Formal Modeling of a Slicing Algorithm for Java Event Spaces in PVS

Néstor Cataño

catano@cs.york.ac.uk
Department of Computer Science, The University of York, U.K.

Abstract. This paper presents the formalization of an algorithm for slicing Java event spaces in PVS. In short, Java event spaces describe how multi-threaded Java programs operate in memory. We show that Java event spaces can be sliced following an algorithm introduced in previous work and still preserve properties in a subset of CTL. The formalization and proof presented in this paper can be extended to other state-space reduction techniques as long as some *sufficient* conditions are fulfilled.

1 Introduction

Java event spaces [2, 10] are partial orders of the actions performed by the main memory and the threads of a multi-threaded Java program. In previous work [1] we showed how classical slicing techniques can be employed to reduce the size of Java event spaces. Roughly speaking, when slicing, only the parts of the Java event space upon which the elements of the slicing criterion depend are retained, while the underlying structure of the Java event space is preserved. Furthermore, we dealt with the problem of *aliasing* that arises when two variables of the event space point to the same memory address. An algorithm that takes a Java event space and calculates aliasing dependencies for relevant variables of the slicing criterion was outlined.

Here, we formalize the aliasing algorithm in PVS [8] and, for parts of the algorithm, show how Java event spaces can be sliced and still verify the same properties in CTL without the *next* operator [4, 5] as non-sliced Java event spaces. To cope with the proof, we propose a two-step trace reconstruction approach. This process of reconstruction outlines conditions which must be verified for other algorithms working with state-space reduction to preserve properties in CTL. The outline of these conditions and the PVS formalization are the two main contributions of this paper.

This paper is structured as follows. Section 2 introduces Java event spaces formally and gives an example where an event space for a multi-threaded Java program is calculated. Section 3 presents the slicing algorithm introduced in [1]. Section 4 formalizes Java event spaces as finite-state automata. This allows for defining how CTL properties are evaluated on Java event spaces. Section 5 shows that the algorithm introduced in [1] preserves properties expressed in a subset of CTL. During the proof we identify sufficient conditions employable in similar proofs when different kinds of state-space algorithms are applied. Finally, Section 6 gives conclusions and presents future work.

2 Java event spaces

Chapter 17 of the Java Language Specification (JLS) [7] gives a detailed yet not formal specification of how multi-threaded Java programs should operate. This specification states that a main memory shared by all the threads in the program exists, and that it keeps a global copy of the variable values. The specification also says that each thread has its own local working memory which keeps a copy of variables of the main memory. As a thread executes code, some *events* in memory happen. An event in memory represents the occurrence of some *action* either in the main memory or in the working memory of some thread. A thread θ can for example use the *right value* v of a *left value* l, action $\mathbf{use}(\theta, l, v)$, or it can assign it a new value v, action $\mathbf{assign}(\theta, l, v)$. Right values represent object values as seen in memory: native type values and references. Left values are memory addresses. When copying the value v of l from the main memory to the working memory of θ, two actions must occur: first, a $\mathbf{read}(\theta, l, v)$ action performed by the main memory, followed at some unspecified time later by a $\mathbf{load}(\theta, l, v)$ action performed by the working memory. When copying the value v of l from the working memory of θ to the main memory, two actions must occur as well: a $\mathbf{store}(\theta, l, v)$ action performed by the working memory, followed at some unspecified time later by a $\mathbf{write}(\theta, l, v)$ action performed by the main memory. Actions $\mathbf{lock}(\theta, o)$ and $\mathbf{unlock}(\theta, o)$ acquire and relinquish a lock on the object o on behalf of the thread θ.

We use the notation $x{:}\,\mathbf{y}$ to indicate that x is an event labeled with an action $\mathbf{y}$. Memory actions are **read**, **write**, **lock** and **unlock**; thread actions are **load**, **use**, **assign**, **store**, **write**, **lock** and **unlock**; and lock actions **lock** and **unlock**. With $x{:}\,\mathbf{read}(l)$ we indicate that $x{:}\,\mathbf{read}(\theta, l, v)$ for some θ and v. Analogously $x{:}\,\mathbf{write}(v)$ means that $x{:}\,\mathbf{write}(\theta, l, v)$ for some θ and l. Similarly for the other actions. We use $ref_{\mathtt{x}}$ to indicate the reference associated with variable `x`.

Formally expressed, a Java event space is a set of events X labeled with actions, provided with a partial order $\leq$, such that $(X, \leq)$ respects the rules of well-formedness enunciated in the specification of the Java Memory Model (JMM) [2, 10]. We give an example of event space generation for a Java program that describes the interaction of two threads executing two methods in parallel.

Example 1 Suppose that two threads θ_1 and θ_2, executing respectively methods `p()` and `q()` on some object `this`, exist.

```
void p(){ synchronized(this){x.i = 7; x.j = 5;} y.i = x.j; }
void q(){ synchronized(this){y = x; z.i = y.i;} z.i = 9; }
```

Further, suppose that variables `x`, `y` and `z` are instances of some class `C` with variables `i` and `j` of type `int`, whose initial values are 0 for both. Figure 1 shows an event space for the interaction of actions generated by both the main memory and the working memories local to θ_1 and θ_2. This event space represents a possible execution of the program for the interaction of these two threads. The partial order relation $\leq$ of actions is represented in the figure by (multiple) vertical, horizontal and diagonal arrows.

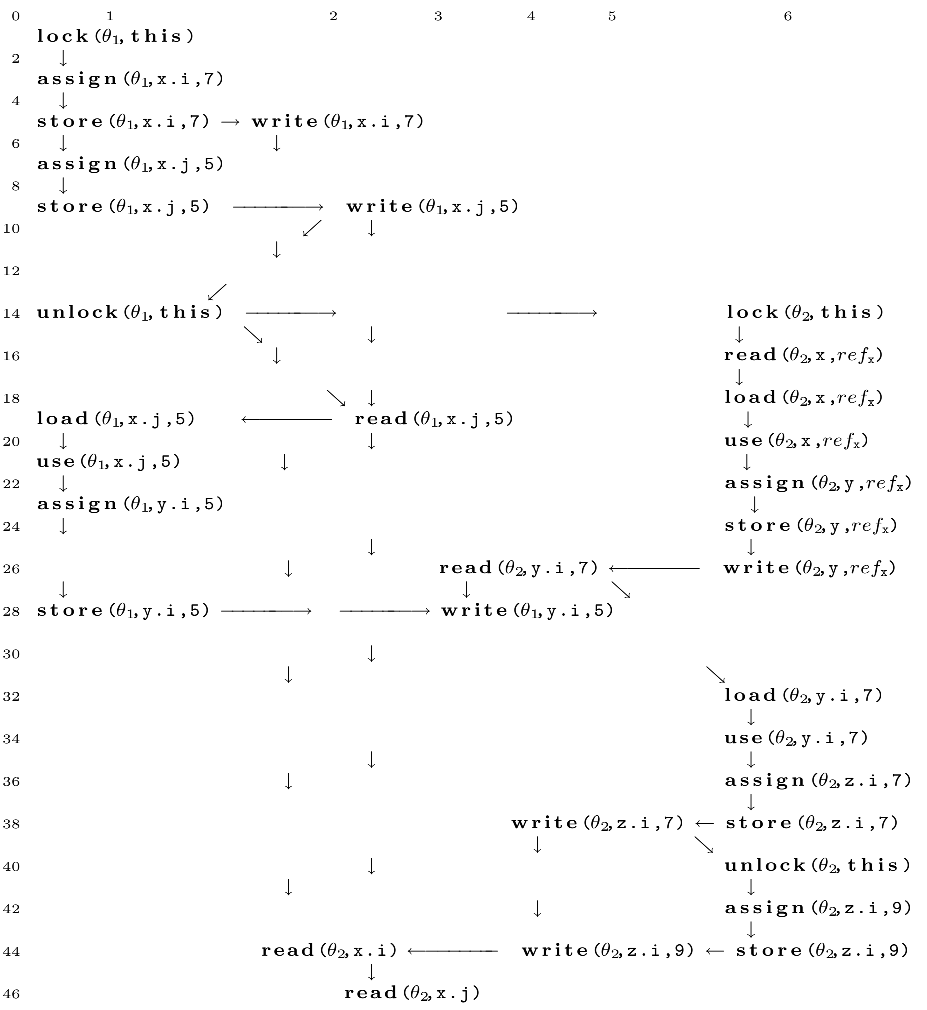

Fig. 1. Java event space generation

For readability, the reflexivity and transitivity of the partial order relation have not been sketched. Because, according to the JMM, thread actions for the same thread make up a total order, actions for θ_1 and θ_2 in Columns 1 and 6 form increasing chains. The same thing happens for memory actions on the same variable, hence actions **read** and **write** for variables `x.i`, `x.j`, `y.i` and `z.i` in Columns 2 to 5 also form increasing chains.

In Java, when a thread is executing a synchronized code fragment of some object, no other thread may acquire a lock associated with the same object. Though, a single thread may acquire several times the lock associated with the same object. In our example, `p()` and `q()` are synchronized. We sketch the case when θ_1 acquires the lock on object `this` before θ_2. After acquiring the lock associated with `this` (Column 1, Line 1), θ_1 executes the body of the synchronized part of `p()` (Lines 3 to 9), and finally relinquishes the lock (Column 1, Line 14). In Line 3, θ_1 assigns 7 to `x.i` and, in Line 5, it writes the new value of `x.i` from its working memory to the main memory. The thread θ_1 then executes the rest of the synchronized part of `p()`, since θ_2 cannot acquire a lock on the object `this` in order to execute the synchronized part of `q()`.

From then on θ_1 and θ_2 continue to execute concurrently, in particular the Java Language Specification does not specify whether the locking of `this` on behalf of θ_2 or the execution of `y.i` $=$ `x.j` by θ_1 occurs first. One can only be sure that *writing to* and *reading from* a certain variable must respect a total order. Figure 1 sketches the case when reading from `y.i` in the synchronized statement of `q()` occurs before the writing to `y.i` in `p()` (Column 4). In Column 6, θ_2 reads the value of `x` from the main memory (Lines 16 and 18), uses its value (Line 20), and finally assigns the reference of `x` to `y` (Line 22). Consequently, from there on, `x` and `y` will be *aliased*. The rest of Column 6 shows the memory interactions corresponding to the execution of `z.i` $=$ `9`; by θ_2.

3 The slicing algorithm

A *program slice* consists of those parts of a program that potentially affect some points of interest, which in turn depend on the property that is checked. These points of interest are called *slicing criterion* and its variables *relevant variables*. A *slice set* S_C is composed of nodes in the Java event space from which nodes in the slicing criterion C are reachable via the dependency relation $\xrightarrow{fd_a}$, defined below, with the intuitive meaning $w \xrightarrow{fd_a} r$ if the event r is *aliasing flow dependent* on the event w. In addition to elements in S_C, a *residual slice set* S_{Cr} must contain other events, so that the Java event space formed of events in S_{Cr} and having as order relation the partial order relation of the event space restricted to S_{Cr} make up a *program slice*.

Predicate $FlowDep?$ below constitutes the basis for the slicing algorithm introduced in [1]. This predicate formalizes the notion of aliasing flow dependency $\xrightarrow{fd_a}$; more concretely $w \xrightarrow{fd_a} r$ is given by $FlowDep?(w, r)$. In the first case of definition of $FlowDep?$, when `y=x` and w "occurs before" r ($w \leq r$), **read**(`x.i`) is alias flow dependent of **write**(`y.i`) if there is no **write** action w_1 (other than w) between w and r that modifies the field `i` of `x`. When `y` and `x` are distinct two other cases arise. First, if $w \leq r$, it is not only necessary to ensure that there is no

write action w_1 between w and r modifying `x.i`, but also that `y` is an alias of x when w happened. When `y` and `x` are different, it is possible that w:**write**(`y.i`) and r:**read**(`x.i`) are not related, since the e Java Language Specification does not ensure a total order for **write** and **read** actions on different left values. In this case a *defensive* approach is adopted by considering that **write**(`y.i`) modifies `x.i` provided that `y` is an alias of `x` when w happened.

$$FlowDep?(w{:}\,\mathbf{write}(\mathtt{y.i}), r{:}\,\mathbf{read}(\mathtt{x.i})) =$$

$$\begin{cases} true, \text{ if } \begin{cases} 1.\ \mathtt{y}{=}\mathtt{x} \ \wedge \ w{\le}r \ \wedge \\ 2.\ \neg\exists w_1{:}\,\mathbf{write}(\mathtt{z.i}).Alias?(\mathtt{x},\mathtt{z},w_1) \wedge w{<}w_1{\le}r \wedge FlowDep?(w_1,r) \end{cases} \\ true, \text{ if } \begin{cases} 1.\ \mathtt{y}{\neq}\mathtt{x} \ \wedge \ w{\le}r \ \wedge \\ 2.\ Alias?(\mathtt{x},\mathtt{y},w) \ \wedge \\ 3.\ \neg\exists w_1{:}\,\mathbf{write}(\mathtt{z.i}).Alias?(\mathtt{x},\mathtt{z},w_1) \wedge w{<}w_1{\le}r \wedge FlowDep?(w_1,r) \end{cases} \\ true, \text{ if } \begin{cases} 1.\ \mathtt{y}{\neq}\mathtt{x} \ \wedge \ w{\not\le}r \ \wedge \ r{\not\le}w \ \wedge \\ 2.\ Alias?(\mathtt{x},\mathtt{y},w) \end{cases} \\ false \quad \text{otherwise} \end{cases}$$

The predicate $FlowDep?$ uses the predicate $Alias?(x, y, w)$ to decide whether `x` and `y` are aliased at the moment the action w occurs in the Java event space. This last predicate is defined as the disjunction of the predicate $AliasAux?$ with the parameters swapped. The predicate $AliasAux?(\mathtt{y},\mathtt{x},w)$ checks whether, at the moment w occurs, `y` references the same address as `x`, as a consequence of an assignment to `y` from an alias of `x`. $AliasAux?(\mathtt{y},\mathtt{x},w)$ holds if (*i.*) `y` is written to by `x` — w_2:**write**$(\mathtt{y}, ref_{\mathtt{x}})$ — and the reference of `y` is not modified afterward by any w_1: **write** to some reference $ref_{\mathtt{t}}$ which is not alias of `x`, or (*ii.*) `y` is written to by a `z` other than `x`, and `z` was an alias of `x` before w_2 occurred.

$$Alias?(\mathtt{x},\mathtt{y},w) = AliasAux?(\mathtt{x},\mathtt{y},w) \ \vee \ AliasAux?(\mathtt{y},\mathtt{x},w)$$

$$\begin{aligned} &AliasAux?(\mathtt{y},\mathtt{x},w{:}\,\mathbf{write}) = \\ &(\exists w_2{:}\,\mathbf{write}(\mathtt{y},ref_{\mathtt{x}}).w_2{\le}w \wedge \neg\exists w_1{:}\ \mathbf{write}(\mathtt{y},ref_{\mathtt{t}}).w_2{\le}w_1{<}w \wedge \neg Alias?(\mathtt{x},\mathtt{t},w_1)) \ \vee \\ &(\exists w_2{:}\,\mathbf{write}(\mathtt{y},ref_{\mathtt{z}}).w_2{<}w \wedge \mathtt{z}{\neq}\mathtt{x} \wedge \\ &\qquad\qquad \neg\exists w_1{:}\,\mathbf{write}(\mathtt{y},ref_{\mathtt{t}}).w_2{<}w_1{<}w \wedge \neg Alias?(\mathtt{x},\mathtt{t},w_1) \wedge Alias?(\mathtt{y},\mathtt{z},w_2)) \end{aligned}$$

Slice sets are formalized by S_C below, where an event w labeled with an action is considered to be in the *carrier* of a Java event space η, $w \in carrier(\eta)$, if the event is related to itself. When S_C is applied to η and an r:**read**(`x.i`), it returns the set of events w:**write**(`y.i`) in the *carrier* of η such that the predicate $FlowDep?$(**write**(`y.i`), **read**(`x.i`)) holds.

$$S_C(\eta, r{:}\,\mathbf{read}(\mathtt{x.i})) = \{\ w{:}\,\mathbf{write}(\mathtt{y.i}) | w \in carrier(\eta) \wedge FlowDep?(w,r)\ \}$$

Example 2 Given the *slicing criterion* $C = \{\ \mathbf{read}(\mathtt{x.i}),\ \mathbf{read}(\mathtt{x.j})\ \}$ and the event space in Figure 1, $S_C(\mathbf{read}(\mathtt{x.i})) = \{\ \mathbf{write}(\theta_1, \mathtt{x.i}, 7), \mathbf{write}(\theta_1, \mathtt{y.i}, 5)\ \}$ and $S_C(\mathbf{read}(\mathtt{x.j})) = \{\ \mathbf{write}(\theta_1, \mathtt{x.j}, 5)\ \}$. Therefore:

$$S_C = \{\ \mathbf{write}(\theta_1, \mathtt{x.i}, 7),\ \mathbf{write}(\theta_1, \mathtt{x.j}, 5),\ \mathbf{write}(\theta_1, \mathtt{y.i}, 5)\ \}$$

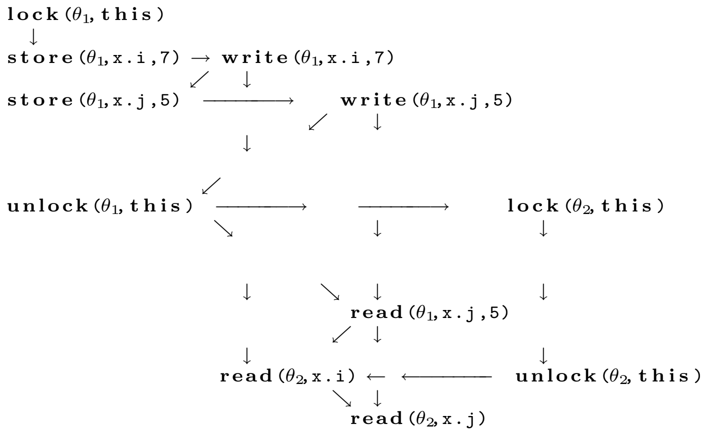

Fig. 2. Sliced event space

Definition 1 below formalizes the construction of residual slices. We are not going in details on this definition here, but want to indicate that Items (*i.*) through (*iv.*) of this definition rely on the formalization of the JMM [2, 10]. For instance, Item (*ii.*) comes from the JMM well-formedness rule *"A thread is not permitted to write data from its working memory back to the main memory for no reason"*. In Definition 1, *storeof* is a function that pairs a **write** action with a unique preceding **store** action.

Definition 1 (residual slice set construction S_{Cr}) Given an event space η and a slice set S_C, the residual set S_{Cr} for S_C is constructed from S_C by adding actions to it as follows:

(*i.*) For each **write** action w: $\textbf{write}(\theta, l, v)$ in S_C, the only action s in η such that s: $\textbf{store}(\theta, l, v)$ and $s = \mathit{storeof}(w)$ is added to S_{Cr}.
(*ii.*) For each pair of **store** actions s: $\textbf{store}(\theta, l)$, s': $\textbf{store}(\theta, l)$ in η such that $s \neq s'$ and $s \leq s'$, every action a: $\textbf{assign}(\theta, l)$ in η such that $s \leq a \leq s'$ is added.
(*iii.*) Each **lock** and **unlock** actions in η are added to S_{Cr}.
(*iv.*) For each **write** action w: $\textbf{write}(l)$ in S_C, all **read** actions r: $\textbf{read}(l)$ in η such that $w \leq r$ or $r \leq w$ are added to S_{Cr}.

Example 3 calculates the residual slice set S_{Cr} for S_C in Example 2.

Example 3 Given S_C as in Example 2, the residual slice set S_{Cr} for the event space in Figure 1 is:

$$\begin{aligned} S_{Cr} = \{\ & \textbf{write}(\theta_1, \texttt{x.i}, 7),\ \textbf{write}(\theta_1, \texttt{x.j}, 5),\ \ \textbf{write}(\theta_1, \texttt{y.i}, 5), \textbf{store}(\theta_1, \texttt{x.i}, 7), \\ & \textbf{store}(\theta_1, \texttt{x.i}, 5),\ \textbf{store}(\theta_1, \texttt{y.i}, 5),\ \ \textbf{read}(\theta_1, \texttt{x.j}, 5),\ \textbf{read}(\theta_2, \texttt{x.i}), \\ & \textbf{read}(\theta_2, \texttt{y.i}, 7),\ \textbf{read}(\theta_2, \texttt{x.j}),\ \ \textbf{lock}(\theta_1, \texttt{this}),\ \textbf{lock}(\theta_2, \texttt{this}), \\ & \textbf{unlock}(\theta_1, \texttt{this}), \textbf{unlock}(\theta_2, \texttt{this})\ \} \end{aligned}$$

Figure 2 presents the event space with events in S_{Cr} preserving the partial order relation in Figure 1.

4 Expressing Java event spaces as finite-state automata

To check the correctness of a dynamic system, we should be able to specify the kind of properties the system is expected to have. As dynamic systems can be modeled as finite-state transition systems, the formalism behind the specification should be appropriate to express properties about state transitions. *Temporal logic* is a particular formalism suitable to specify properties in terms of sequence of transitions between states in the system. The Computation Tree Logic (CTL) [4, 5] is one of the most commonly used temporal logic in model checking. Validity of CTL formulae depends only on the current state of the transition system, this is the reason why CTL formulae are referred to as *state formulae* in literature. CTL formulae are formed of *path quantifiers* and *temporal operators*. Path quantifiers specify that all paths, `A`, or some paths, `E`, starting at some *initial state* have a certain property. Four basic operators exist: `X` (*next*), which requires a property to hold at the second state of the path; `G` (*globally*) requires a property to hold at every state along the path; `F` (*future*) requires a property to hold at some states on the path; and `U` (*until*), combining two properties, which requires that the second property holds at some state along the path and the first holds in any preceding state. CTL requires that each use of a temporal operator be immediately proceeded by the use of a path quantifier. Hence, valid formulae in CTL are in the shape of $\mathtt{EX}\phi$, $\mathtt{EG}\phi$, $\mathtt{EF}\phi$, $\mathtt{E}[\phi_1\mathtt{U}\phi_2]$, $\mathtt{AX}\phi$, $\mathtt{AG}\phi$, $\mathtt{AF}\phi$, and $\mathtt{A}[\phi_1\mathtt{U}\phi_2]$.

We are interested in proving that, after the program slice procedure presented in Section 3 is applied to a Java event space, the sliced Java event space verifies the same CTL properties (when the next operator is not considered) as the original Java event space. To prove that, Java event spaces must be formalized as finite-state automata. In the following we present such a formalization in PVS [8]. First Java event spaces are modeled.

Java event spaces. Predicate `IsEventSpace?` below formalizes Java event spaces. A Java event space `E` is an `evtrelation`, *i.e.* a set of pairs of events, that respects the (17) well-formedness rules regarding the JMM enunciated in [7], that is is a partial order — *i.e.* that is reflexive, antisymmetric and transitive — and that has a finite history of elements preceding any event — `FiniteHistory?(E)`. This last predicate holds if for every event `e` in the *carrier* of `E` only a finite number of elements preceding it exists.

```
IsEventSpace?(E:evtrelation) : bool =
  rule1?(E) ∧ ··· ∧ rule17?(E) ∧
  reflexive?(E) ∧ antisymmetric?(E) ∧ transitive?(E) ∧ FiniteHistory?(E)

FiniteHistory?(E:evtrelation) : bool =
 ∀(e:event): is_finite({(d:event)|carrier(E)(e) ∧ carrier(E)(d) ∧ E(d,e)})
```

Java event spaces as finite-state automata. We first define a `store` as an association of right values `rval` to left values `lval`. Then, states are defined as records having two fields: a finite history h of events occurring before reaching the current state, and a store σ which is updated as events in the history occur. Initial states of the finite-state automaton have an `empty?` history of events and each element of the store has a default value `rdefault`.

```
store: TYPE = [lval → rval]
state: TYPE = [# h: (is_finite[event]), σ: store #]
InitialState: [state → bool] =
  λ(s:state): empty?(s‘h) ∧ s‘σ = λ(l:lval): rdefault
```

Predicate `NextState(E)` below decides whether a one-step transition between two states `s` and `s1` exists; `E` represents the event space to be expressed as a transition system. Formally expressed, `NextState(E)` holds for two states `s` and `s1` if their histories differ by a *single* element `e`, which moreover must belong to the *carrier* of `E`. Additionally, each element `f` in the *carrier* of `E` happening before `e`[1] must be in the history of `s`, and the history and store of `s1` can be obtained respectively from the history and store of `s` when considering only the effect produced by the event `e`. Notice that only `Write` events affect the store. This respects the definition of S_c in Section 3 where only `Write` events are retained in the sliced event space.

```
NextState(E:(IsEventSpace?)): [state,state → bool] =
 λ(s:state,s1:state):
  let {e} = s1‘h \s‘h in
    carrier(E)(e) ∧
    (∀(f:event): carrier(E)(f) ∧ E(f,e) ∧ f/=e ⇒ s‘h(f)) ∧
    s1‘h = s‘h ∪{e} ∧
    s1‘σ = cases e of Write(t,l,r): s‘σ with [l:=r] else s‘σ endcases
```

We can now define whether a trace `tr`, *i.e.* an infinite sequence of states indexed by natural numbers, constitutes a path in a finite-state automaton. First, the initial state of the trace must be an `InitialState`, and a transition between each pair of successive elements `i`, `i+1` of the trace should exist.

```
trace: TYPE = [nat → state]
Path(E:(IsEventSpace?)): [trace → bool] =
 λ(tr:trace): InitialState(tr(0)) ∧ ∀(i:nat): NextState(E)(tr(i),tr(i+1))
```

Figure 3(b) shows the states transitions of the Java event space in Figure 3(a) after slicing (removing) events `a`, `b`, `c` and `d`. Symbols $\prec\succ$ stand for records of type `state`. Thus, `NextState(E)`$(\prec\{\mathtt{e}_1,\mathtt{e}_2,\mathtt{e}_3\},\sigma_3\succ,\prec\{\mathtt{e}_1,\mathtt{e}_2,\mathtt{e}_3,\mathtt{e}\},\sigma_4\succ)$, for example. Lemmas below follow directly from the definitions of `NextState`, `trace` and `Path`, where `Program_Slice(E,C)` stands for the program slice of the Java event space `E` with respect to the slicing criterion `C`.

[1] `E(e1,e2)` corresponds to the notation $e_1 \leq e_2$ in the event space `E` used before. Symbol `/=` denotes inequality in PVS, and `s‘h` stands for field h of `s`.

$$
\begin{tikzcd}[column sep=small]
 & \mathtt{e}_1 \arrow[d] & \\
 & \mathtt{e}_2 \arrow[d] & \\
\mathtt{a} \arrow[dr] & \mathtt{e}_3 \arrow[d] & \mathtt{b} \arrow[dl] \\
\mathtt{c} \arrow[dr] & \mathtt{e} \arrow[d] & \mathtt{d} \arrow[dl] \\
 & \mathtt{f} &
\end{tikzcd}
$$

(a) Trace in the event space

0 $\prec \varnothing, \sigma_0 \succ$
1 $\prec \{\mathtt{e}_1\}, \sigma_1 \succ$
2 $\prec \{\mathtt{e}_1, \mathtt{e}_2\}, \sigma_2 \succ$
3 $\prec \{\mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3\}, \sigma_3 \succ$
4 $\prec \{\mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3, \mathtt{e}\}, \sigma_4 \succ$
5 $\prec \{\mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3, \mathtt{e}, \mathtt{f}\}, \sigma_5 \succ$

(b) Sliced trace

0 $\prec \varnothing \succ$
1 $\prec \{\mathtt{e}_1\} \succ$
2 $\prec \{\mathtt{e}_1, \mathtt{e}_2\} \succ$
3 $\prec \{\mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3\} \succ$
4 $\prec \{\mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3, \mathtt{e}, \mathtt{a}, \mathtt{b}\} \succ$
5 $\prec \{\mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3, \mathtt{e}, \mathtt{a}, \mathtt{b}, \mathtt{f}, \mathtt{c}, \mathtt{d}\} \succ$

(c) Extended trace

0 $\prec \varnothing \succ$
1 $\prec \{\mathtt{e}_1\} \succ$
2 $\prec \{\mathtt{e}_1, \mathtt{e}_2\} \succ$
3 $\prec \{\mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3\} \succ$
4 $\prec \{\mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3, \mathtt{a}\} \succ$
5 $\prec \{\mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3, \mathtt{a}, \mathtt{b}\} \succ$
6 $\prec \{\mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3, \mathtt{a}, \mathtt{b}, \mathtt{e}\} \succ$
7 $\prec \{\mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3, \mathtt{a}, \mathtt{b}, \mathtt{e}, \mathtt{c}, \} \succ$
8 $\prec \{\mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3, \mathtt{a}, \mathtt{b}, \mathtt{e}, \mathtt{c}, \mathtt{d}\} \succ$
9 $\prec \{\mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3, \mathtt{a}, \mathtt{b}, \mathtt{e}, \mathtt{c}, \mathtt{d}, \mathtt{f}\} \succ$

(d) Original trace

Fig. 3. Reconstruction of traces in the event space

The first lemma says that the history for the first state (index 0) of any sliced trace[2] is empty; the second lemma states that histories of successive states differ by a single event; additionally, the third lemma says that these histories grow. The last lemma combines the third and the fourth lemmas.

```
sliced_traces_are_empty_initially: lemma
 ∀(E:(IsEventSpace?),C:setof[event],tr:trace):
  Path(Program_Slice(E,C))(tr) ⇒ empty?(tr(0)‘h)

sliced_traces_make_single_steps: lemma
 ∀(E:(IsEventSpace?),C:setof[event],tr:trace,i:nat):
  Path(Program_Slice(E,C))(tr) ⇒
   subset?(tr(i)‘h,tr(i+1)‘h) ∧ singleton?(tr(i+1)‘h \tr(i)‘h)

sliced_traces_are_strict_subsets: lemma
 ∀(E:(IsEventSpace?),C:setof[event],tr:trace,i:nat):
  Path(Program_Slice(E,C))(tr) ⇒ strict_subset?(tr(i)‘h,tr(i+1)‘h)

sliced_traces_as_increments_the : lemma
 ∀(E:(IsEventSpace?),C:setof[event],tr:trace,i:nat):
   Path(Program_Slice(E,C))(tr) ⇒
     tr(i+1)‘h = add(the(tr(i+1)‘h \tr(i)‘h),tr(i)‘h)
```

[2] A sliced trace is a trace in a program slice.

Evaluating properties. The evaluation of CTL properties follows directly from the standard definitions of CTL operators. For example, given a Java event space E and a state s, property EGϕ holds in s, if a trace tr starting at s — tr(0)=s — and being a path — Path(E)(tr) — exists, such that ϕ is true along the trace — $\forall$(j : nat) : Eval(ϕ)(tr(j)). Further, a property Holds provided that it holds at every initial state.

```
Sem(E:(IsEventSpace?)): [property → [state → bool]] =
 λ(prop:property)(s:state):
 cases prop of
  EG(P): ∃(tr:trace): tr(0)=s ∧ Path(E)(tr) ∧ ∀(j:nat): Eval(P)(tr(j)),
   ...
 endcases

Holds(E:(IsEventSpace?)): [property → bool] =
 λ(prop:property): ∀(s:state): InitialState(s) ⇒ Sem(E)(P)(s)
```

5 Program_Slice is CTL property-preserving

We want to prove that, for any proper slicing criterion C, if a CTL property prop holds in the whole Java event space E, then prop holds in the sliced Java event space Program_Slice(E,C), and vice-versa. This is expressed in the following two theorems respectively:

```
preserving_slice_fi : theorem
 ∀(E:(IsEventSpace?),C:setof[event],prop:property):
  Holds(E)(prop) ⇒ Holds(Program_Slice(E,C))(prop)

preserving_slice_if : theorem
 ∀(E:(IsEventSpace?),C:setof[event],prop:property):
  Holds(Program_Slice(E,C))(prop) ⇒ Holds(E)(prop)
```

First, notice that slicing can not preserve properties constructed with the aid of the CTL operators EX and AX because slicing does not preserve *next* states. Second, when using Holds(R)(prop) in the definition of preserving_slice_fi and preserving_slice_if above, a proof of the following lemma, stating that sliced Java event spaces are still Java event spaces, must be first provided.

```
slice_sets_are_event_spaces : lemma
 Program_Slice(E:(IsEventSpace?),C:setof[event]) has_type (IsEventSpace?)
```

This lemma ensures that expressing sliced Java event spaces as finite-state automata according to Section 4 is still valid. We will not focus on the proof of this last lemma here, but will use it in the proof of the second theorem above. This theorem has been proved for the existential operators EG, EU and EF[3]. Our approach considers the reconstruction of traces which incorporate all those

[3] Additionally, the first theorem has been proved for the universal CTL operators AG, AU and AF.

events in the event space removed when slicing. For one, this approach allows the making of the whole proof process easier and secondly, if we know that original traces verify the same properties as sliced traces, we are sure that our program slice is correct in the sense that only those elements that do not change the validity of underlying properties were removed when slicing. This construction is accomplished in two steps.

Constructing original traces from sliced traces. Firstly, we construct *extended traces* from sliced traces `tr`, which modify the history of every state in `tr` in such a way that those events in the event space who relate to any event in the history are added to it (`extended_trace` below gives a precise definition of extended traces). Note that stores on extended traces coincide respectively with stores on sliced traces; that is in accordance with the fact that, when slicing, we rule out only those events that do not affect the store and hence do not affect the validity of the CTL property that is checked. Figure 3(c) shows the extended trace constructed from the sliced trace presented in Figure 3(b). We have intentionally omitted stores as part of states.

```
extended_trace(E:(IsEventSpace?),C:setof[event]):
                                 [(Path(Program_Slice(E,C))) → trace] =
 λ(tr:(Path(Program_Slice(E,C)))) (n:nat):
  (# h:={ e:event | Carrier(E)(e) ∧
                   ∃(a:event): Carrier(E)(a) ∧ tr(n)‘h(a) ∧ E(e,a) },
    σ:= tr(n)‘σ #)
```

We want to make single steps between consecutive indexes, *i.e.* histories between consecutive states should differ by a single element only. However, single steps are not provided by the definition of `extended_trace` (see histories for indexes 3 and 4 in Figure 3(c) for example). To achieve this singleness, events in the history of every state on the extended trace are spelled out, making up *original traces* (see definition below). Figure 3(d) presents the original trace for the extended trace in Figure 3(c). Notice that if the history at index 4 in Figure 3(c) is spelt out, one sole element between `a` or `b` should be chosen from the history first; `e` cannot be chosen because it requires that both `a` and `b` occur before. To make this choice, the *least* between `a` or `b` can be selected; but since, Java event spaces do not provide *total* orders in general, the least between `a` and `b` might not be defined. Assume that such a function `spell_history(E(IsEventSpace?))(k:nat, S:setof[(Carrier(E))]`, spelling the `k` least elements of `S`, exists; as well as `spell_store(E:(IsEventSpace?))(k:nat,S:setof[event]) (st:strstate)`, which spells the `k` least elements of `S` and return `st` after making it `k` single updates.

From the definition of `original_trace` below, given an index `n`, if `n=0` then `original_trace` returns `tr(0)`. If `n>0`, then `original_trace` takes the minimum index `m` such that `Card(etr(m)‘`h`)>=n`, where `Card` is the standard cardinality function for sets. If `Card(etr(m)‘`h`)` is `n`, then the original trace coincides with the extended trace; otherwise, `n-p` — where `p` is `Card(etr(m-1)‘`h`)` — events are spelt from the difference between `Card(etr(m)‘`h`)` and `Card(etr(-m-1)‘`h`)`. The same is done for the store.

```
original_trace(E:(IsEventSpace?),C:setof[event]):
                                [(Path(Program_Slice(E,C))) → trace] =
 λ(tr:(Path(Program_Slice(E,C))))(n:nat):
  let etr = extended_trace(E,C)(tr) in
   if n=0 then tr(0) else
    let m = min(λ(x:nat): Card(etr(x)‘h)>=n) in
     if Card(etr(m)‘h)= n then etr(m) else
      let D = etr(m)‘h \etr(m-1)‘h, p = Card(etr(m-1)‘h) in
       (# h := union(etr(m-1)‘h,spell_history(E)(n-p,D)),
          σ := spell_store(E)(n-p,D)(etr(m-1)‘σ) #)
     endif
   endif
```

The approach used to reconstruct traces (paths) on the original system from traces in the reduced system is general, so it can be extended to other state-space reduction techniques, provided that some *sufficient* conditions are verified. Note that the definition of `original_trace` depends on the proper definition of `extended_trace`. The three lemmas below summarize those sufficient conditions. The first lemma says that the *minimum* index `i` for which the cardinality of `extended_trace` is greater than or equal than `n`, for some `n`, always exists. The second lemma says that this cardinality is always positive for any positive `n`. And the third lemma says that `extended_trace` histories grow.

```
etr_nonempty_n_positive: lemma
 ∀(E:(IsEventSpace?),C:setof[event],tr:(Path(Program_Slice(E,C))),n:nat):
  let S=λ(i:nat): Card(extended_trace(E,C)(tr)(i)‘h) >= n in
   nonempty?[nat](S)

etr_min_positive: lemma
 ∀(E:(IsEventSpace?),C:setof[event],tr:(Path(Program_Slice(E,C))),n:nat):
  let S=λ(i:nat): Card(extended_trace(E,C)(tr)(i)‘h) >= n in
   n>0 ⇒ min(S)>0

etr_n_minus_p_nonnegative: lemma
 ∀(E:(IsEventSpace?),C:setof[event],tr:(Path(Program_Slice(E,C))),n:nat):
  let S=λ(i:nat): Card(extended_trace(E,C)(tr)(i)‘h) >= n in
   let m=min(S) in let p=Card(extended_trace(E,C)(tr)(m-1)‘h) in
    n>0 ⇒ n-p>=0
```

Further, the lemmas below summarize some properties about original traces. The first lemma follows from the proper definition of `spell_history`; the second from the definition of the first lemma, and the third from the first lemma and some results on sets theory.

```
otr_makes_single_steps : lemma
 ∀(E:(IsEventSpace?),C:setof[event],tr:trace,i:nat):
  Path(Program_Slice(E,C))(tr) ⇒
  subset?(original_trace(E,C)(tr)(i)‘h,original_trace(E,C)(tr)(i+1)‘h) ∧
  singleton?(original_trace(E,C)(tr)(i+1)‘h \original_trace(E,C)(tr)(i)‘h)
```

```
otr_are_strict_subsets : lemma
 ∀(E:(IsEventSpace?),C:setof[event],tr:trace,i:nat) :
  Path(Program_Slice(E,C))(tr) ⇒
  strict_subset?(original_trace(E,C)(tr)(i)‘h,
                 original_trace(E,C)(tr)(i+1)‘h)

otr_as_increments_the : lemma
 ∀(E:(IsEventSpace?),C:setof[event],tr:trace,i:nat) :
  Path(Program_Slice(E,C))(tr) ⇒
  original_trace(E,C)(tr)(i+1)‘h =
    add(the(original_trace(E,C)(tr)(i+1)‘h \original_trace(E,C)(tr)(i)‘h),
        original_trace(E,C)(tr)(i)‘h)
```

Now, we go into the proof of the following theorem, which summarizes the process of constructing original traces described before.

```
constructing_original_traces_from_traces : theorem
 ∀(E:(IsEventSpace?),C:setof[event],tr:trace):
  Path(Program_Slice(E,C))(tr) ⇒ Path(E)(original_trace(E,C)(tr))
```

Theorem 1 (constructing_original_traces_from_traces) Because of the following equivalence:

`∀(E:(IsEventSpace?),tr:trace): Path(E)(tr) ⇔ ∀(i:nat): PathUpTo(E)(tr)(i),`

where `PathUpTo` is given by:

```
PathUpTo(E:(IsEventSpace?))(tr:trace)(n:nat) : RECURSIVE bool =
 if n=0 then InitialState(tr(0))
 else PathUpTo(E)(tr)(n-1) ∧ NextState(E)(tr(n-1),tr(n)) endif
measure n
```

the proof reduces to:

`Path(Program_Slice(E,C))(tr) ⇒ ∀(i:nat): PathUpTo(E)(tr)(i)`

Then, by induction on `i`, the base case becomes:

`Path(Program_Slice(E,C))(tr) ⇒ PathUpTo(E)(tr)(0)`

When expanding `Path` and `PathUpTo` definitions, the base case reduces to:

```
InitialState(tr(0)) ∧ ∀(i:nat): NextState(Program_Slice(E,C))(tr(i),tr(i+1))
⇒ InitialState(original_trace(E,C)(tr)(0))
```

Because `original_trace(E,C)(tr)(0)` is `tr(0)`, this goal reduces trivially. For the case `i=k` we have:

```
Path(Program_Slice(E,C))(tr) ∧ PathUpTo(E)(original_trace(E,C)(tr))(k)
⇒ PathUpTo(E)(original_trace(E,C)(tr))(k+1)
```

Then, because `PathUpTo(E)(original_trace(E,C)(tr))(k+1)` can be expressed as the conjunction between `PathUpTo(E)(original_trace(E,C)(tr))(k)` and `NextState(E)(original_trace(E,C)(tr)(k), original_trace(E,C)(tr)(k + 1))`, the case `i=k` reduces to:

```
Path(Program_Slice(E,C))(tr) ∧ PathUpTo(E)(original_trace(E,C)(tr))(k)
⇒ NextState(E)(original_trace(E,C)(tr)(k),original_trace(E,C)(tr)(k+1))
```

When expanding the definition of `NextState`, the proof of the `i=k` reduces to three sub-cases, namely, (*ii.a*) which states that if `original_trace` makes a transition from state at index `k` to state at index `k+1` using `the` single event `e` in the difference between the state histories, then any event `f` occurring before `e` must belong to the history of `original_trace` at index `k`.

```
Path(Program_Slice(E,C))(tr) ∧ PathUpTo(E)(original_trace(E,C)(tr))(k) ⇒
 ∀(f:event):
 (carrier(E)(f) ∧
  E(f,the(original_trace(E,C)(tr)(k+1)‘h \original_trace(E,C)(tr)(k)‘h)) ∧
  f /= the(original_trace(E,C)(tr)(k+1)‘h \original_trace(E,C)(tr)(k)‘h)
 ) ⇒ original_trace(E,C)(tr)(k)‘h(f)
```

(*ii.b*) which states that for any indexes `k` and `k+1` in `original_trace`, the history at index `k+1` can be obtained from the history at index `k` when adding `the` event in the difference.

```
Path(Program_Slice(E,C))(tr) ∧ PathUpTo(E)(original_trace(E,C)(tr))(k) ⇒
original_trace(E,C)(tr)(k+1)‘h =
 add(the(original_trace(E,C)(tr)(k+1)‘h \original_trace(E,C)(tr)(k)‘h),
     original_trace(E,C)(tr)(k)‘h)
```

and (*ii.c*) which states something similar to (*ii.b*), but considering stores instead of histories:

```
Path(Program_Slice(E,C))(tr) ∧ PathUpTo(E)(original_trace(E,C)(tr))(k) ⇒
original_trace(E,C)(tr)(k+1)‘σ =
 cases the(original_trace(E,C)(tr)(k+1)‘h \original_trace(E,C)(tr)(k)‘h) of
  Write(t,l,r): original_trace(E,C)(tr)(k)‘σ with [l:=r]
 else original_trace(E,C)(tr)(k)‘σ endcases
```

We are not going into details about the proof of these three sub-cases here; we just want to say that the proof of (*ii.b*) is based on the correct definition of `spell_history`; (*ii.c*) on the correct definition of `spell_store`; and (*ii.a*) on the correct definition of both `spell_history` and `spell_store`.

Theorem 2 uses lemma `constructing_original_traces_from_traces` to prove `preserving_slice_if`.

Theorem 2 (`preserving_slice_if`**)** After expanding the definition of `Holds` and `Sem` and doing induction on `prop`, `preserving_slice_if` reduces to:

```
( ∀(s:state): InitialState(s) ⇒
   (∃(tr: trace): tr(0) = s ∧ Path(Program_Slice(E,C))(tr) ∧
                             ∀(j:nat): Eval(P)(tr(j)‘σ)) ) ⇒
( ∀(s:state): InitialState(s) ⇒
  (∃(tr: trace): tr(0) = s ∧ Path(E)(tr) ∧ ∀(j:nat): Eval(P)(tr(j)‘σ)) )
```

Then, when considering the same state `s` in the hypothesis as in the goal, the theorem reduces to:

```
( InitialState(s) ∧
  ∃(tr: trace):
   tr(0) = s ∧ Path(Program_Slice(E,C))(tr) ∧ (∀(j:nat): Eval(P)(tr(j)‘σ))
) ⇒
( ∃(tr: trace): tr(0) = s ∧ Path(E)(tr) ∧ (∀(j:nat): Eval(P)(tr(j)‘σ)) )
```

To instantiate the goal, `original_trace(E,C)(tr)` is used. Thereafter, three subgoals are to be proved, namely:

```
(i.)   tr(0)=s ⇒ original_trace(E,C)(tr)(0)=s
(ii.)  Path(Program_Slice(E,C))(tr) ⇒ Path(E)(original_trace(E,C)(tr))
(iii.) ∀(j:nat): Eval(P)(tr(j)‘σ) ⇒
                 ∀(j:nat): Eval(P)(original_trace(E,C)(tr)(j)‘σ)
```

Since `original_trace(E,C)(tr)(0)` is `tr(0)`, (*i.*) is trivially verified; (*ii.*) reduces from Theorem 1; to prove (*iii.*), `original_traces_are_well_formed` below is used. This lemma says that for all index `j` in the original trace exists an index `i` in the sliced trace such that any event in the difference is unimportant, *i.e.* it does not modify the validity of the property that is checked. This last lemma constitutes a new *sufficient* condition.

```
original_traces_are_well_formed : lemma
 ∀(E:(IsEventSpace?),C:setof[event],tr:trace) :
  Path(Program_Slice(E,C))(tr) ⇒
  ∀(j:nat):∃(i<=j): subset?(tr(i)‘h,original_trace(E,C)(tr)(j)‘h) ∧
                    ∀(b:event): (original_trace(E,C)(tr)(j)‘h \tr(i)‘h)(b)
                                ⇒ unimportant_event(C)(b)
```

6 Conclusion

The full PVS formalization presented here consists of 1800 lines of code, including 32 theorems. This formalization shows how theorem proving techniques can effectively be used to prove general properties about state-space reduction algorithms. We presented the formalization of a slicing algorithm introduced previously in [1], and the proof that this algorithm preserves a subset of the properties that can be modeled using Computation Tree Logic (CTL): *next-state* properties formed of `AX`, `EX` CTL operators are not preserved under slicing.

Furthermore, during the proof some *sufficient* conditions were outlined to extend the two-steps path reconstruction proof approach to other proofs that involve proving CTL property-preserving under similar state-space reduction techniques. In particular, in future work, we are interested in exploring how partial order reduction techniques [6] can be employed to reduce the number of states generated in `MDD`s based symbolic state generation techniques [3]. And we are interested in the correctness proofs involved in that reduction.

The Java Memory Model (JMM) as specified in Chapter 17 of the Java Language Specification presents some inconsistencies as highlighted by *W. Pugh* in [9]. A new document of Java Specification Requests (JSR-133) (see `http://-www.cs.umd.edu/~pugh/java/memoryModel/`) has been produced to fix these inconsistencies. This document is part of the most recent Tiger 5.0 release of Java. We consider that main results presented here are still valid for these new specifications: our results apply not only for slicing techniques in the context of Java, but for reduction techniques in general.

Acknowledgements. We thank anonymous referees for useful feedback and Gerald Lüttgen for his comments on the previous of this paper. This work has been partially supported by the EPSRC under grant GR/S86211/01.

References

1. N. Cataño. Slicing event spaces: Towards a Java programs checking framework. In Thomas Arts and Wan Fokkink, editors, *FMICS: Eighth International Workshop on Formal Methods for Industrial Critical Systems*, volume 80 of *Lecture Notes in Computer Science*, Trondheim, Norway, Jun. 5–7 2003. Elsevier.
2. P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An event-based structural operational semantics of multi-threaded Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 157–200. Springer-Verlag, 1999.
3. G. Ciardo, R. Marmorstein, and R. Siminiceanu. Saturation unbound. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003)*, volume 2619 of *Lecture Notes in Computer Science*, pages 379–393, Warsaw, Poland, 2003. Springer-Verlag.
4. E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings of Logics of Programs*, Lecture Notes in Computer Science, pages 52–71, Yorktown Heights, New York, May 1981.
5. E.A. Emerson and J.Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. In *Proceedings of 14th Symposium on Theory of Computing (STOC'82)*, pages 169–180, San Francisco, CA, May 1982. ACM.
6. Patrice Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032. Springer-Verlag Inc., New York, NY, USA, 1996.
7. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, 2000.
8. S. Owre, N. Shankar, J.M. Rushby, and D.W.J Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI, Menlo Park, CA, Nov. 2001.
9. W. Pugh. Fixing the Java memory model. In *Proceedings of the ACM 1999 conference on Java Grande*, pages 89–98. ACM Press, 1999.
10. B. Reus and T. Hein. Towards a machine-checked Java specification book. In *TPHOL, Theorem Proving in Higher Order Logics*, volume 1869 of *Lecture Notes in Computer Science*, pages 480–497. Springer-Verlag, 2000.