Slicing Event Spaces: Towards a Java Programs Checking Framework

Néstor Cataño [1]

INRIA, France

Abstract

Java event spaces are partial orders of memory and thread actions as generated by a multi-threaded Java program in execution. This paper shows how standard techniques of slicing can be used to reduce the size of Java event spaces. Furthermore, we face the problem that arises when two or more variables of an event space are *aliased* and we outline an algorithm that goes through an event space and calculates aliases of variables. We incorporate this algorithm in the calculation of the program program slice.

Key words: Java, Java event spaces, multi-threading, slicing, checking, aliasing.

1 Introduction

The Java programming language has quickly become the most popular programming language for Internet applications. Hence several efforts have been put in formally specifying its semantics [17,19] and to construct frameworks in which properties can be checked. These efforts can be seen as falling in two categories. Firstly, some tools and specification languages for these tools have been developed to face problems concerning the *sequential* part of Java.

As an example of tools working with Java sequential programs we mention the LOOP tool [11], which transforms sequential Java programs and their specifications into PVS theories which are then proved using the PVS theorem prover [13]. Further, *Gary T. Leavens et al.* at Iowa State University have developed a runtime checker for Java programs called JML using as specification language the Java Modeling Language [9]. A tool tightly related with the JML runtime checker is ESC/Java [6], a static checker for Java programs using as specification language a subset JML. The ESC/Java tool tries to

[1] `Nestor.Catano@sophia.inria.fr`

check that a program satisfies its annotations by using a dedicated automatic theorem prover. Furthermore, Chase [3], a static checker for so-called JML `assignable` clauses, has been shown to be an efficient tool to check frame conditions in sequential Java programs. This tool has been successfully applied to a Java Card application case study [2].

Secondly, tools checking specifications of *multi-threaded* Java programs have been developed. However, problems concerning multi-threading are evidently more difficult. Multiple errors such as deadlock and race conditions can arise from the interaction among multiple threads and the main memory. Also, the behavior of a multi-threaded Java program is not deterministic, thus the errors are difficult to reproduce, making the debugging work harder. Some tools have been developed to overcome these problems. The Bandera toolkit [1], developed by *Dwyer*, *Hatcliff et al.* in the SAnToS laboratory, is an interesting framework in which several model checkers such as SPIN and SMV are integrated together. In Bandera a big effort has been put in maintaining trace errors and in reducing the size of programs by using standard techniques of slicing and abstraction. One of the problems using Bandera though, is that it does not handle recursive methods and exceptions in general.

When verifying programs using model checking techniques, an important aspect to consider is the reduction of the size of programs, since model checking techniques are only feasible in practice when the size of the underlying transition system and the domain of the concerned variables are all finite. Slicing techniques appear thus as a means of reducing the size of programs. In this paper we are concerned with the problem of finding a representation (model) of multi-threaded Java programs that considers *recursion* and *exceptions handling* constructions, and with drastically reducing the size of this representation using standard slicing techniques.

In a previous work, *Cenciarelli*, *Reus*, *Knapp*, *Wirsing* and *Hein* [5,10,15] formalized the Java memory model based on Chapter 17 of the Java language specification and defined an operational semantics to calculate the so-called *event spaces* for Java programs. These event spaces are partial orders of the actions performed by the main memory and by threads. We use event spaces as a representation for multi-threaded Java programs. An advantage of using event spaces as a model for Java programs compared to the work of Bandera described before, is that event spaces allow one to manage *recursive methods* and *exceptions handling* provided that the original Java program does not have an infinite execution. Although event spaces are finite, their size can increase unboundedly if for example the original Java program loops infinitely. We thus need to reduce their size in order to make formal verification feasible. This paper shows how standard slicing techniques can be used to reduce the size of the event spaces. In short, when slicing, we retain the events of the event space upon which the events of the slicing criterion depend, but respecting the underlying structure of event space. Also, we deal with the problem of *aliasing* that arises when two variables of the event space point to the same memory address. We outline an algorithm that goes through an event space and calculates aliasing dependences for relevant variables of the slicing criterion. We incorporate this algorithm as part of the definition of dependence.

The main contributions of this paper are the following: *i*. We show how slicing techniques can easily be adapted for the purpose of reducing the size of event spaces and thus of multi-threaded Java programs. *ii*. We deal with the problem of *aliasing*, presenting an algorithm which finds aliasing dependences and incorporating this algorithm in the calculation of the program slice. Event spaces can potentially be used to manage recursive methods and exceptions handling since there exist event space representations for these constructions, provided that the original program does not loop infinitely. We do not face this problem here but plan to go into this in future work.

Organization. Section 2 gives an overview of event spaces, presenting its formal definition, giving an example in which an event space for a multi-threaded Java program is generated and explaining the Java memory model. Section 3 defines how event spaces can be sliced, removing non-interesting events but preserving their structure. Section 4 deals with the problem that arises when variables in the event space are aliased, outlining an algorithm that finds aliases of relevant variables. Section 5 shows how events spaces can easily be represented as labeled transition systems and finally Section 6 gives conclusions and presents future work.

2 Event Spaces

Chapter 17 of the Java language specification [7] gives a detailed yet not formal specification of how a multi-threaded Java program should operate. This specification states that there exists a main memory that is shared by all the threads and that keeps a global copy of the values of the variables. It also says that each thread has its own local working memory which can keep a copy of variables of the main memory. As a thread executes code, it carries out a sequence of actions. A thread θ can use the value v of a variable l, $\mathbf{use}(\theta, l, v)$, or it can assign it a new value v, $\mathbf{assign}(\theta, l, v)$. When copying the value v of the variable l from the main memory to the working memory of θ, two actions must occur: a $\mathbf{read}(\theta, l, v)$ action performed by the main memory followed some unspecified time later by a $\mathbf{load}(\theta, l, v)$ action performed by the working memory. When copying the value v of the variable 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 some unspecified time later by a

write(θ, l, v) action performed by the main memory. Actions **lock**(θ, o) and **unlock**(θ, o) acquire and relinquish a lock on the object o on behalf of the thread θ.

An *event* represents the occurrence of some action either in the main memory or in the working memory of some thread, regardless of any concept of time. *Cenciarelli, Reus, Knapp, Wirsing* and *Hein* formalize the interaction of these events, making up so-called *Java event spaces* [4,5,10,15]. Formally expressed, a Java event space is a partial order $(X, \leq)$ with actions in X, partial order relation $\leq$ and satisfying rules of Appendix A which are the formalization of the Java memory model. We use the notation $x : M$ to indicate that x is an action done by the main memory, $x : T$ to indicate that x is an action done by some thread and $x : K$ to indicate that x is a lock action. Actions in M are named **read**(θ, l, v), **write**(θ, l, v), **lock**(θ, o) and **unlock**(θ, o), actions in T are named **load**(θ, l, v), **use**(θ, l, v), **assign**(θ, l, v), **store**(θ, l, v), **write**(θ, l, v), **lock**(θ, o) and **unlock**(θ, o), and actions in K are named **lock**(θ, o) and **unlock**(θ, o). In this context θ represents a thread and l and v are left and right values, where left values are 3-tuples composed by a memory address, an identifier and its type, and right values represent values of objects in memory: native type values and references.

According to the Java language specification, *read* and *write* actions for the same variable, thread actions for the same thread and *lock* and *unlock* actions on the same object must be totally ordered. The Java language specification introduces more similar rules described in words, e.g, *"A lock action by T on L may occur only if, for every thread S other than T, the number of preceding unlock actions by S on L equals the number of preceding lock actions by S on L"*. Appendix A presents the formalization of these rules as they are described in Chapter 17 of the Java language specification.

In the following we show an example of event space generation for the interaction of two threads executing two methods in parallel. Section 3 explains how slicing techniques can be used to reduce the size of an event space in a context without aliasing. Section 4 proposes an algorithm to detect aliasing of left values based on the static information provided by the event space and shows how to incorporate this algorithm in the calculation of a program slice.

Example 1 Suppose there exist two threads θ_1 and θ_2 executing respectively the methods `p()` and `q()` on some object `this` and whose code is shown below.

```
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;
          System.out.println("x.i:" +x.i +" x.j:" +x.j); }
```

Suppose as well that variables `x`, `y` and `z` are instances of some class with two instance variables `i` and `j` of type `int` and whose initial values are 0.

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 of the actions is represented by arrows. Since an order is anti-symmetric, the relation goes from top to bottom. Because of space constraints the reflexivity and transitivity of the partial order relation has not been sketched. Since 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 read and write actions for the variables `x.i`, `x.j`, `y.i` and `z.i` in columns 2 to 5 also form increasing chains.

In Java, a synchronized statement acquires a lock associated with an object on behalf of an executing thread in such a way that 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 object. In our example, we sketch the case when θ_1 acquires the lock on object `this` before θ_2.

After acquiring a 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` to execute the synchronized part of `q`(). Afterwards θ_1 and θ_2 continue to execute concurrently, however 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. Finally, we are interested in knowing the values of field `x.i` and `x.j` as seen by thread θ_2 (Column 2, Line 44 and Column 3, Line 46).

The next section presents how standard slicing techniques can be used to reduce the size of event spaces [18]. Section 4 redefines the notion of dependence in order to consider aliased variables.

3 Slicing Event Spaces in a Context without Aliasing

A *program slice* consists of the parts of a program that potentially affect the value computed at some points of interest, which depend on the property one

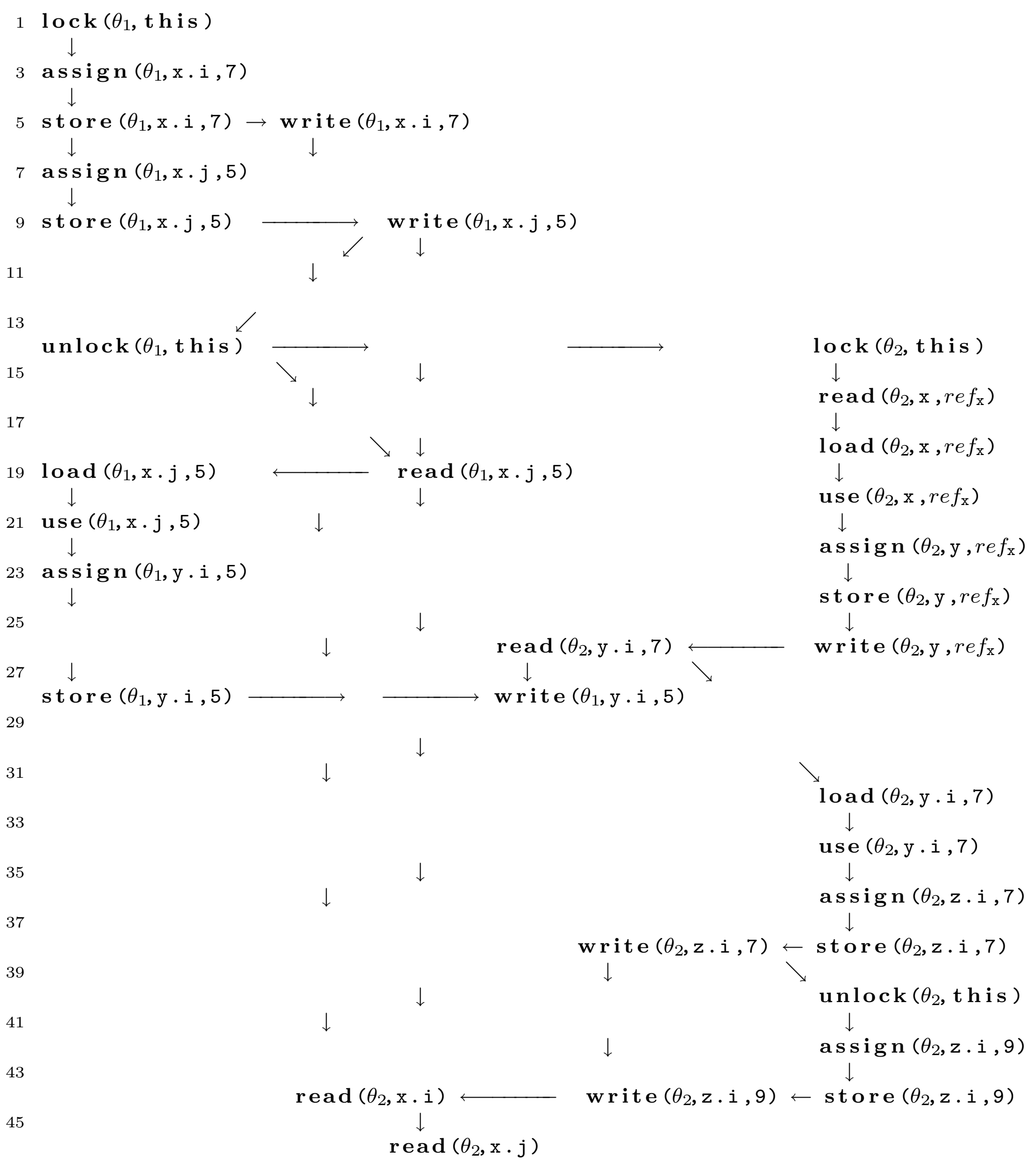

Fig. 1. An event space for Example 1

is interested in. These points of interest are called a *slicing criterion* and its variables are called *relevant variables.* Slicing techniques can be classified in two categories: *static slicing* techniques in which only statically available information is used for computing slices and *dynamic slicing* techniques which rely on test cases. We are not interested in defining specific test cases for event spaces, but interested in slicing event spaces based on its static information, so the work presented here relies on techniques in the first category.

We define a *slicing criterion* as a set $C = \{e_1, e_2, \cdots, e_n\}$ of *read* events representing the value of left values in certain points of interest for the program. A *control flow graph* (CFG) is a graph where a separate node for each statement of the program exists, as well as edges between nodes i and j indicating the possibility of going from i to j. One can see an event space as a CFG including information about the memory and thread actions generated for all possible executions of a multi-threaded program, with events as nodes and the partial order relation representing edges in the graph. A *program dependence graph* (PDG) is a directed graph with nodes corresponding with events and edges corresponding with dependences. We use a PDG approach to slicing, thus for constructing the program slice we find first the so-called *slice set* S_C of nodes in the CFG from which the nodes in C are reachable via the dependence $\xrightarrow{fd}$ defined below. In addition to elements in S_C, a *residual slice set* must be constructed that contains other events to preserve the structure of the event space. The event space formed of events in the residual slice set and having as order relation the partial order relation of the program restricted to the residual slice set, will make up the *program slice*.

When doing slicing, there exist several kind of dependences such as diverge dependence, interference dependence, synchronization dependence and control dependence and *data dependence* [8]. We are interested in a particular kind of data dependence called *flow dependence* whose definition is given below. Basically, this definition states that two nodes are dependent if the information *writing to* one of them is then *read from* the other node.

Definition 1 (flow dependence $p \xrightarrow{fd} q$) A node q is *flow dependent* on a node p, if p *writes* to a variable l that q *reads* and there is no action $w : \mathbf{write}(l)$ such that $p \leq w \leq q$.

Taking as basis the event space shown in Figure 1, Example 2 finds the slice set corresponding with the slicing criterion formed of the read actions in which we are interested.

Example 2 For the event space in Figure 1, we are interested in finding the flow dependences for nodes in the slicing criterion $C = \{\mathbf{read}(\texttt{x.i}),\ \mathbf{read}(\texttt{x.j})\}$, in which $\mathbf{read}(\texttt{x.i})$ refers to some event $\mathbf{read}(\theta, \texttt{x.i}, v)$ for some thread θ and right value v (Similar for $\mathbf{read}(\texttt{x.j})$). According to Definition 1 presented above, we have $\mathbf{write}(\theta_1, \texttt{x.i}, 7) \xrightarrow{fd} \mathbf{read}(\texttt{x.i})$ for the first element of the slicing criterion and $\mathbf{write}(\theta_1, \texttt{x.j}, 5) \xrightarrow{fd} \mathbf{read}(\texttt{x.j})$ for the second element. Therefore, the slice set for C is given by:

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

Besides the events in the slice set S_C, the residual slice set must contain other elements in order to preserve the structure of event space. From the rules formalizing the Java memory model shown in Appendix A we can deduce which other elements must be added to the slice set to make up the residual slice set. Rule 11 states that each *write* action is uniquely paired (function *storeof*) with a preceding *store* action, hence *store* actions must be added to the residual slice set. Rule 4 states that *"A thread is not permitted to write data from its working memory back to the main memory for no reason"*. This rules forces one to add *assign* actions to the residual slice set and finally, rules 13 and 14 force one to add *lock* and *unlock* actions as well. Definition 2 formalizes the construction of a residual slice set as described before.

Definition 2 (residual slice set construction S_{Cr}) Given an event space η and a slice set S_C, one constructs a residual slice set S_{Cr} from S_C adding actions to S_C as follows:

(i) For each write action $w\!:\!\mathbf{write}(\theta, l, v)$ in S_C, the only action s in η such that $s : \mathbf{store}(\theta, l, v)$ and $s = \mathit{storeof}(w)$ must be added to S_{Cr}. The uniqueness of this element is given by the definition of event space and because *storeof* is a total function.

(ii) For each pair of store actions $s\!:\!\mathbf{store}(\theta, l)$, $s'\!:\!\mathbf{store}(\theta, l)$ in η such that $s \neq s'$ and $s \leq s'$, all actions $a\!:\!\mathbf{assign}(\theta, l)$ in η such that $s \leq a \leq s'$ must be added to S_{Cr}.

(iii) Each *lock* and *unlock* actions in η must be added to S_{Cr}.

(iv) For each write action $w\!:\!\mathbf{write}(l)$ in S_C, all read actions $r\!:\!\mathbf{read}(l)$ in η such that $w \leq r$ or $r \leq w$ must be added to S_{Cr}.

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

Example 3 Given the slice set $S_C = \{\mathbf{write}(\theta_1, \texttt{x.i}, 7),\ \mathbf{write}(\theta_1, \texttt{x.j}, 5)\}$, the residual slice set S_{Cr} for the event space of Figure 1 is:

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

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

4 Slicing Event Spaces in a Context with Aliasing

Unfortunately, if one wishes to deduce the value that `x.i` has when executing the program in Example 1, we can not do it based just on the information

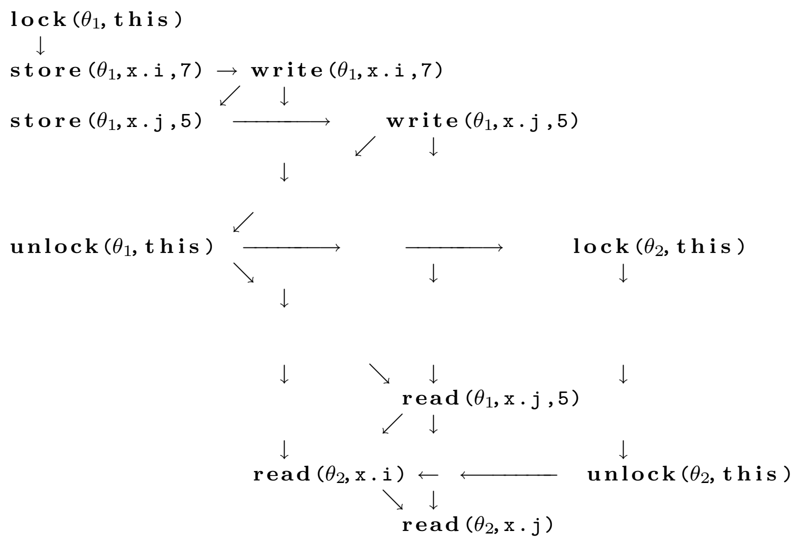

Fig. 2. Sliced event space

provided by the program slice in Figure 2 since this program slice does not contain any information about the field `y.i`, *aliased* with `x.i` and modified afterwards.

In the event space presented in Example 1, when the thread θ_2 executes the synchronized part of method `q()` and after assigning `x` to `y` ($\mathbf{write}(\theta_2, \mathtt{y}, ref_{\mathtt{x}})$ in Column 6), these two variables become *aliases.* Hence, when thread θ_1 modifies `y.i` by assigning `x.j` to it ($\mathbf{write}(\theta_1, \mathtt{y.i}, 5)$ in Column 4), it also modifies `x.i` since the aliasing between `x` and `y` happens before the assignment to `y.i` ($\mathbf{write}(\theta_2, \mathtt{y}, ref_{\mathtt{x}}) \leq \mathbf{write}(\theta_1, \mathtt{y.i}, 5)$). This section faces the problem of aliasing when calculating a program slice from event spaces, outlining an algorithm that finds aliased variables based on the *static* information provided by the event space.

Given a slicing criterion $C = \{\mathbf{read}(x_1), \cdots, \mathbf{read}(x_n)\}$ of *read* actions on variables x_i, we want to construct a slice set S_C of *write* actions on variables x_i and to find their aliases, so we should rephrase the definition of *flow dependence* given by Definition 1 in order to consider aliased variables.

Definition 3 (aliasing flow dependence $p \xrightarrow{fd_a} q$) An event q is *aliasing flow dependent* on an event p, if p *writes* to a variable `y.i`, q *reads* from a variable `x.i` with `y` *alias* of `x` and there is no action w: $\mathbf{write}(\mathtt{z.i})$ with `z` alias of `x` such that $p \leq w \leq q$.

Now, given an object `x.i`, we want to find all the aliases of `x` that modify the field `i`. One should only consider aliases `y` of `x` that affect `x.i`, *i.e*, the aliasing between `y` and `x` must occur before the assignment to `y.i` (if any) and there should not exist an interleaving aliasing between `y` and a variable other than x along the path that leads from the aliasing between y and x and the assignment to y.i. We formalize this idea with the aid of an function *SetW* (see below) which, when applied to an action $r\!:\!\mathbf{read}(\mathtt{x.i})$, returns the set of actions $w\!:\!\mathbf{write}(\mathtt{y.i})$ in the carrier of the event space η such that the predicate $\mathit{SetW}(\mathbf{write}(\mathtt{y.i}), \mathbf{read}(\mathtt{x.i}))$? is fulfilled. An event is considered to be in the carrier of an event space if it is related to itself.

$$\mathit{SetW}(r\!:\!\mathbf{read}(\mathtt{x.i}), \eta) = \{\ w\!:\!\mathbf{write}(\mathtt{y.i}).w \in \mathit{carrier}(\eta)\ \&\ \mathit{SetW?}(w,r)\ \}$$

The predicate *SetW*? formalizes the notion of aliasing flow dependence presented above. In the first case, when the variables y and x are equal (*w.r.t* string equality) and w occurs before r ($w{\le}r$), $\mathbf{read}(\mathtt{x.i})$ is alias flow dependent of $\mathbf{write}(\mathtt{y.i})$ 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 different strings we have two cases. First, if w and r are related and the former occurs before the latter ($w{\le}r$) one not only needs 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. Moreover, when y and x are different, it is possible that $w : \mathbf{write}(\mathtt{y.i})$ and $r : \mathbf{read}(\mathtt{x.i})$ are not related, since the Java language specification does not ensure a total order for write and read actions on different variables. In this case one can opt for a defensive approach, considering that $\mathbf{write}(\mathtt{y.i})$ modifies x.i provided that y is an alias of x when w happened. In our example in Figure 1, we consider $w\!:\!\mathbf{write}(\mathtt{y.i}, 5)$ in Row 28 and Column 4 alias flow dependent of $r\!:\!\mathbf{read}(\mathtt{x.i})$ in Row 46 and Column 2 even though $w{\not\le}r$.

$$\mathit{SetW?}(w\!:\!\mathbf{write}(\mathtt{y.i}), r\!:\!\mathbf{read}(\mathtt{x.i})) =$$
$$\begin{cases} \mathit{true}, \text{ if } \begin{cases} 1.\ \mathtt{y}{=}\mathtt{x} \\ 2.\ w{\le}r \\ 3.\ \neg\exists w_1\!:\!\mathbf{write}(\mathtt{z.i}).\mathit{Alias?}(\mathtt{x},\mathtt{z},w_1)\ \&\ w{<}w_1{\le}r\ \&\ \mathit{SetW?}(w_1,r) \end{cases} \\ \mathit{true}, \text{ if } \begin{cases} 1.\ \mathtt{y}{\neq}\mathtt{x} \\ 2.\ w{\le}r \\ 3.\ \mathit{Alias?}(\mathtt{x},\mathtt{y},w) \\ 4.\ \neg\exists w_1\!:\!\mathbf{write}(\mathtt{z.i}).\mathit{Alias?}(\mathtt{x},\mathtt{z},w_1)\ \&\ w{<}w_1{\le}r\ \&\ \mathit{SetW?}(w_1,r) \end{cases} \\ \mathit{true}, \text{ if } \begin{cases} 1.\ \mathtt{y}{\neq}\mathtt{x} \\ 2.\ w{\not\le}r\ \&\ r{\not\le}w \\ 3.\ \mathit{Alias?}(\mathtt{x},\mathtt{y},w) \end{cases} \\ \mathit{false} \quad \text{otherwise} \end{cases}$$

The predicate $SetW?$ uses the predicate $Alias?(x, y, w)$ to establish whether x and y are aliased at the moment the action w occurs in the event space. This last predicate is defined as the disjunction of the predicate $AliasAux?$ with the parameters swapped.

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

Below, we present the definition of $AliasAux?(\mathtt{y}, \mathtt{x}, w)$. Notice that, according to the description presented up to now, x and y are field identifiers, that is, they are represented as 2-tuples consisting of an identifier and the type information, hence the aliasing problem is reduced to an *"assignment"* matter. The predicate $AliasAux?(\mathtt{y}, \mathtt{x}, w)$ checks that at the moment w occurs, y references the same variable as x as a consequence of an assignment to y from an alias of x. In the first case, the writing to y is from exactly the variable x, $w_2 : \mathbf{write}(\mathtt{y}, ref_{\mathtt{x}})$; in this case one must check that afterwards the reference of y is not modified by some write action w_1 to some reference $ref_{\mathtt{t}}$ not alias of x. In the second case, the writing to y is from a variable z other than x, hence additionally one must check that z was an alias of x before w_2 occurred.

$$AliasAux?(\mathtt{y}, \mathtt{x}, w : \mathbf{write}) = \begin{cases} true, \text{ if } \begin{cases} 1.\ \exists w_2 : \mathbf{write}(\mathtt{y}, ref_{\mathtt{x}}).w_2 \leq w \\ 2.\ \neg\exists w_1 : \mathbf{write}(\mathtt{y}, ref_{\mathtt{t}}).w_2 \leq w_1 < w \ \&\ \neg Alias?(\mathtt{x}, \mathtt{t}, w_1) \end{cases} \\ true, \text{ if } \begin{cases} 1.\ \exists w_2 : \mathbf{write}(\mathtt{y}, ref_{\mathtt{z}}).w_2 \leq w \ \&\ \mathtt{z} \neq \mathtt{x} \\ 2.\ \neg\exists w_1 : \mathbf{write}(\mathtt{y}, ref_{\mathtt{t}}).w_2 \leq w_1 < w \ \&\ \neg Alias?(\mathtt{x}, \mathtt{t}, w_1) \\ 3.\ Alias?(\mathtt{y}, \mathtt{z}, w_2) \end{cases} \\ false \quad \text{otherwise} \end{cases}$$

The example below incorporates the new definition of aliasing flow dependence $p \xrightarrow{fd_a} q$ in the calculation of the slice set S_C from the slicing criterion C and uses the definition of $SetW$ for obtaining aliases of elements in C which should also belong to S_C.

Example 4 Given the same *slicing criterion* $C = \{\ \mathbf{read}(\mathtt{x.i}),\ \mathbf{read}(\mathtt{x.j})\ \}$ presented in Example 3 and given the event space in Figure 1, we use the function $SetW$ to find alias flow dependences for elements of C:

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

and therefore the *slice set* S_C becomes:

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

Now, following the definition of *residual slice set construction* given by Definition 2 we have:

$$\begin{array}{llll}
\multicolumn{4}{l}{S_{Cr} = \{} \\
\textbf{write}(\theta_1, \texttt{x.i}, 7), & \textbf{write}(\theta_1, \texttt{x.j}, 5), & \textbf{write}(\theta_1, \texttt{y.i}, 5), & \\
\textbf{store}(\theta_1, \texttt{x.i}, 7), & \textbf{store}(\theta_1, \texttt{x.i}, 5), & \textbf{store}(\theta_1, \texttt{y.i}, 5), & \\
\textbf{lock}(\theta_1, \texttt{this}), & \textbf{unlock}(\theta_1, \texttt{this}), & \textbf{lock}(\theta_2, \texttt{this}), & \textbf{unlock}(\theta_2, \texttt{this}), \\
\textbf{read}(\theta_1, \texttt{x.j}, 5), & \textbf{read}(\theta_2, \texttt{x.i}), & \textbf{read}(\theta_2, \texttt{y.i}, 7), & \textbf{read}(\theta_2, \texttt{x.j})\ \}
\end{array}$$

5 Event Spaces as Labeled Transition Systems

Event spaces can easily be represented as labeled transition systems with event spaces as nodes and actions as labels. Take for example the event space generated for the assignment `x = y + 1`; as presented by the left side in Figure 3, and suppose that initially `y` is `5`. This event space can be seen as the labeled transition system in the right side. In that figure, we have placed the nodes of the transition system on the *odd* lines and we have marked the transitions with arrows. Initially, on Line 1 we have the event space formed only of the event **read**$(0, y, 5)$, afterwards the action **load**$(0, y, 5)$ is added to the event space, having now two actions (Line 3). The added action is seen as a label between the old event space and the event space produced by adding the new action. The same happens as the other actions are added.

```
read(0,y,5)        { read(0,y,5) }
  ↓                  | A=load(0,y,5)
load(0,y,5)        { read(0,y,5) → load(0,y,5) }
  ↓                  | A=use(0,y,5)
use(0,y,5)         { read(0,y,5) → load(0,y,5) → use(0,y,5) }
  ↓                  | A=assign(0,x,6)
assign(0,x,6)      { read(0,y,5) →···→use(0,y,5) → assign(0,x,6) }
  ↓                  | A→store(0,x,6)
store(0,x,6)       { read(0,y,5) →···→ assign(0,x,6) → store(0,x,6) }
  ↓                  | A=write(0,x,6)
write(0,x,6)       { read(0,y,5) →···→ store(0,x,6) → write(0,x,6) }
```

Fig. 3. Event spaces as labeled transition systems

One constructs a labeled transition system from an event space as expressed the following definition, which is taken from [10].

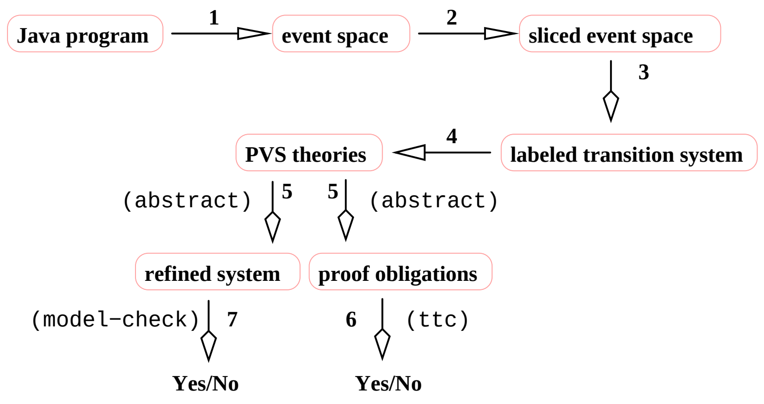

Fig. 4. general framework

Definition 4 **($(\mathcal{E}, \mathcal{A}, (\xrightarrow{A}_{A\in\mathcal{A}}))$)** Given the event spaces $\eta' = (X', \leq')$ and $\eta = (X, \leq)$. The event space η' is a one-step extension of η by action A, $\eta \xrightarrow{A} \eta'$, if $\eta' \in \eta \oplus A$ with $X' \backslash X = \{x\}$. We define the transition system $(\mathcal{E}, \mathcal{A}, (\xrightarrow{A}_{A\in\mathcal{A}}))$ for all event spaces in the set of all event spaces $\mathcal{E}$ such that $\eta \xrightarrow{A} \eta'$ if and only if η' is a one-step extension of η.

6 Conclusion and Future Work

We are planning to express labeled transition systems as theories in the logic of PVS and to use model checking and theorem proving techniques to check specifications on event spaces. Figure 4 presents our methodology. Firstly, we model Java programs as event spaces (Arrow 1) as sketched in Section 2. In short, an event space is a partial order of both memory and thread actions generated by the Java program. The construction of event spaces is based on the operational semantics defined in [10] and, in turn, this operational semantics is based in the detailed yet not formal specification given in Chapter 17 of the Java language specification [7]. We have implemented these operational semantics rules as an ML program taking as input a Java program and returning the corresponding event space structure. The Java program must be provided in ML syntax. Event spaces can be sliced as depicted by Arrow 2 and presented in Section 4. Moreover, event spaces can be seen as labeled transition systems (Arrow 3 and Section 5), which in turn can be represented as theories in the logic of a theorem prover (Arrow 4).

We plan to use PVS as a framework in which model checking and theorem proving techniques are integrated for the purpose of checking specifications on event spaces. In the following we sketch this integration as formally presented by *Owre*, *Rajan*, *Rushby*, *Shankar*, *Srivas* and *Saïdi* in [12,14,16]. In short, in PVS it is possible to define boolean abstractions for finite systems while preserving properties in the *μ–calculus*. This abstraction is implemented as the `abstract` decision procedure (Arrow 5) and its algorithm refines the abstraction until the property is verified or a counterexample is found. This abstraction process generates proof obligations that can be discharged in PVS (Arrow 6). In PVS the *μ–calculus* has been given a representation, hence model checking implemented as a proof rule can be used as the decision procedure `model–check` (Arrow 7). This decision procedure can be used provided that the system ranges over finite domains, hence the big challenge for us is to provide a finite representation for the states of the labeled transition system.

Acknowledgments

The author wishes to thank the anonymous referees for useful comments and M. Huisman for valuable discussions.

References

[1] The Bandera project. `http://www.cis.ksu.edu/santos/bandera/`.

[2] N. Cataño and M. Huisman. Formal specification of GEMPLUS' electronic purse case study. In Lars-Henrik Eriksson and Peter A. Lindsay, editors, *FME: Formal Methods Europe*, volume 2391 of *Lecture Notes in Computer Science*, pages 272–289, Copenhagen, Denmark, July 22-24 2002. Springer.

[3] N. Cataño and M. Huisman. Chase: A static checker for JML's assignable clause. In Lenore D. Zuck, Paul C. Attie, Agostino Cortesi, and Supratik Mukhopadhyay, editors, *VMCAI: Verification, Model Checking and Abstract Interpretation*, volume 2575 of *Lecture Notes in Computer Science*, pages 26–40, New York, NY, USA, January 9-11 2003. Springer.

[4] P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. From sequential to multi-threaded Java: An event-based operational semantics. In *Proceedings in Algebraic Methodology and Software Technology*, volume 1349 of *Lecture Notes in Computer Science*, pages 75–90. Springer, 1997.

[5] P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An event-based structural operational semantics of multi-threaded Java. In In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 157–200. Springer, Berlin, 1999.

[6] Extended static checking for ESC/Java. Compaq Research Center, `http://www.research.compaq.com/SRC/esc/Esc.html`.

[7] J. Gosling, B. Joy, and G. Steel. *The Java Language Specification.* Addison-Wesley, 2001.

[8] J. Hatcliff, J. Corbett, M. Dwyer, S. Sokolowski, and H. Zheng. A formal study of slicing for multi-threaded programs with JVM concurrency primitives. In A. Cortesi and G. Filé, editors, *SAS, Static Analysis*, volume 1694 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 1999.

[9] The Java Modeling Language JML. `http://www.cs.iastate.edu/~leavens/JML/`.

[10] Alexander Knapp. *A Formal Approach to Object-Oriented Software Engineering, Ph.D. Thesis.* Ludwig-Maximilians-Universität München, 2000.

[11] The LOOP project. `http://www.cs.kun.nl/~bart/LOOP/index.html`.

[12] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *Computer-Aided Verification (CAV '96)*, number 1102 in Lecture Notes in Computer Science, pages 411–414. Springer-Verlag, 1996.

[13] The PVS Specification and Verification System. `http://pvs.csl.sri.com/`.

[14] S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In Pierre Wolper, editor, *Computer-Aided Verification, CAV '95*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97, Liège, Belgium, June 1995. Springer-Verlag.

[15] 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, 2000.

[16] H. Saïdi and N. Shankar. Abstract and model check while you prove. In Nicolas Halbwachs and Doron Peled, editors, *Computer-Aided Verification, CAV '99*, volume 1633 of *Lecture Notes in Computer Science*, pages 443–454, Trento, Italy, July 1999. Springer-Verlag.

[17] D. Syme. Proving Java type soundness. In In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 83–118. Springer, Berlin, 1999.

[18] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[19] D. von Oheimb and T. Nipkow. Machine-checking the Java specification: Proving type-safety. In In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 119–156. Springer, Berlin, 1999.

A Java Memory Model Formalization

$$t\!:\!(T,\theta),\ t'\!:\!(T,\theta) \Rightarrow t\!:\!(T,\theta)\leq t'\!:\!(T,\theta)\ \vee\ t'\!:\!(T,\theta)\leq t\!:\!(T,\theta) \tag{1}$$

$$m\!:\!(M,l),\ m'\!:\!(M,l) \Rightarrow m\!:\!(M,l)\leq m'\!:\!(M,l)\ \vee\ m'\!:\!(M,l)\leq m\!:\!(M,l) \tag{2a}$$

$$k\!:\!(K,o),\ k'\!:\!(K,o) \Rightarrow k\!:\!(K,o)\leq k'\!:\!(K,o)\ \vee\ k'\!:\!(K,o)\leq k\!:\!(K,o) \tag{2b}$$

$$\mathbf{assign}(\theta,l)\leq\mathbf{load}(\theta,l) \Rightarrow \mathbf{assign}(\theta,l)\leq\mathbf{store}(\theta,l)\leq\mathbf{load}(\theta,l) \tag{3}$$

$$\begin{gathered} s\!:\!\mathbf{store}(\theta,l)\leq s'\!:\!\mathbf{store}(\theta,l) \Rightarrow \\ s\!:\!\mathbf{store}(\theta,l)\leq\mathbf{assign}(\theta,l)\leq s'\!:\!\mathbf{store}(\theta,\mathtt{1}) \end{gathered} \tag{4}$$

$$\mathbf{use}(\theta,l) \Rightarrow \mathbf{assign}(\theta,l)\leq\mathbf{use}(\theta,l)\ \vee\ \mathbf{load}(\theta,l)\leq\mathbf{use}(\theta,l) \tag{5}$$

$$\mathbf{store}(\theta,l) \Rightarrow \mathbf{assign}(\theta,l)\leq\mathbf{store}(\theta,l) \tag{6}$$

$$\begin{gathered} \mathbf{assign}(\theta,l,v)\leq\mathbf{use}(\theta,l,v') \Rightarrow \\ v{=}v'\ \vee\ \mathbf{assign}(\theta,l,v)<\mathbf{assign}(\theta,l)\leq\mathbf{use}(\theta,l,v') \vee \\ \mathbf{assign}(\theta,l,v)\leq\mathbf{load}(\theta,l)\leq\mathbf{use}(\theta,l,v') \end{gathered} \tag{7}$$

$$\begin{gathered} \mathbf{load}(\theta,l,v)\leq\mathbf{use}(\theta,l,v') \Rightarrow \\ v{=}v'\ \vee\ \mathbf{load}(\theta,l,v)\leq\mathbf{assign}(\theta,l)\leq\mathbf{use}(\theta,l,v') \vee \\ \mathbf{load}(\theta,l,v)<\mathbf{load}(\theta,l)\leq\mathbf{use}(\theta,l,v') \end{gathered} \tag{8}$$

$$\begin{gathered} \mathbf{assign}(\theta,l,v)\leq\mathbf{store}(\theta,l,v') \Rightarrow \\ v{=}v'\ \vee\ \mathbf{assign}(\theta,l,v)<\mathbf{assign}(\theta,l,v')\leq\mathbf{store}(\theta,l,v') \end{gathered} \tag{9}$$

$$l_1\!:\!\mathbf{load}(\theta,l,v) \Rightarrow r\!:\!\mathbf{read}(\theta,l,v){=}\mathit{readof}(l_1)\leq l_1\!:\!\mathbf{load}(\theta,l,v) \tag{10}$$

$$w\!:\!\mathbf{write}(\theta,l,v) \Rightarrow s\!:\!\mathbf{store}(\theta,l,v){=}\mathit{storeof}(w)\leq w\!:\!\mathbf{write}(\theta,l,v) \tag{11}$$

$$s\!:\!\mathbf{store}(\theta,l,v)\leq l\!:\!\mathbf{load}(\theta,l) \Rightarrow \mathit{writeof}(s)\leq\mathit{readof}(l) \tag{12}$$

$$n\!:\!\mathbf{unlock}(\theta,o) \Rightarrow \mathit{lockof}(n)\leq n\!:\!\mathbf{unlock}(\theta,o) \tag{13}$$

$$o\!:\!\mathbf{lock}(\theta,o)\leq\mathbf{lock}(\theta',o)\ \wedge\ \theta{\neq}\theta' \Rightarrow \mathit{unlockof}(o)\leq\mathbf{lock}(\theta',o) \tag{14}$$

$$\begin{gathered} \mathbf{assign}(\theta,l)\leq\mathbf{unlock}(\theta) \Rightarrow \\ \mathbf{assign}(\theta,l)\leq\mathit{storeof}(w)\leq w\!:\!\mathbf{write}(\theta,l)\leq\mathbf{unlock}(\theta) \end{gathered} \tag{15}$$

$$\begin{gathered} \mathbf{lock}(\theta)\leq\mathbf{use}(\theta,l) \Rightarrow \\ \mathbf{lock}(\theta)\leq\mathbf{assign}(\theta,l)\leq\mathbf{use}(\theta,l) \vee \\ \mathbf{lock}(\theta)\leq r\!:\!\mathbf{read}(\theta,l)\leq\mathit{loadof}(r)\leq\mathbf{use}(\theta,l) \end{gathered} \tag{16}$$

$$\mathbf{lock}(\theta)\leq\mathbf{store}(\theta,l) \Rightarrow \mathbf{lock}(\theta)\leq\mathbf{assign}(\theta,l)\leq\mathbf{store}(\theta,l) \tag{17}$$