} & ::= & \texttt{(}\ (\textit{ValuePattern}\ (\texttt{,}\ \textit{ValuePattern})^*)^?\ \texttt{)} \\
\textit{ArgumentsPattern} & ::= & \texttt{[}\ \textit{IntegerPattern}\ \texttt{]} \\
\textit{ValuePattern} & ::= & \texttt{*} \mid \texttt{<}\ \textit{Literal}\ \texttt{>} \mid \textit{UppercaseId} \mid \textit{LowercaseId} \\
\textit{IntegerPattern} & ::= & \texttt{*} \mid \textit{IntegerLiteral}
\end{array}
$$

Figure 5. Syntax of TOPL

$$
\begin{array}{rcl}
\textit{Property} & ::= & \texttt{property}\ \textit{Identifier}\ \textit{Transition}^* \\
\textit{Transition} & ::= & \textit{Arc}\ \texttt{:}\ \textit{TransitionStep}^* \\
\textit{TransitionStep} & ::= & \texttt{(}\textit{TagGuard}\ \textit{MethodGuard}\ \textit{ValueGuard}^*\ \textit{Action}^*\texttt{)} \\
\textit{TagGuard} & ::= & \texttt{*} \mid \texttt{call} \mid \texttt{return} \\
\textit{MethodGuard} & ::= & \textit{NamePattern}\ \texttt{[}\ \textit{IntegerPattern}\ \texttt{]} \\
\textit{ValueGuard} & ::= & \texttt{[}\ \textit{ValueIndex}\ \texttt{]}\ \texttt{=}\ (\ \textit{Identifier} \mid \textit{Literal}\ ) \\
\textit{Action} & ::= & \textit{Identifier}\ \texttt{:=}\ \texttt{[}\ \textit{ValueIndex}\ \texttt{]} \\
\textit{ValueIndex} & ::= & \textit{IntegerLiteral}
\end{array}
$$

Figure 6. Syntax of iTOPL

- Automaton variables must be written before being read. In other words, lowercase patterns must be preceded by corresponding uppercase patterns on all paths coming from `start`.

It is easy to check that a property is well-formed [20]. From now on we assume TOPL properties to be well-formed.

Figure 6 shows the syntax of iTOPL. The missing productions (such as *Arc*) are the same as for TOPL. Each transition is labeled by a list of steps. Each step has guards and actions. There are several types of guards. A tag guard may restrict the type of events (`call` or `return`). A method guard requires that the method name and its arity match certain patterns. Value guards impose restrictions on the values carried by events. Both value guards and actions refer to the values carried by events using an index.

3.2 Translating TOPL into iTOPL

TOPL is designed to express properties over traces of events pertaining to method calls. A method call $u.m(v_1, \ldots, v_n)$ that returns w will generate the following two events:

$$\texttt{call}_{m[n+1]}\ [u, v_1, \ldots, v_n]$$

and

$$\texttt{return}_{m[n+1]}\ [w]$$

The call events are matched by labels of the form

$$\texttt{call}\ p_0.m(p_1, \ldots, p_n)$$

with n value patterns for the arguments, while the return events are matched by labels of the form

$$\texttt{return}\ p_0 := m$$

with one value pattern for the return value. In both cases the TOPL labels give rise to an iTOPL transition with one step.

The guard is constructed by stating that all automaton variables should have the appropriate values. That is, all lowercase automaton

patterns $\underline{x} = p_i$ amongst the value patterns give rise to a guard term of the form

$$[i] = x,$$

referring to the array of values carried by the event.

The action expresses that automaton variables should be updated. So all uppercase automaton variables $\underline{X} = p_i$ amongst the value patterns give rise to an update term of the form

$$x := [i]$$

The action is then the concatenation of all such update terms.

Example 1. The TOPL label `call C.bar(*, x)` becomes the iTOPL step

$$\texttt{call}\ \ \texttt{\{foo.,\}bar}[3] \quad [2] = x \quad c := [0],$$

assuming the prefix directive `prefix <foo>`.

But it is also possible to write labels of a third form in TOPL, namely

$$p_0' := p_0.m(p_1, \ldots, p_n).$$

These are translated into two iTOPL steps corresponding, in order, to the TOPL labels `call` $p_0.m(p_1, \ldots, p_n)$ and `return` $p_0' := m$.

In our implementation [20], the translation from TOPL to iTOPL is done during parsing. In other words, TOPL corresponds to the concrete syntax, and iTOPL corresponds to the AST data structures.

3.3 SOOL

Traces of events act as a thin interface between programs and iTOPL properties. To use iTOPL with some programming language, one must define how programs in that language produce events. Also, because iTOPL is low-level, one should design a high-level property language. TOPL is an example of a high-level property language built on top of iTOPL that is suitable in an object-oriented setting. SOOL is an example of an object-oriented programming language that produces events.

The following is a small snippet of a SOOL program.

```
class User
  Unit remove(Collection c, Object x)
    var Bool hasNext
    var Iterator i := c.iterator()
    do { hasNext := i.hasNext() }
    while hasNext
      var Object y := i.next()
      if x == y { i.remove() }
    return unit
```

In SOOL, statements are grouped by {curly brackets} or by indentation. SOOL expressions do not have side-effects. Method calls have side-effects—they are statements. There is only one type of loop, which checks its condition at some arbitrary point (like `\loop` in TEX). Other aspects of SOOL are fairly standard.

Figure 7 shows a big part of SOOL's syntax.

Program	::= *Class** *Main*	
Class	::= `class` *Identifier Member**	
Main	::= `main` *Body*	
Member	::= *Type Identifier*	data
Member	::= *Type Identifier* `(` *Formals*$^?$ `)` *Body*	code
Body	::= *Statement**	
Formals	::= *Type Identifier* (`,` *Type Identifier*)*	
Statement	::= `return` *Expression*	return
Statement	::= *Reference* `:= new`	allocate
Statement	::= *Reference* `:= *`	read
Statement	::= *Reference* `:=` *Reference* `(` *Actuals*$^?$ `)`	call
Statement	::= `do` *Body* `while` *Expression Body*	loop
Statement	::= `if` *Expression Body* `else` *Body*	test
Reference	::= *Expression* `.` *Identifier*	
Actuals	::= *Expression* (`,` *Expression*)*	

Figure 7. (Partial) Syntax of SOOL

4. Semantics

A program's semantics is a set of event traces; an automaton's semantics is also a set 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.

Notation 3. Sets are typeset in SansSerif with two exceptions—$\mathbb{B}$ is the set $\{0, 1\}$, and $\mathbb{N}$ is the set $\{0, 1, 2, \ldots\}$. We write $\mathsf{A} \times \mathsf{B}$ for the set of all pairs (a, b) with $a \in \mathsf{A}$ and $b \in \mathsf{B}$. We write $\mathsf{A} \to \mathsf{B}$ for the set of functions from A to B. In particular, the powerset of A is isomorphic to $\mathsf{A} \to \mathbb{B}$. Note that $(\mathsf{A} \to \mathsf{B}) \subset \big((\mathsf{A} \times \mathsf{B}) \to \mathbb{B}\big)$. For all $f \in \mathsf{A} \to \mathsf{B}$, we write $f\ a$ or $f(a)$ to denote the unique b such that $(a, b) \in f$. We write $\mathsf{A} \rightharpoonup \mathsf{B}$ for the set of finite partial maps from A to B. Again, $(\mathsf{A} \rightharpoonup \mathsf{B}) \subset \big((\mathsf{A} \times \mathsf{B}) \to \mathbb{B}\big)$. However, there are functions that are not finite partial maps, and there are finite partial maps that are not functions. For all $f \in \mathsf{A} \rightharpoonup \mathsf{B}$, the domain $\{a \mid (a, b) \in f \text{ for some } b\}$ is finite. We write $f\ a$ or $f(a)$ only when a is in the domain of f. We write $\mathsf{A}\ \mathsf{array}$ for $\bigcup_{n \in \mathbb{N}} \big(\{0, 1, \ldots, n-1\} \to \mathsf{A}\big)$. For $i \in \mathbb{N}$ and $a \in \mathsf{A}\ \mathsf{array}$, we will usually write a_i instead of $a\ i$.

4.1 Semantics of iTOPL

Each property gives rise to an automaton defined over the labeled multigraph (Vertex, Arc) given by the transitions: Vertex is the set of vertices mentioned as endpoints of the arcs and Arc is the set of labeled arcs mentioned in transitions.

The automaton takes as input a trace of events. Each event e contains an array of values. Within guards and actions we write $e[i]$ for the ith value associated to event e (0-based). For values, we assume a countable set Value and for events we assume a finite set Event of events with known arity.

We will now describe how the automaton reacts to events, specifically how labels are evaluated. As this is a slightly non-standard automaton, there are some important things to point out. Firstly, the state of the automaton is not just a vertex but also a store.

That is, there is a set Variable of automaton variables, and the state of the automaton is given by specifying the vertex in the graph as well as the value of all automaton variables.

We model stores as finite partial maps with finite domain.

$$\mathsf{Store} = \mathsf{Variable} \rightharpoonup \mathsf{Value}$$

And so a state of the automaton has the type

$$\mathsf{State} = \mathsf{Vertex} \times \mathsf{Store}$$

Given the state of the automaton and an event we would like to know which transitions are enabled. But actually, we do not have enough information to determine that. This is a consequence of the second slightly non-standard aspect of our automatons, namely that each label carries not only a single guard but potentially a list of guards. We call the length of this list, the *depth* of the transition.

In order for a transition to be enabled, all the guards along its list have to evaluate to true. We will describe this evaluation in a moment, but now we note that each guard along the transition consumes an event. Thus, if a transition has depth n, then we have to examine the next n events to see if the transition is enabled.

A further consequence is that the following events to be received are not the same at the end of each enabled transition. If two transitions turn out to be enabled, one with depth 2 and the other with depth 5, then the end state of the first transition will see the third event of the trace next, while the end state of the second transition will see the sixth event next. For this reason, it is necessary to keep

track of which events are next to be received by each state during a run of the automaton.

To formalize this bookkeeping, we introduce the notion of *execution state*. For brevity, we call execution states worlds.

$$\mathsf{World} = \mathsf{State} \times \mathsf{Trace}$$

The first component records the state of the automaton (vertex and store) and the second component records the remaining trace of events for that state.

We now describe when transitions are enabled. For this we have to look at labels. A label is a list of pairs of guards and actions. A guard compares the values in an event with those in a store and concludes either pass or no pass:

$$\mathsf{Guard} = \mathsf{Event} \times \mathsf{Store} \to \mathbb{B}$$

if the event passes, the corresponding action is performed on that event. Actions modify the store, using values from an event:

$$\mathsf{Action} = \mathsf{Event} \times \mathsf{Store} \to \mathsf{Store}$$

In order for a transition of depth n to be enabled for a trace of events, the first n events of the trace must pass the n guards on the label with the stores modified by the guards:

$$\begin{array}{rl} & \mathit{enabled}((g_1, a_1), \ldots, (g_n, a_n); e_1, \ldots, e_n; s) \\ = & g_1(e_1, s_0) \ \wedge \ \ldots \ \wedge \ g_n(e_n, s_{n-1}) \end{array}$$

where

$$s_0 = s \tag{1}$$

$$s_i = a_i(e_i, s_{i-1}) \qquad \text{for } i \in 1..n \tag{2}$$

If the transition is enabled and performed, the store for the target vertex will be s_n.

Remark 1. It is natural to ask whether transitions of depth > 1 could be desugared into transitions of depth 1. We tried (1) to rewrite guards and do all the actions at the end, (2) to exploit nondeterminism, and (3) to use a special *undo* action. All three approaches run into fundamental barriers [19].

And now we reach the final slightly non-standard aspect of our automatons, namely that if no transitions are enabled for a given state and trace of events, the automaton does not get stuck but is allowed to consume one event without changing its state. Note that this is not equivalent to an implied self-loop on all states, as dropping events is not allowed if there are any enabled transitions. In that case one of the enabled transitions is performed and the automaton execution state becomes $((v, s_n), \mathit{events}_n)$, where v is the vertex at the end of the arc of the transition, s_n is defined as above and events_n is the current executions state's event trace with the first n events dropped.

We can define a predicate that determines the possible execution state transitions

$$\mathit{Step} \in \mathsf{World} \to \mathsf{World} \to \mathbb{B}$$

this predicate is defined in Figure 8. From this we can define a non-deterministic step function, where all enabled transitions are performed:

$$\mathit{NdetStep} \in (\mathsf{World} \to \mathbb{B}) \to (\mathsf{World} \to \mathbb{B}) \tag{3}$$

$$\mathit{NdetStep}\ S = \{s \mid \mathit{Step}(s', s) \text{ for some } s' \in S\} \tag{4}$$

An iterated version of *NdetStep* is useful for defining reachable states. We define it as the least fixed point for the following equation.

$$\mathit{NdetStep}^\star\ S = S \cup \mathit{NdetStep}^\star\ (\mathit{NdetStep}\ S) \tag{5}$$

Finally, we can define the set of traces described by an automaton:

$$\{e \mid \exists \sigma' e',\ ((\texttt{error}, \sigma'), e') \in \mathit{NdetStep}^\star\ \{((\texttt{start}, []), e)\}\}$$

```
STEP ((x1, σ1), e1) ((x2, σ2), e2)
 1  if e2 is not a suffix of e1 then return 0
 2  e := e1 without the suffix e2
 3  for each transition ((y1, y2), l)
 4       if (y1, y2) ≠ (x1, x2) ∨ len l ≠ len e then continue
 5       σ := σ1
 6       for each k ∈ 1 ... len l
 7            (g, a) := l[k]
 8            if ¬g(e_k, σ) then continue to line 3
 9            σ := a(e_k, σ)
10       if σ = σ2 then return 1
11  return len e = 1 ∧ σ1 = σ2
```

Figure 8. One automaton step

These are the traces that drive the automaton from the `start` vertex (with an empty store) to the `error` vertex.

4.2 Semantics of SOOL

The *program state* holds method parameters and dynamically allocated objects. Method parameters are held in a store; dynamically allocated objects are held in the heap. The *heap* is a finite partial map from (object reference, field name) pairs to values.

$$\mathsf{SoolState} = \mathsf{Store} \times \mathsf{Heap} \tag{6}$$

$$\mathsf{Store} = \mathsf{Variable} \rightharpoonup \mathsf{Value} \tag{7}$$

$$\mathsf{Heap} = (\mathsf{Value} \times \mathsf{Variable}) \rightharpoonup \mathsf{Value} \tag{8}$$

The input is a stream of values; the output is an array of events. The *program world* keeps track of the program state, the input yet to be consumed, and of the output already produced.

$$\mathsf{Input} = \mathbb{N} \to \mathsf{Value} \tag{9}$$

$$\mathsf{Trace} = \mathsf{Event\ array} \tag{10}$$

$$\mathsf{SoolWorld} = \mathsf{SoolState} \times \mathsf{Input} \times \mathsf{Trace} \tag{11}$$

$$((\sigma, h), i, o) \in \mathsf{SoolWorld} \tag{12}$$

Equation (12) shows the general form of an element of SoolWorld: σ is the store that holds parameters, h is the heap that holds dynamically allocated objects, i is the input stream yet to be processed, and o is the array of events already emitted. The implementation [20] also has local variables, which are treated similarly to method parameters.

Executing a SOOL program amounts to executing the `main` body, which is a compound statement.

$$\mathit{exec} \in (\mathsf{SoolWorld} \times \mathsf{Statement}) \to (\mathsf{SoolWorld} \times \mathsf{Value}) \tag{13}$$

Method calls are interesting. The general form of a method call is

$$e.x := f.m(g_1, \ldots, g_n)$$

where e, f, $g_1, \ldots, g_n$ are expressions, x identifies a data field, and m identifies a method. Expressions evaluate to values and do not have side-effects.

$$\mathsf{Expression} = \mathsf{SoolState} \to \mathsf{Value} \tag{14}$$

$$e, f, g_1, \ldots, g_n \in \mathsf{Expression} \tag{15}$$

Notation 4. The map $\sigma[k : v]$ is the same as the map σ, except it maps k to v.

Figure 9 shows the definition of *exec* for method calls. In spite of the imperative-looking notation, there is no mutation: Different versions of σ, h, i, and o get different indices. Line 1 introduces a shorthand notation for the program state s. Line 2 looks up in the program text a method named m. We assume that method names are unique. The formal parameters of m are $x_1, \ldots, x_n$. We assume a

$\text{EXEC}\Big(\big((\sigma_0, h_0), i_0, o_0\big), \big(e.x := f.m(g_1, \ldots, g_n)\big)\Big)$

1 $s := (\sigma, h)$
2 $((x_1, \ldots, x_n), b) := \mathit{methodOfName}\ m$
3 $\sigma_1 := [\textbf{this} : (f\ s), x_1 : (g_1\ s), \ldots, x_n : (g_n\ s)]$
4 $o_1 := o_0$ with $(\texttt{call}m, [f\ s, g_1\ s, \ldots, g_n\ s])$ appended
5 $(((\sigma_2, h_2), i_2, o_2), v) := \mathit{exec}\big(((\sigma_1, h_0), i_0, o_1), b\big)$
6 $o_3 := o_2$ with $(\texttt{return}m, [v])$ appended
7 $h_3 := h_2[(e\ s, x) : v]$
8 **return** $(((\sigma_2, h_3), i_2, o_3), ())$

Figure 9. Executing one SOOL method call

type-checker enforces that all calls are made with the correct number of arguments. The body of m is b. Line 3 constructs a store that maps formal arguments to the actual values; σ_1 is commonly called the *call stack frame* of m. Line 4 emits the first event: It has tag `call`m and carries the actual argument values. Line 5 executes the body, by calling *exec* recursively. Line 6 emits the second event: It has tag `return`m and carries the return value. Line 7 stores the return value in the heap. Finally, line 8 returns the updated program execution state and the unit value $()$.

Other statements are interpreted as usual and are not interesting.

Given a program with the main body b, the output trace o corresponding to some input i is obtained by starting the execution with an empty store $[]$, an empty heap $[]$, and an empty output $[]$.

$$((\_, \_, o), \_) := \mathit{exec}\big(((([], []), i, []), b\big) \tag{16}$$

4.3 Implementation

In order to test the semantics, we implemented a TOPL and SOOL interpreter [20]. Indeed, the implementation helped us understand TOPL. There are two main differences with the presentation above.

First, we under-approximate nondeterminism by randomness. Where the semantics prescribe a set S of active states, the interpreter tracks only one active state $s \in S$. A random number generator guides the choice of s at each step. Thus, different seeds of the random number generator explore different traces. Simulating nondeterminism by randomness is unsound. However, the TOPL and SOOL interpreter is not meant to prescribe how dynamic analysis of TOPL properties should be done.

Second, the trace of events is communicated from the SOOL interpreter to the TOPL interpreter lazily. The only consequence is that less memory is used. At most two events are remembered at any one time, rather than the whole trace.

If the reader finds any part of Section 4 ambiguous or incomplete, then the code should clarify the details.

5. Related Work

Our work is based on the concept of typestate [29] originally developed for imperative programs and extends this fundamental concept by integrating notions typical of object-oriented programs. We are certainly not the first in doing this: there are several extensions of typestate to object-oriented programming in the literature. A modular static verification method for typestate protocols is introduced in [8]. The specification method is based on linear logic and relations among objects in the protocol are monitored by a tailored system of permissions. The method is highly modular and presumably efficient. 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 [7, 13] 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. This work focuses on analysis whereas we focus on presenting a useful specification language as the base for verification. It is likely that our language could be fruitfully combined with their analysis technique.

An automaton-based formalism for specifying properties of software interfaces were introduced in [12] . This language aims at capturing assumptions about the order in which the methods of a component are called and the order in which the component calls external method. In contrast to TOPL, this formalism is mainly used to check the compatibility of the interfaces of two components and it is designed to be applied at model level rather than code level. A specification language for interface checking aimed to C programs (called SLIC) is introduced in [5]. 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 [4]. 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 [1] and Spec♯ [6]. However the latter two languages focus on class specifications and do not have temporal features.

In [14] 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 so look similar to our automatons, if for a quite different setting. Here, the specifications are function-centered, though, and again, capturing inter object relations seems somewhat awkward.

ConSpec [3] 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 `IteratorComodification` (Section 2.1).

6. Conclusions and Future Work

This paper presents ongoing work on the design of TOPL—a language for expressing temporal safety properties for object-oriented programs. TOPL is translated into an intermediate language, iTOPL. The low-level iTOPL is very regular and so its semantics are easy to define. The high-level TOPL is designed to look familiar to an object-oriented programmer. More importantly, TOPL curtails iTOPL's expressivity in a way that, we expect, will benefit static verification. Our implementation helped clarify TOPL's semantics.

We designed TOPL for dynamic and static analysis of Java programs. The next steps are to develop a dynamic analyzer and a static analyzer. For dynamic analysis, we intend to instrument Java bytecode and run a checker in parallel with the real program. The challenge we expect here is to reduce the run-time overhead. For static analysis, we intend to build on the jStar framework [15]. Remember that TOPL's guards are basically aliasing checks. Separation logic [28] is a good formalism for reasoning about aliasing and is the foundation of jStar. The challenges we expect here are convergence and scalability. For convergence, we must find suitable abstractions, which obtain meaningful and precise over-approximations of the state space of the programs. For scalability, we may need a tailored version of bi-abduction inference [11].

Another possible development would be to design other high-level property languages on top of iTOPL. For example, it is conceivable that iTOPL could be used as an intermediate language in tools that verify temporal properties of programs written in Haskell [27] or Boogie [23].

Finally, we intend to define TOPL's semantics also in a concurrent setting.

Acknowledgments

The authors thank the anonymous reviewers for the thorough feedback. This work was supported by the EPSRC grants EP/H011749/1 and EP/G006245/1.

References

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

[2] man glob.h.

[3] I. Aktug and K. Naliuka. ConSpec — a formal language for policy specification. *Electr. Notes Theor. Comput. Sci.*, 197(1):45–58, 2008.

[4] 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. ISBN 3-540-42345-1.

[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] 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.

[7] 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. ISBN 1-59593-014-0.

[8] 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. ISBN 978-1-59593-786-5.

[9] K. Bierhoff, N. E. Beckman, and J. Aldrich. Practical API protocol checking with access permissions. In S. Drossopoulou, editor, *ECOOP*, volume 5653 of *Lecture Notes in Computer Science*, pages 195–219. Springer, 2009. ISBN 978-3-642-03012-3.

[10] E. Bodden, P. Lam, and L. J. Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In M. J. Harrold and G. C. Murphy, editors, *SIGSOFT FSE*, pages 36–47. ACM, 2008. ISBN 978-1-59593-995-1.

[11] 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. ISBN 978-1-60558-379-2.

[12] L. de Alfaro and T. A. Henzinger. Interface automata. In *ESEC / SIGSOFT FSE*, pages 109–120, 2001.

[13] 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. ISBN 3-540-22159-X.

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

[15] D. Distefano and M. J. Parkinson. jStar: towards practical verification for Java. In Harris [22], pages 213–226. ISBN 978-1-60558-215-3.

[16] J. Field, D. Goyal, G. Ramalingam, and E. Yahav. Typestate verification: Abstraction techniques and complexity results. *Sci. Comput. Program.*, 58(1-2):57–82, 2005.

[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. ISBN 1-59593-263-1.

[18] J.-Y. Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

[19] R. Grigore. TOPL implementation design note. http://goo.gl/PF35z, 2011. Non-unit transitions.

[20] R. Grigore and R. L. Petersen. TOPL implementation. http://goo.gl/KD8Sy, 2011.

[21] C. Haack and C. Hurlin. Resource usage protocols for iterators. *Journal of Object Technology*, 8(4):55–83, 2009.

[22] 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*, 2008. ACM. ISBN 978-1-60558-215-3.

[23] K. R. M. Leino. This is Boogie 2. KRML 178, 2008.

[24] Z. Manna and A. Pnueli. *Temporal verification of reactive systems - safety*. Springer, 1995. ISBN 978-0-387-94459-3.

[25] N. A. Naeem and O. Lhoták. Typestate-like analysis of multiple interacting objects. In Harris [22], pages 347–366. ISBN 978-1-60558-215-3.

[26] M. Parkinson. Class invariants: the end of the road? In *IWACO 2007*. Position Paper.

[27] S. Peyton Jones, editor. *Haskell 98 Language and Libraries — The Revised Report*. Cambridge University Press, Cambridge, England, 2003.

[28] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002. ISBN 0-7695-1483-9.

[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.