

A Certified Data Race Analysis for a Java-like Language^{*}

Frédéric Dabrowski and David Pichardie

INRIA, Centre Rennes - Bretagne Atlantique, Rennes, France

Abstract. A fundamental issue in multithreaded programming is detecting *data races*. A program is said to be well synchronised if it does not contain data races w.r.t. an interleaving semantics. Formally ensuring this property is central, because the JAVA Memory Model then guarantees that one can safely reason on the interleaved semantics of the program. In this work we formalise in the COQ proof assistant a JAVA bytecode data race analyser based on the conditional must-not alias analysis of Naik and Aiken. The formalisation includes a context-sensitive points-to analysis and an instrumented semantics that counts method calls and loop iterations. Our JAVA-like language handles objects, virtual method calls, thread spawning and lock and unlock operations for threads synchronisation.

1 Introduction

A fundamental issue in multithreaded programming is *data races*, i.e., the situation where two threads access a memory location, and at least one of them changes its value, without proper synchronisation. Such situations can lead to unexpected behaviours, sometimes with damaging consequences [14, 20]. The semantics of programs with multiple threads of control is described by architecture-dependent *memory models* [10, 1] which define admissible executions, taking into account optimisations such as caching and code reordering. Unfortunately, these models are generally not *sequentially consistent*, i.e., it might not be possible to describe every execution of a program as the *serialization*, or *interleaving*, of the actions performed by its threads. Although common memory models impose restrictions on admissible executions, these are still beyond intuition: writes can be seen out of order and reads can be speculative and return values from the future.

Reasoning directly on memory models is possible but hard, counter-intuitive and probably infeasible to the average programmer. As a matter of fact, the *interleaving semantics* is generally assumed in most formal developments in compilation, static analysis and so on. Hopefully, under certain conditions, the interleaving semantics can be turned into a correct approximation of admissible behaviors. Here, we focus on programs expressed in JAVA, which comes with its own, relieved from architecture specific details, memory model. Although the JAVA memory model [21, 15] does not guarantee sequential consistency for all programs, race free programs are guaranteed to be sequentially consistent. Moreover, it enjoys a major property, so called, *the data race free guarantee*. This property states that a program whose all sequentially consistent

^{*} Work partially supported by EU project MOBIUS, and by the ANR-SETI-06-010 grant.

executions are race free, only admit sequentially consistent executions. In other words, proving that a program is race free can be done on a simple interleaving semantics; and doing so guarantees the correctness of the interleaving semantics for that program. It is worth noticing that data race freedom is important, not only because it guarantees semantic correctness, but also because it is at the basis of a higher level property called atomicity. The possibility to reason sequentially about atomic sections is a key feature in analysing multithreaded programs. Designing tools, either static or dynamic, aiming at proving datarace freeness is thus a fundamental matter.

This paper takes root in the european MOBIUS project¹ where several program verification techniques have been machine checked with respect to a formal semantics of the sequential JAVA bytecode language. The project has also investigated several verification techniques for multithreaded JAVA but we need a formal guarantee that reasoning on interleaving semantics is safe. While a JAVA memory model's formalisation has been done in COQ [9] and a machine-checked proof of the data race free guarantee has been given in [2] we try to complete the picture formally proving data race freeness. We study how such a machine-checked formalisation can be done for the race detection analysis recently proposed by Naik and Aiken [18, 17, 16].

The general architecture of our development is sketched in Figure 1. We formalise four static analyses : a context-sensitive points-to analysis, a must-lock analysis, a conditional must-not alias analysis based on disjoint reachability and a must-not thread escape analysis. In order to ensure the data-race freeness of the program, these analyses are used to refine, in several stages an initial over-approximation of the set of potential races of a program, with the objective to obtain an empty set at the very last stage. Each analysis is mechanically proved correct with respect to an operational semantics. However, we consider three variants of semantics. While the first one is a standard small-step semantics, the second one attaches context information to each reference and frame. This instrumentation makes the soundness proof of the points-to analysis easier. The last semantics handles more instrumentation in order to count method calls and loop iterations. Each instrumentation is proved correct with respect to the semantics just above it. The notion of safe instrumentation is formalised through a standard simulation diagram.

The main contributions of our work are as follows.

- Naik and Aiken have proposed one of the most powerful data race analysis of the area. Their analyser relies on several stages that remove pairs of potential races. Most of these layers have been described informally. The most technical one has been partially proved correct with pencil and paper for a sequential While language [17]. We formalise their work in COQ for a realistic bytecode language with unstructured control flow, operand stack, objects, virtual method calls and lock and unlock operations for threads synchronization.
- Our formalisation is an open framework with three layers of semantics. We formalise and prove correct four static analyses on top of these semantics. We expect our framework to be sufficiently flexible to allow easy integration of new certified blocks for potential race pruning.

¹ <http://mobius.inria.fr>

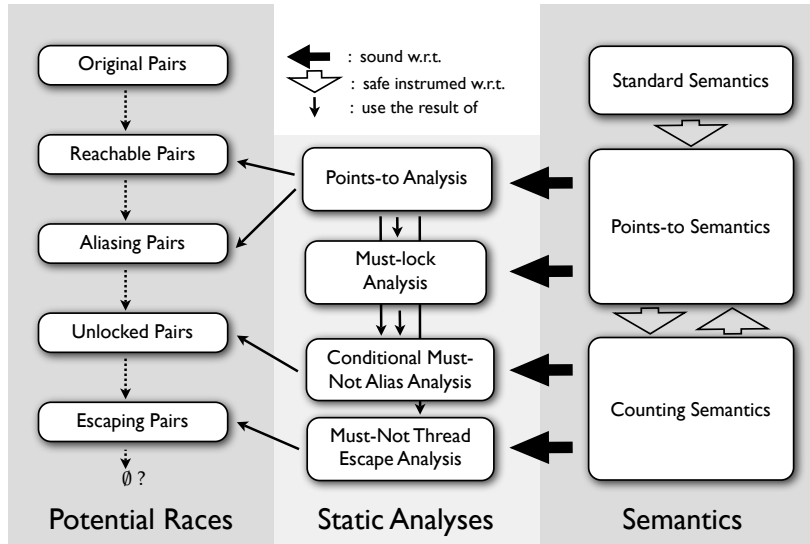


Fig. 1. Architecture of the development

2 A challenging example program

Figure 2 presents an example of a source program for which it is challenging to formally prove race freeness. This example is adapted from the running example given by Naik and Aiken [17] in a While language syntax. The program starts in method `main` by creating in a first loop, a simple linked list `l` and then launches a bunch of threads of class `T` that all share the list `l` in their field `data`. Each thread, launched in this way, chooses non deterministically a cell `m` of the list and then updates `m.val.f`, using a lock on `m`.

Figure 3 presents the potential races computed for this example. A data race is described by a triplet (i, f, j) where i and j denote the program points in conflicts and f denotes the accessed field. The first over-approximation, the set of *Original Pairs* is simply obtained by typing: thanks to JAVA’s strong typing, a pair of accesses may be involved in a race only if both access the same field and at least one access is a write. For each other approximation, a marked indicates a potential race. The program is in fact data race free but the size of the set of *Original Pairs* (13 pairs here) illustrates the difficulty of statically demonstrating it.

Following [18], a first approximation of races computes the field accesses that are reachable from the entry point of the program and removes also pairs where both accesses are taken by the main thread (*Reachable pairs*). Some triplets may also be removed with an alias analysis that shows that two potential conflicting accesses are not in alias (*Aliasing pairs*). Both sets rely on the points-to analysis presented in Section 4. Among the remaining potential races, several triplets can be disabled by a *must-not thread escape* analysis that predicts a memory access only concerns a reference which

```

class Main() {
    void main() {
        List l = null;
        while (*) {
            List temp = new List;
1:     temp.val = new T;
2:     temp.val.f = new A;
3:     temp.next = l;
        l = temp; };
        while (*) {
            T t = new T;
4:     t.data = l;
            t.start();
5:     t.f = null; }};
}

class A{};
class List{ T val; List next; };

class T extends java.lang.Thread {
    A f;
    List data;
    void run(){
        while(*) {
6:     List m = this.data;
7:     while (*) { m = m.next; }
            synchronized(m)
8:         { m.val.f = new A; }};
        }
    }
};

```

Fig. 2. A challenging example program

is local to a thread at the current point (*Escaping pairs*). The last potential race (8, f, 8) requires the most attention since several threads of class T are updating fields f in parallel. These writes are safe because they are guarded by a synchronization on an object which is the only ancestor of the write target in the heap. Such reasoning relies on the fact that if locks guarding two accesses are different then so are the targeted memory locations. The main difficulty comes when several objects allocated at the same program point, e.g. within a loop, may point to the same object. This last triplet is removed by the *conditional must not alias* presented in Section 5.

Original	Reachable	Aliasing	Unlocked	Escaping
(1, val, 1), (1, val, 2), (2, f, 2), (3, next, 3), (4, data, 4)		✓	✓	
(5, f, 5)		✓	✓	✓
(2, f, 5)			✓	
(5, f, 8)	✓		✓	✓
(4, data, 6), (3, next, 7), (1, val, 8), (2, f, 8)	✓	✓	✓	
(8, f, 8)	✓	✓		✓

Fig. 3. Potential race pairs in the example program

3 Standard Semantics

The previous example can be compiled into a bytecode language whose syntax is given below. The instruction set allows to manipulate objects, call virtual methods, start threads and lock (or unlock) objects for threads synchronization.

Compared to real JAVA, we discard all numerical manipulations because they are not relevant to our purpose. Static fields, static methods and arrays are not managed here but they are nevertheless source of several potential data races in JAVA programs. Naik’s approach [18] for these layers is similar to the technique developed for objects. We estimate that adding these language features would not bring new difficulties that we have not covered yet in the current work. At last, as Naik and Aiken did before us, we only cover synchronization by locks without join, wait and interruption mechanisms. Our approach is sound in presence of such statements, but doesn’t take into account the potential races they could prevent. The last missing feature is the JAVA’s exception mechanism. Exceptions complicate the control flow of a JAVA program. We expect that handling this mechanism would increase the amount of formal proof but will not require new proof techniques. This is left for further work.

Program syntax A program is a set of classes, coming with a *Lookup* function matching signatures and program points (allocation sites denoting class names) to methods.

$$\begin{array}{l} \mathbb{C}_{id} \supseteq \{c_{id}, \dots\} \quad \mathbb{F} \supseteq \{f, g, h, \dots\} \quad \mathbb{M}_{id} \supseteq \{m_{id}, \dots\} \\ \mathbb{V} \supseteq \{x, y, z, \dots\} \quad \mathbb{M}_{sig} = \mathbb{M}_{id} \times \mathbb{C}_{id}^n \times (\mathbb{C}_{id} \cup \{\text{void}\}) \end{array}$$

$$\begin{array}{l} \mathbb{M} \ni \{\text{sig} \in \mathbb{M}_{sig}; \text{body} \in \mathbb{N} \rightarrow \text{inst}\} \\ \mathbb{C} \ni \{\text{name} \in \mathbb{C}_{id}; \text{fields} \subseteq \mathbb{F}; \text{methods} \subseteq \mathbb{M}\} \end{array}$$

$$\begin{array}{l} \text{inst} ::= \text{aconstnull} \mid \text{new } c_{id} \mid \text{aload } x \mid \text{astore } x \mid \text{getfield } f \mid \text{putfield } f \\ \mid \text{areturn} \mid \text{return} \mid \text{invokevirtual } m_{id} : (c_{id}^n) \text{ rtype} \quad (n \geq 0) \\ \mid \text{monitorenter} \mid \text{monitorexit} \mid \text{start} \mid \text{ifnd } \ell \mid \text{goto } \ell \end{array}$$

Semantics Domain The dynamic semantics of our language is defined over states as a labelled transition system. States and labels, or events, are defined in Figure 4, where \rightarrow stands for total functions and \dashrightarrow stands for partial functions. We distinguish *location* and *memory location* sets. The set of locations is kept abstract in this presentation. In this section, a memory location is itself a location ($\mathbb{L} = \mathbb{O}$). This redundancy will be useful when defining new instrumented semantics where memory locations will carry more information (Sections 4 and 5). In a state (L, σ, μ) , L maps memory locations (that identify threads) to call stacks, σ denotes the heap that associates memory locations to objects (c_{id}, map) with c_{id} a class name and map a map from fields to values. We note $\text{class}(\sigma, l)$ for $\text{fst}(\sigma(l))$ when $l \in \text{dom}(\sigma)$. A locking state μ associates with every location ℓ a pair (ℓ', n) if ℓ is locked n times by ℓ' and the constant free if ℓ is not held by any thread. An event $(\ell, ?_f^{ppt}, \ell')$ (resp. $(\ell, !_f^{ppt}, \ell')$) denotes a read (resp. a write) of a field f , performed by the thread ℓ over the memory location ℓ' , at a program point ppt . An event τ denotes a silent action.

Transition system Labelled transitions have the form $st \xrightarrow{e} st'$ (when e is τ we simply omit it). They rely on the usual interleaving semantics, as expressed in the rule below.

$$\frac{L \ell = cs \quad L; \ell \vdash (cs, \sigma, \mu) \xrightarrow{e} (L', \sigma', \mu')}{(L, \sigma, \mu) \xrightarrow{e} (L', \sigma', \mu')}$$

$\mathbb{L} \ni \ell$	(location)
$\mathbb{O} = \mathbb{L} \ni \ell$	(memory location)
$\mathbb{O}_\perp \ni v \quad ::= \ell \mid \mathbf{Null}$	(value)
$s \quad ::= v \quad :: s \mid \varepsilon$	(operand stack)
$\mathbb{V} \rightarrow \mathbb{O}_\perp \ni \rho$	(local variables)
$\mathbb{O} \rightarrow \mathbb{C}_{id} \times (\mathbb{F} \rightarrow \mathbb{O}_\perp) \ni \sigma$	(heap)
$PPT = \mathbb{M} \times \mathbb{N} \ni ppt \quad ::= (m, i)$	(program point)
$CS \ni cs \quad ::= (m, i, s, \rho) \quad :: cs \mid \varepsilon$	(call stack)
$\mathbb{O} \rightarrow CS \ni L$	(thread call stacks)
$\mathbb{O} \rightarrow ((\mathbb{O} \times \mathbb{N}^*) \cup \{\mathbf{free}\}) \ni \mu$	(locking state)
$st \quad ::= (L, \sigma, \mu)$	(state)
$e \quad ::= \tau \mid (\ell, ?_f^{ppt}, \ell') \mid (\ell, !_f^{ppt}, \ell')$	(event)

Fig. 4. States and actions

$$\begin{aligned}
\sigma[\ell.f \leftarrow v] \ell f &= v \\
\sigma[\ell.f \leftarrow v] \ell' f &= \sigma \ell' f \text{ if } \ell' \neq \ell \\
\sigma[\ell.f \leftarrow v] \ell f' &= \sigma \ell f' \text{ if } f' \neq f
\end{aligned}
\quad
\begin{aligned}
(\text{acquire } \ell \ell' \mu) \ell' &= \begin{cases} (\ell, 1) & \text{if } \mu(\ell') = \mathbf{free} \\ (\ell, n+1) & \text{if } \mu(\ell') = (\ell, n) \end{cases} \\
(\text{acquire } \ell \ell' \mu) \ell'' &= \mu \ell'' \text{ if } \ell'' \neq \ell'
\end{aligned}$$

(a) Notations

$$\begin{aligned}
\text{getfield } f; \ell; ppt \vdash (i, \ell' :: s, \rho, \sigma) &\xrightarrow{\ell \overset{ppt}{f} \ell'} (i+1, (\sigma \ell' f) :: s, \rho, \sigma) \quad \text{if } \ell' \in \text{dom}(\sigma) \\
\text{putfield } f; \ell; ppt \vdash (i, v :: \ell' :: s, \rho, \sigma) &\xrightarrow{\ell \overset{ppt}{f} \ell'} (i+1, s, \rho, \sigma[\ell'.f \mapsto v]) \text{ if } \ell' \in \text{dom}(\sigma) \\
\text{new } c_{id}; \ell; ppt \vdash (i, s, \rho, \sigma) &\rightarrow (i+1, \ell' :: s, \rho, \sigma[\ell' \mapsto \text{new}(c_{id})]) \text{ where } \ell' \notin \text{dom}(\sigma)
\end{aligned}$$

$$\frac{(m.\text{body}) i; \ell; (m, i) \vdash (i, s, \rho, \sigma) \xrightarrow{e} (i', s', \rho', \sigma') \quad L' = L[\ell \mapsto (m, i', s', \rho') :: cs]}{L; \ell \vdash ((m, i, s, \rho) :: cs, \sigma, \mu) \xrightarrow{e} (L', \sigma', \mu)} \quad (1)$$

$$\frac{(m.\text{body}) i = \text{invokevirtual } m_{id} : (c_{id}^n) \text{ rtype} \quad (m.\text{body}) i = \text{start} \quad \neg(\ell' \in \text{dom}(L)) \quad s = v_n :: \dots :: v_1 :: \ell' :: s' \quad \text{Lookup } (\text{run} : () \text{void}) \text{ class}(\sigma, \ell') = m_1 \quad \rho_1 = [0 \mapsto \ell'] \quad \text{Lookup } (m_{id} : (c_{id}^n) \text{ rtype}) \text{ class}(\sigma, \ell') = m_1 \quad L' = L[\ell \mapsto (m, i+1, s', \rho) :: cs, \ell' \mapsto (m_1, 0, \varepsilon, \rho_1) :: \varepsilon]}{L; \ell \vdash ((m, i, s, \rho) :: cs, \sigma, \mu) \rightarrow (L', \sigma, \mu) \quad L; \ell \vdash ((m, i, \ell' :: s', \rho) :: cs, \sigma, \mu) \rightarrow (L', \sigma, \mu)}$$

$$\frac{(m.\text{body}) i = \text{monitorenter} \quad \mu \ell' \in \{\mathbf{free}, (\ell, n)\} \quad \mu' = \text{acquire } \ell \ell' \mu \quad L' = L[\ell \mapsto (m, i+1, s, \rho) :: cs]}{L; \ell \vdash ((m, i, \ell' :: s, \rho) :: cs, \sigma, \mu) \rightarrow (L', \sigma, \mu')} \quad
\frac{(m.\text{body}) i = \text{monitorexit} \quad \mu = \text{acquire } \ell \ell' \mu' \quad L' = L[\ell \mapsto (m, i+1, s, \rho) :: cs]}{L; \ell \vdash ((m, i, \ell' :: s, \rho) :: cs, \sigma, \mu) \rightarrow (L', \sigma, \mu')}$$

(b) Reduction rules

Fig. 5. Standard Dynamic Semantics

Reductions of the shape $L; \ell \vdash (cs, \sigma, \mu) \xrightarrow{e} (L', \sigma', \mu')$ are defined in Figure 5. Intuitively, such a reduction expresses that in state (L, σ, μ) , reducing the thread defined by the memory location ℓ and the call stack cs , by a non deterministic choice, produces the new state (L', σ', μ') . For the sake of readability, we rely on an auxiliary relation of the shape $instr; \ell; ppt \vdash (i, s, \rho, \sigma) \xrightarrow{e}_1 (i', s', \rho', \sigma')$ for reduction of intra-procedural instructions. In Figure 5, we consider only `putfield`, `getfield` and `new`. Reductions for instructions are standard and produce a τ event. The notation $\sigma[\ell.f \leftarrow v]$ for field update, where $\ell \in \text{dom}(\sigma)$, is defined in Figure 5(a). It does not change the class of an object. The reduction of a `new` instruction pushes a fresh address onto the operand stacks and allocates a new object in the heap. The notation $\sigma[\ell \leftarrow \text{new}(cid)]$, where $\neg(\ell \in \text{dom}(\sigma))$, denotes the heap σ with a new object, at location ℓ , of class cid and with all fields equals to `Null`. The auxiliary relation is embedded into the semantics by rule (1). Method invocation relies on the *lookup* function for method resolution and generates a new frame. Thread spawning is similar to method invocation. However, the new frame is put on top of an empty call stack. We omit the reduction rules for `return` and `areturn`, those rules are standard and produce a τ event. For `monitorenter` and `monitorexit` we use a partial function *acquire* defined in Figure 5(a). Intuitively, *acquire* $\ell \ell' \mu$ results from thread ℓ locking object ℓ' in μ .

We write $RState(P)$ for the set of states that contains the initial state of a program P , that we do not describe here for conciseness concerns, and that is closed by reduction. A data race is a tuple (ppt_1, f, ppt_2) such that $Race(P, ppt_1, f, ppt_2)$ holds.

$$\frac{st \in RState(P) \quad st \xrightarrow{\ell_1!_f^{ppt_1} \ell_0} st_1 \quad st \xrightarrow{\ell_2 \mathcal{R} \ell_0} st_2 \quad \mathcal{R} \in \{\gamma_f^{ppt_2}, !_f^{ppt_2}\} \quad \ell_1 \neq \ell_2}{Race(P, ppt_1, f, ppt_2)}$$

The ultimate goal of our certified analyser is to guarantee *Data Race Freeness*, i.e. for all $ppt_1, ppt_2 \in PPT$ and $f \in \mathbb{F}$, $\neg Race(P, ppt_1, f, ppt_2)$.

4 Points-to Semantics

Naik and Aiken make intensive use of points-to analysis in their work. Points-to analysis computes a finite abstraction of the memory where locations are abstracted by their allocation site. The analysis can be made context sensitive if allocation sites are distinguished wrt. the calling context of the method where the allocation occurs.

Many static analyses use this kind of information to have a conservative approximation of the call graph and the heap of a program. Such analyses implicitly reason on instrumented semantics that directly manipulates informations on allocation sites while a standard semantics only keeps track of the class given to a reference during its allocation. In this section we formalise such an intermediate semantics.

This *points-to semantics* takes the form of a COQ module functor

```
Module PointsToSem (C:CONTEXT). ... End PointsToSem.
```

parameterised by an abstract notion of context which captures a large variety of points-to contexts. Figure 6 presents this notion.

A context is given by two abstract types `pcontext` and `mcontext`² for pointer contexts and method contexts. Function `make_new_context` is used to create a new pointer context (`make_new_context m i cid c`) when an allocation of an object of class `cid` is performed at line `i` of a method `m`, called in a context `c`. We create a new method context (`make_call_context m i c p`) when building the calling context of a method called on an object of context `p`, at line `i` of a method `m`, itself called in a context `c`. At last, (`get_class prog p`) allows to retrieve the class given to an object allocated in a context `p`. The hypothesis `class_make_new_context` ensures consistency between `get_class` and `make_new_context`.

Module Type `CONTEXT`.

```

Parameter pcontext : Set. (* pointer context *)
Parameter mcontext : Set. (* method context *)

Parameter make_new_context :
  method → line → classId → mcontext → pcontext.
Parameter make_call_context :
  method → line → mcontext → pcontext → mcontext.
Parameter get_class : program → pcontext → option classId.

Parameter class_make_new_context : ∀ p m i cid c,
  body m i = Some (New cid) →
  get_class p (make_new_context m i cid c) = Some cid.

Parameter init_mcontext : mcontext.
Parameter init_pcontext : pcontext.

Parameter eq_pcontext : ∀ c1 c2:pcontext, {c1=c2}+{c1<>c2}.
Parameter eq_mcontext : ∀ c1 c2:mcontext, {c1=c2}+{c1<>c2}.

```

End `CONTEXT`.

Fig. 6. The Module Type of Points-to Contexts

The simplest instantiation of this semantics takes class name as pointer contexts and uses a singleton type for method context. A more interesting instantiation is k -objects sensitivity: contexts are sequences of at most k allocation sites $(m, i) \in \mathbb{M} \times \mathbb{N}$. When creating an object at site (m, i) in a context c , we attach to this object a pointer context $(m, i) \oplus_k c$ defined by $(m, i) \cdot c'$ if $|c|=k$, $c = c' \cdot (m', i')$ and $(m, i) \cdot c$ if $|c| < k$, without any change to the current method context. When calling a method on an object we build a new frame with the same method context as the pointer context of the object.

The definition of this semantics is similar to the standard semantics described in the previous section, except that memory location are now couples of the form (ℓ, p)

² `mcontext` is noted *Context* in Section 5

with ℓ a location and p a pointer context. We found convenient for our formalisation to deeply instrument the heap that is now a partial function from memory location of the form (ℓ, p) to objects. This allows us to state a property about the context of a location without mentioning the current heap (in contrast to the class of a location in the previous standard semantics). The second change concerns frames that are now of the form (m, i, c, s, ρ) with c being the method context of the current frame.

In order to reason on this semantics and its different instantiations we give to this module a module type `POINTSTO_SEM` such that for all modules of type `CONTEXT`, `PointsToSem(C) : POINTSTO_SEM`.

Several invariants are proved on this semantics, for example that if any memory locations (ℓ, p_1) and (ℓ, p_2) are in the domain of a heap reachable from a initial state, then $p_1 = p_2$.

```

Module PointsToSemInv (S:POINTSTO_SEM). ...
  Lemma reachable_wf_heap :  $\forall$  p st,
    reachable p st  $\rightarrow$ 
    match st with (L, sigma, mu)  $\Rightarrow$ 
       $\forall$  l p1 p2, sigma (l, p1) <> None  $\rightarrow$  sigma (l, p2) <> None  $\rightarrow$  p1=p2
    end.
  Proof. ... Qed.
End PointsToSemInv.

```

Safe Instrumentation The analyses we formalise on top of this points-to semantics are meant for proving absence of race. To transfer such a semantic statement in terms of the standard semantics, we prove simulation diagrams between the transitions systems of the standard and the points-to semantics. Such diagram then allows us to prove that each standard race corresponds to a points-to race.

```

Module SemEquivProp (S:POINTSTO_SEM).
  Lemma race_equiv :
     $\forall$  p ppt ppt', Standard.race p ppt ppt'  $\rightarrow$  S.race p ppt ppt'.
  Proof. ... Qed.
End SemEquivProp.

```

Points-to Analysis A generic context-sensitive analysis is specified as a set of constraints attached to a program. The analysis is flow-insensitive for heap and flow-sensitive for local variables and operand stacks. Its result is given by four functions

```

PtL: mcontext  $\rightarrow$  method  $\rightarrow$  line  $\rightarrow$  var  $\rightarrow$  (pcontext  $\rightarrow$  Prop) .
PtS: mcontext  $\rightarrow$  method  $\rightarrow$  line  $\rightarrow$  list (pcontext  $\rightarrow$  Prop) .
PtR: mcontext  $\rightarrow$  method  $\rightarrow$  (pcontext  $\rightarrow$  Prop) .
PtF: pcontext  $\rightarrow$  field  $\rightarrow$  (pcontext  $\rightarrow$  Prop) .

```

that attach pointer context properties to local variables (`PtL`), operand stack (`PtS`) and method returns (`PtR`). `PtF` is the flow-insensitive abstraction of the heap.

This analysis is parameterized by a notion of context and proved correct with respect to a suitable points-to semantics. The final theorem says that if (PtL, PtS, PtR, PtF) is a solution of the constraints system then it correctly approximates any reachable states.

The notion of correct approximation expresses, for example, that all memory locations (ℓ, p) in the local variables ρ or operand stack s of a reachable frame (m, i, c, s, ρ) is such that p is in the points-to set attached to PtL or PtS for the corresponding flow position (m, i, c) .

Must-Lock Analysis Fine lock analysis requires to statically understand which locks are definitely held when a given program point is reached. For this purpose we specify and prove correct a flow sensitive must-lock analysis that computes the following informations:

```
Locks: method  $\rightarrow$  line  $\rightarrow$  mcontext  $\rightarrow$  (var  $\rightarrow$  Prop) .
Symbolic: method  $\rightarrow$  line  $\rightarrow$  list expr.
```

At each flow position (m, i, c) , `Locks` computes an under-approximation of the local variables that are currently held by the thread reaching this position. The specification of `Locks` depends on the points-to information `PtL` computed before. This is a useful information for the `monitorexit` instruction because the unlocking of a variable x can only cancel the lock information of the variables that may be in alias with x . `Symbolic` is a flow sensitive abstraction of the operand stack that manipulate symbolic expressions. Such expressions are path expressions of the form $x, x.f$, etc... Lock analysis only requires variable expressions but more complex expressions are useful for the conditional must lock analysis given in Section 5.

Removing False Potential Races The previous points-to analysis supports the first two stages of the race analyser of Naik et al [18]. The first stage prunes the so called *ReachablePairs*. It only keeps in *OriginalsPairs* the accesses that may be reachable from a `start()` call site that is itself reachable from the `main` method, according to the points-to information. Moreover, it discards pairs where each accesses are performed by the main thread because there is only one thread of this kind.

The next stage keeps only the so called *AliasingPairs* using the fact that a conflicting access can only occur on references that may alias. In the example of the Figure 2, the potential race $(5, f, 8)$ is cancelled because the points-to information of `t` and `m.val` are disjoint.

For each stage we formally prove that all these sets over-approximate the set of real races wrt. the points-to semantics.

5 Counting Semantics

The next two stages of our analysis require a deeper instrumentation. We introduce a new semantics with instrumentation for counting method calls and loop iterations. This semantics builds on top of the points-to semantics and uses k -contexts. All developments of this section were formalized in COQ. However, for the sake of conciseness we introduce them in a paper style. In addition to the allocation site (m, i) and the calling context c of an allocation, this semantics captures counting information. More precisely, it records that the allocation occurred after the n^{th} iteration of flow edge $\mathcal{L}(m, i)$ in the

k^{th} call to m in context c . Given a program P , the function $\mathcal{L} \in \mathbb{M} \times \mathbb{N} \rightarrow \text{Flow}$, for $\text{Flow} = \mathbb{N} \times \mathbb{N}$, must satisfy $\text{Safe}_P(\mathcal{L})$ as defined below:

$$\begin{aligned} \text{Safe}_P(\mathcal{L}) \equiv \forall m, i, c_{id}. (m.\text{body}) i = \mathbf{new} \ c_{id} \Rightarrow \\ \forall n > 0, j_1, \dots, j_n. \text{leadsTo}(m, i, i, j_1 \cdot \dots \cdot j_n) \Rightarrow \\ \exists k < n. (j_k, j_{k+1}) = \mathcal{L}(m, i) \end{aligned}$$

where $\text{leadsTo}(m, i, j, j_1 \cdot \dots \cdot j_n)$ states that $j_1 \cdot \dots \cdot j_n$ is a path from i to j in the control flow graph of m . Intuitively, the semantics counts, and records, iteration of all flow edges while \mathcal{L} maps every allocation site to a flow edge, typically a loop entry. Obviously, the function defined by $\mathcal{L}(m, i) = (i, i+1)$ is safe. However, for the purpose of static analysis we need to observe that two allocations occurred within the same loop and, thus, we need a less strict definition (but still strict enough to discriminate between different allocations occurring at the same site). The function \mathcal{L} might be provided by the compiler or computed afterwards with standard techniques. For example, in the bytecode version of our running example, mapping the three allocation sites of the first loop with the control flow edge of the first one is safe. We update the semantic domain as follows:

$$\begin{array}{lll} mVect = \mathbb{M} \times \text{Context} \rightarrow \mathbb{N} & \ni \omega & \text{(method vector)} \\ lVect = \mathbb{M} \times \text{Context} \times \text{Flow} \rightarrow \mathbb{N} & \ni \pi & \text{(iteration vector)} \\ CP & \ni cp ::= \langle m, i, c, \omega, \pi \rangle & \text{(code pointer)} \\ \textcircled{0} = \mathbb{L} \times CP & \ni \bar{\ell} & \text{(memory location)} \\ CS & \ni cs ::= (cp, s, \rho) :: cs \mid \varepsilon & \text{(call stack)} \\ & st ::= (L, \sigma, \mu, \omega_g) & \text{state} \end{array}$$

A frame holding the code pointer $\langle m, i, c, \omega, \pi \rangle$ is the $\omega(m, c)^{\text{th}}$ call to method m in context c (a k -context) since the execution began and, so far, it has performed $\pi(m, c, \phi)$ steps through edge ϕ of its control flow graph. In a state $(L, \sigma, \mu, \omega_g)$, ω_g is a global method vector used as a shared call counter by all threads.

Below, we sketch the extended transition system by giving rules for allocation and method invocation.

$$\begin{array}{c} (m.\text{body}) i = \mathbf{new} \ cid \quad \forall cp. \neg((\ell', cp) \in \text{dom}(\sigma)) \\ L' = L[\bar{\ell} \mapsto (\langle m, i+1, c, \omega, \pi' \rangle, \bar{\ell}' :: s, \rho) :: cs] \\ \pi' = \pi[(m, c, (i, i+1)) \mapsto \pi(m, c, (i, i+1)) + 1] \quad \bar{\ell}' = (\ell', \langle m, i, c, \omega, \pi \rangle) \\ \hline L; \bar{\ell} \vdash ((\langle m, i, c, \omega, \pi \rangle, s, \rho) :: cs, \sigma, \mu, \omega_g) \rightarrow (L', \sigma[\bar{\ell}' \mapsto \text{new}(cid)], \mu, \omega_g) \end{array}$$

$$\begin{array}{c} (m.\text{body}) i = \mathbf{invokevirtual} \ m_{id} : (cid^n) \ rtype \\ s = v_n :: \dots :: v_1 :: \bar{\ell}' :: s' \quad \bar{\ell}' = (a_0, \langle m_0, i_0, c_0, \omega_0, \pi_0 \rangle) \\ \text{Lookup}(m_{id} : (cid^n) \ rtype) \ \mathbf{class}(\sigma, \bar{\ell}') = m_1 \\ c_1 = (m_0, i_0) \oplus_k c_0 \\ \pi' = \pi[(m, c, (i, i+1)) \mapsto \pi(m, c, (i, i+1)) + 1] \\ \omega_1 = \omega[(m_1, c_1) \mapsto \omega_g(m_1, c_1) + 1] \quad \pi_1 = \pi'[(m_1, c_1, \cdot, \cdot) \mapsto 0] \\ \rho_1 = [0 \mapsto \bar{\ell}', 1 \mapsto v_1, \dots, n \mapsto v_n] \quad \omega'_g = \omega_g[(m_1, c_1) \mapsto \omega_g(m_1, c_1) + 1] \\ L' = L[\bar{\ell} \mapsto (\langle m_1, 0, c_1, \omega_1, \pi_1 \rangle, \varepsilon, \rho_1) :: (\langle m, i+1, c, \omega, \pi' \rangle, s', \rho) :: cs] \\ \hline L; \bar{\ell} \vdash ((\langle m, i, c, \omega, \pi \rangle, s, \rho) :: cs, \sigma, \mu, \omega_g) \rightarrow (L', \sigma, \mu, \omega'_g) \end{array}$$

For allocation, we simply annotate the new memory location with the current code pointer and record the current move. For method invocation, the caller records the current move. The new frame receives a copy of the vectors of the caller (after the call) where the current call is recorded and the iteration vector corresponding to this call is reset. Except for thread spawning, omitted rules simply record the current move. Thread spawning is similar to method invocation except that the new frame receives fresh vectors rather than copies of the caller's vectors.

Safe Instrumentation As we did between the standard and the points-to semantics we prove a diagram simulation between the points-to semantics and the counting semantics. It ensures that all *points-to races* correspond to a *counting race*. However, in order to use the soundness theorem of the must-lock analysis we also need to prove a bisimulation diagram. It ensures that all states that are reachable in the counting semantics correspond to a reachable state in the points-to semantics. It allows us to transfer the soundness result of the must-lock analysis in terms of the counting semantics.

Semantics invariants Proposition 1 states that our instrumentation discriminates between memory locations allocated at the same program point. As expected, to discriminate between memory locations allocated at program point (m, i) in context c , it is sufficient to check the values of $\omega(m, c)$ and $\pi(m, c, \mathcal{L}(m, i))$.

Proposition 1. *Given a program P , if $(L, \sigma, \mu, \omega_g)$ is a reachable state of P and if $\text{Safe}_P(\mathcal{L})$ holds then, for all $\bar{\ell}_1, \bar{\ell}_2 \in \text{dom}(\sigma)$, we have*

$$\left(\bar{\ell}_1 = (a_1, \langle m, i_1, c, \omega_1, \pi_1 \rangle) \wedge \bar{\ell}_2 = (a_2, \langle m, i_2, c, \omega_2, \pi_2 \rangle) \wedge \begin{array}{l} \omega_1(m, c), \pi_1(m, c, \mathcal{L}(m, i)) = \omega_2(m, c), \pi_2(m, c, \mathcal{L}(m, i)) \end{array} \right) \Rightarrow \bar{\ell}_1 = \bar{\ell}_2$$

Proving Proposition 1 requires stronger semantics invariants. Intuitively, when reaching an allocation site, no memory location in the heap domain should claim to have been allocated at the current iteration. More formally, we have proved that any reachable state is well-formed. A state $(L, \sigma, \mu, \omega_g)$ is said to be well formed if for any frame $(\langle m, i, c, \omega, \pi \rangle, s, \rho)$ in L and for any memory location $(\ell, \langle m_0, i_0, c_0, \omega_0, \pi_0 \rangle)$ in the heap domain we have

$$\begin{array}{l} \omega(m, c) \leq \omega_g(m, c) \quad \omega(m_0, c_0) \leq \omega_g(m_0, c_0) \\ \text{localCoherency}(\mathcal{L}, (\ell, \langle m_0, i_0, c_0, \omega_0, \pi_0 \rangle), \langle m, i, c, \omega, \pi \rangle) \\ \omega(m, c) \neq \omega'(m, c) \text{ for all distinct frame } (\langle m, i', c, \omega', \pi' \rangle, s', \rho') \text{ in } L \end{array}$$

where $\text{localCoherency}(\mathcal{L}, (\ell, \langle m_0, i_0, c_0, \omega_0, \pi_0 \rangle), \langle m, i, c, \omega, \pi \rangle)$ stands for

$$\left(\begin{array}{l} (m, c) = (m_0, c_0) \Rightarrow \omega(m, c) = \omega_0(m, c) \Rightarrow \\ \pi_0(m, c, \mathcal{L}(m, i_0)) \leq \pi(m, c, \mathcal{L}(m, i_0)) \wedge \\ \left(\begin{array}{l} \pi_0(m, c, \mathcal{L}(m, i_0)) = \pi(m, c, \mathcal{L}(m, i_0)) \Rightarrow \\ \left(\begin{array}{l} i_0 \neq i \quad \wedge \\ \text{leadsTo}(m, i, i_0, j_1 \cdot \dots \cdot j_n) \Rightarrow \exists k < n, (j_k, j_{k+1}) = \mathcal{L}(m, c) \end{array} \right) \end{array} \right) \end{array} \right)$$

Type And Effect System We have formalized a type and effect system which captures the fact that some components of vectors of a memory location are equals to the same components of vectors of : (1) the current frame when the memory location is in local variables or in the stack of the frame or (2) of another memory location pointing to it in the heap. By lack of space, we cannot describe the type and effect system here. Intuitively, we perform a points-to analysis where allocation sites are decorated with masks which tell us which components of vectors of the abstracted memory location match the same components in vectors of a given code pointer (depending on whether we consider (1) or (2)). Formally, an abstract location $\tau \in \mathbb{T}$ in our extended points-to analysis is a pair (A, F) where A is a set of allocation sites and F maps every element of A to a pair Ω, Π of abstract vectors. Abstract vectors are defined by $\Omega \in MVect = \mathbb{M} \times Context \rightarrow \{\mathbf{1}, \top\}$ and $\Pi \in LVect = \mathbb{M} \times Context \times Flow \rightarrow \{\mathbf{1}, \top\}$.

Our analysis computes a pair (\mathcal{A}, Σ) where \mathcal{A} provides flow-sensitive points-to information with respect to local variables and Σ provides flow-insensitive points-to information with respect to the heap. The decoration of a points-to information acts as a mask. For a memory location held by a local variable, it tells us which components of its vectors (those set to $\mathbf{1}$) match those of the current frame. When a memory location points-to another one in the heap, it tells us which components of their respective vectors are equal.

Below we present the last stages we use for potential race pruning. For each stage the result stated by proposition 1 is crucial. Indeed, given an abstract location with allocation site (m, i, c) , they rely on a property stating that whenever the decoration states that $\Omega(m, c) = \Pi(m, c, \mathcal{L}(m, i)) = \mathbf{1}$, the abstraction describes a unique concrete location. This property results from the combination of the abstraction relation defined in our type system and of Proposition 1.

Must Not Escape Analysis We use the flow sensitive element \mathcal{A} of the previous type and effect system to check that, at some program point, an object allocated by a thread is still local to that thread (or has not escaped yet, i.e. it is not reachable from others). More precisely, the type and effect systems is used to guarantee that, at some program point, the last object allocated by a thread at a given allocation site is still local to that thread. In particular, our analysis proves that an access performed at point 4 in our running example, is on the last object of type T allocated by the main thread (which is local, although at each loop iteration, the new object eventually escapes the main thread). On the opposite, the pair (5, f, 8) cannot be removed by this analysis since the location has already escaped the main thread at point 5. This pair is removed by the aliasing analysis. Our Escape analysis improves on that of Naik and Aiken which does not distinguish among several allocations performed at the same site.

Conditional Must Not Alias Analysis The flow-insensitive element Σ of the previous type and effect system is used to define an under-approximation DR_{Σ} of the notion of *disjoint reachability*. Given a finite set of heaps $\{\sigma_1, \dots, \sigma_n\}$ and a set of allocation sites H , the disjoint reachability set $DR_{\{\sigma_1, \dots, \sigma_n\}}(H)$ is the set of allocation sites h such that whenever an object o allocated at site h may be reachable by one or more field dereferences for some heap in $\{\sigma_1, \dots, \sigma_n\}$, from objects o_1 and o_2 allocated at any sites in H then $o_1 = o_2$. It allows to remove the last potential race of our running

example. For each potential conflict between two program points i_1 and i_2 , we first compute the set May_1 and May_2 of sites that the corresponding targeted objects may points-to, using the previous points-to analysis. Then we use the must-lock analysis to compute the sets $Must_1$ and $Must_2$ of allocation sites such that for any h in $Must_1$ (resp. $Must_2$), there must exist a lock l currently held at point i_1 (resp. i_2) and allocated at h . The current targeted object must furthermore be reachable from l with respect to the heap history that leads to the current point. This last property is ensured by the path expressions that are computed with a symbolic operand stack during the must-lock analysis. At last, we remove the potential race if and only if $Must_1 \neq \emptyset$, $Must_2 \neq \emptyset$ and

$$May_1 \cap May_2 \subseteq DR_{\Sigma}(Must_1 \cup Must_2)$$

We formally prove that any potential race that succeeds this last check is not a real race.

6 Related work

Static race detection Most works on static race detection follow the lock based approach, as opposed with event ordering based approaches. This approach imposes that every pair of concurrent accesses to the same memory location are guarded by a common lock and is usually enforced by means of a type and effect discipline.

Early work [5] proposes an analysis for a λ -calculus extended with support for shared memory and multiple threads. Each allocation comes in the text of the program with an annotation specifying which lock protects the new memory location and the type and effect system checks that this lock is held whenever it is accessed. More precisely, the annotation refers to a lexically scoped lock definition, thus insuring unicity. To overcome the limitation imposed by the lexical scope of locks, existential types are proposed as a solution to encapsulate an expression with the locks required for its evaluation. This approach was limited in that it was only able to consider programs where all accesses are guarded, even when no concurrent access is possible. Moreover, it imposed the use of specific constructions to manage existential types.

A step toward treatment of realistic languages was made in [7] which considers the JAVA language and supports various common synchronization patterns, classes with internal synchronization, classes that require client-side synchronization and thread-local classes. Aside from additional synchronization patterns, the approach is similar to the previous one and requires annotations on fields (the lock protecting the field) and method declarations (locks that must be held at invocation time). However, the object-oriented nature of the JAVA language is used as a more natural mean for encapsulation. Fields of an object must be protected by a lock (an object in JAVA) accessible from this object. For example, $x.f$ may be protected by $x.g.h$ where g and h are final fields (otherwise, two concurrent accesses to $x.f$ guarded by $x.g.h$ could use different locks). Client-side synchronization and thread-local classes are respectively handled by classes parametrized by locks and a simple form of escape analysis. A similar approach, using ownership types to ensure encapsulation, was taken in [4, 3].

The analysis we consider here is that of [18, 17]. Thanks to the disjoint reachability property and to an heavy use of points-to analysis, it is more precise and captures more

idioms than those above. Points-to analysis also makes it more costly but it has been proved that such analyses are tractable thanks to BDD based resolution techniques [22].

Machine checked formalisation for multithreaded JAVA There is a growing interest in machine checked semantics proof. Leroy [13] develops a certified compiler from Cminor (a C-like imperative language) to PowerPC assembly code in COQ, but only in a sequential setting. Hobor *et al.* [8] define a modular operational semantics for Concurrent C minor and prove the soundness of a concurrent separation logic w.r.t. it, in COQ. Several formalisation of the sequential JVM and its type system have been performed (notably the work of Klein and Nipkow [11]), but few have investigated its multithreaded extension. Petri and Huisman [19] propose a realistic formalization of multithreaded JAVA bytecode in COQ, BICOLANO MT that extends the sequential semantics considered in the MOBIUS project. Lochbihler extends the JAVA source model of Klein and Nipkow [11] with an interleaving semantics and prove type safety. Aspinal and Sevcik formalise the JAVA *data race free guarantee* theorem that ensures that data races free program can only have sequentially consistent behaviors. The work of Petri and Huisman [9] follows a similar approach. The only machine checked proof of a data race analyser we are aware of is the work of Lammich and Müller-Olm [12]. Their formalisation is done at the level of an abstract semantics of a flowgraph-based program model. They formalise a locking analyses with an alias analysis technique simpler than the one used by Naik and Aiken.

7 Conclusions and future work

In this paper, we have presented a formalisation of a JAVA bytecode data race analysis based on four advanced static analyses: a context-sensitive points-to analysis, a must-lock analysis, a must-not thread escape analysis and a conditional must-not-alias analysis. Our soundness proofs for these analyses rely on three layers of semantics which have been formally linked together with simulation (and sometimes bisimulation) proofs. The corresponding COQ development has required a little more than 15.000 lines of code. It is available on-line at <http://www.irisa.fr/lande/datarace>.

This is already a big achievement and as far as we know, one of the first attempt to formally prove data race freeness. However the current specification is not executable. Our analyses are only specified as sets of constraints on logical domains as $(pcontext \rightarrow \mathbf{Prop})$. We are currently working on the implementation part, starting by an Ocaml prototype to mechanically check the example given in Section 2. Then we will have to implement in COQ the abstract domains and the transfer functions of each analysis, following the methodology proposed in our previous work [6]. Thanks to the work we have presented in this paper, these transfer functions will not have to be proved sound with respect to an operational semantics. It is sufficient (and far easier) to prove that they refine correctly the logical specification we have developed here. We plan to only formalise a result checker and check with it the result given by the untrusted analyser written in Ocaml. Extracting an efficient checker is a challenging task here because state-of-the-art points-to analysis implementations rely on such complex symbolic techniques as BDD [22].

Acknowledgment We thank Thomas Jensen and the anonymous TPHOLs reviewers for their helpful comments.

References

1. AMD. Amd64 architecture programmer's manual volume 2: System programming. Technical Report 24593, 2007.
2. D. Aspinall and J. Sevcík. Formalising java's data race free guarantee. In *Proc. of TPHOLS '07*, pages 22–37. Springer-Verlag, 2007.
3. C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In ACM Press, editor, *Proc. of OOPSLA '02*, pages 211–230, New York, NY, USA, 2002.
4. C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In ACM Press, editor, *Proc. of OOPSLA '01*, pages 56–69, New York, NY, USA, 2001.
5. C. Flanagan and M. Abadi. Types for safe locking. In *Proc. of ESOP '99*, pages 91–108, London, UK, 1999. Springer-Verlag.
6. D. Cachera, T. Jensen, D. Pichardie, and V. Rusu. Extracting a Data Flow Analyser in Constructive Logic. *Theoretical Computer Science*, 342(1):56–78, 2005.
7. C. Flanagan and S. N. Freund. Type-based race detection for java. In *Proc. of PLDI '00*, pages 219–232, New York, NY, USA, 2000. ACM Press.
8. A. Hobor, A. W. Appel, and F. Zappa Nardelli. Oracle semantics for concurrent separation logic. In *Proc. ESOP 2008*, volume 4960, pages 353–367. Springer-Verlag, 2008.
9. M. Huisman and G. Petri. The Java memory model: a formal explanation. In *Verification and Analysis of Multi-threaded Java-like Programs (VAMP)*, 2007. To appear.
10. Intel. Intel 64 architecture memory ordering white paper. Technical Report SKU 318147-001, 2007.
11. G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006.
12. P. Lammich and M. Müller-Olm. Formalization of conflict analysis of programs with procedures, thread creation, and monitors. In *The Archive of Formal Proofs*, 2007.
13. X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Proc. of POPL'06*, pages 42–54. ACM Press, 2006.
14. Nancy G. Leveson. *Safeware: system safety and computers*. ACM, NY, USA, 1995.
15. J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *Proc. of POPL '05*, pages 378–391, New York, NY, USA, 2005. ACM Press.
16. M. Naik. *Effective Static Data Race Detection For Java*. PhD thesis, Stanford University, 2008.
17. M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *Proc. of POPL '07*, pages 327–338, New York, NY, USA, 2007. ACM Press.
18. M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *Proc. of PLDI '06*, pages 308–319, New York, NY, USA, 2006. ACM Press.
19. G. Petri and M. Huisman. BicolanoMT: a formalization of multi-threaded Java at bytecode level. In *Bytecode 2008*, Electronic Notes in Theoretical Computer Science, 2008.
20. K. Poulsen. Tracking the blackout bug, 2004.
21. Sun Microsystems, Inc. JSR 133 Expert Group, Java Memory Model and Thread Specification Revision, 2004.
22. J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proc. of PLDI'04*, pages 131–144. ACM, 2004.