Racer: Effective Race Detection Using AspectJ

Eric Bodden
Sable Research Group
McGill University
Montréal, Québec, Canada
eric.bodden@mail.mcgill.ca

Klaus Havelund
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA, USA
klaus.havelund@jpl.nasa.gov

ABSTRACT

Programming errors occur frequently in large software systems, and even more so if these systems are concurrent. In the past researchers have developed specialized programs to aid programmers detecting concurrent programming errors such as deadlocks, livelocks, starvation and data races.

In this work we propose a language extension to the aspect-oriented programming language AspectJ, in the form of three new pointcuts, `lock()`, `unlock()` and `maybeShared()`. These pointcuts allow programmers to monitor program events where locks are granted or handed back, and where values are accessed that may be shared amongst multiple Java threads. We decide thread-locality using a static thread-local objects analysis developed by others. Using the three new primitive pointcuts, researchers can directly implement efficient monitoring algorithms to detect concurrent programming errors online. As an example, we expose a new algorithm which we call RACER, an adoption of the well-known ERASER algorithm to the memory model of Java.

We implemented the new pointcuts as an extension to the AspectBench Compiler, implemented the RACER algorithm using this language extension and then applied the algorithm to the NASA K9 Rover Executive. Our experiments proved our implementation very effective. In the Rover Executive RACER finds 70 data races. Only one of these races was previously known. We further applied the algorithm to two other multi-threaded programs written by Computer Science researchers, in which we found races as well.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Monitors*

General Terms

Algorithms, Languages, Reliability, Verification

Keywords

Race detection, runtime verification, aspect-oriented programming, semantic pointcuts, static analysis

ISSTA'08, July 20–24, 2008, Seattle, Washington, USA.

1. INTRODUCTION

Programming errors occur frequently in software systems and therefore researchers have spent much effort on developing methods to detect and remove such errors as easily and early as possible in the development process. Concurrent programs are even more likely to suffer from programming errors, as concurrent programming adds potential sources of failure. In a concurrent program a programmer has to make sure to avoid deadlocks, to properly protect shared state from data races and to protect single threads or processes from starvation. Researchers have developed specialized static and dynamic analyses to aid programmers with these tasks [4, 5, 16, 23, 28–30, 35, 42, 43].

All these approaches share one common concern. They identify events of interest, such as the acquisition and release of locks or the access to shared state. Static approaches analyze source locations that trigger those events at runtime. Dynamic approaches instead intercept those runtime events directly, as the program under test executes. Up to now, most existing approaches use their own means to detect those events, through some form of program analysis and, in the dynamic approach, through program transformation.

In this work we propose an aspect-oriented approach to the problem, by exposing a language extension to the aspect-oriented programming language AspectJ. Aspect-oriented programming allows programmers to use predicates, called pointcuts, to intercept certain events of interest at runtime. In AspectJ, the events that can be intercepted are mostly restricted to method calls, field accesses and exception handling. The language extension which we propose enhances AspectJ with three new pointcuts, to make available to the programmer three additional kinds of events: (1) the acquisition of a lock, (2) the release of a lock and (3) the event of reading from or writing to a field that may be shared among threads.

For instance, the following pointcut captures the event of locking on object l: **lock**() && **args**(l). A programmer can capture the converse event of unlocking l by writing simply writing **unlock**() && **args**(l). Setting a potentially shared field on an object o is captured via the pointcut **set**(!**static** $*$) && **target**(o) && **maybeShared**().

Matching the first two pointcuts against a given program is decidable. The problem of matching the **maybeShared**() pointcut is however generally undecidable. We therefore compute a sound over-approximation using a static thread-local objects analysis [27]. The approximation assures that every access to a field that is indeed shared will be matched by the pointcut. Because of the over-approximation, the pointcut may however also match accesses to fields that are not actually shared, i.e. are only accessed by a single thread.

Using these three novel pointcuts, programmers can implement bug finding algorithms for concurrent programs very easily. The **lock**() and **unlock**() pointcuts allow a programmer to uniformly act on any acquisition and release of a lock using **synchronized** blocks and methods in any Java program. The programmer can use the **maybeShared**() pointcut to gain runtime efficiency by monitoring accesses to only those fields that may be shared among threads.

To demonstrate the feasibility of the approach, we implemented the three novel pointcuts as an extension to the AspectBench Compiler [6]. To show how programmers can make use of this language extension, we adopted a special version of the ERASER race detection algorithm [36] to Java, which we call RACER. Both ERASER and RACER detect program executions with a very high potential for a data race. We then applied the aspects implementing the RACER algorithm to a test harness of the NASA K9 rover and two other multi-threaded programs written by Computer Science researchers. Our results show that the algorithm is effective in finding data races. In the NASA code RACER found 70 races, 69 of which were previously unknown, although extensive studies had been performed on the K9 rover code before. In the other two programs we found, respectively, six and seven races.

The main contributions of this work are:

- a description of three novel AspectJ pointcuts, **lock**(), **unlock**() and **maybeShared**(),
- an implementation of these pointcuts in the AspectBench Compiler, in case of the **maybeShared**() pointcut through a static whole-program analysis,
- a novel algorithm for race detection in Java, coined RACER, and an implementation using the three novel AspectJ pointcuts, and
- experiments that show that our implementation is effective in efficiently finding data races in a test harness of the NASA K9 rover and two other benchmark programs.

2. EXAMPLE PROGRAM

In Figure 1 we show an example program that exposes a data race. The class `Task` holds static fields `shared` and `shared_protected`, as well as an instance field `not_shared`. Within its `run` method each task prints the value of each field, incrementing its value. The programmer protected access to the field `shared_protected` by synchronizing on the object `Task.class`. The program's `main` method creates two `Task` objects and runs each of them in a separate thread.

Both threads execute concurrently, without any synchronization. The program accesses the field `shared_protected` correctly because the programmer protected accesses to this field by consistently locking on the `Task.class` object. Accesses to `not_shared` may occur unprotected because every thread accesses the field of a different `Task` instance, and therefore accesses a memory location different from the one accessed by the other thread. However, the program accesses the field `shared` through both threads `thread1` and `thread2`, without proper synchronization—a data race.

Algorithms that wish to detect such data races or similar programming errors in concurrent programs generally need to capture two types of events, (1) locking and unlocking a particular object, and (2) accesses to fields, particularly fields that are accessed through different threads. Traditionally, bug detection tools would instrument the program under test (e.g. the one from Figure 1) to emit these events at runtime. A special runtime environment would then monitor the events and a report programming error as the error is detected. Programming the automated instrumentation packages that are usually used for those purposes is a tedious and time consuming task.

```
class Task implements Runnable {

    static int shared;
    static int shared_protected;
    int not_shared;

    public void run() {
        System.out.println(shared++);
        synchronized(Task.class) {
            System.out.println(shared_protected++);
        }
        System.out.println(not_shared++);
    }

    public static void main(String[] args) {
        Task task1 = new Task();
        Task task2 = new Task();
        Thread thread1 = new Thread(task1);
        Thread thread2 = new Thread(task2);
        thread1.start ();
        thread2.start ();
    }
}
```

Figure 1: Example program exposing a data race

3. ASPECTJ LANGUAGE EXTENSION

Aspect-oriented programming is a programming style that allows programmers to implement special "crosscutting concerns" in a modular way and then combine these concerns with a base program through a process called weaving. Researchers have identified [19,38] long ago that runtime monitoring for bug detection is one such crosscutting concern. Some bug detection tools nowadays therefore instrument programs by generating aspects in an aspect-oriented programming language, for instance AspectJ for Java-based programs or AspectC for programs written in C. The bug detection tools then weave these aspects into the program, using a standard compiler.

Up until now it was however not possible to develop bug detection tools based on aspects that would detect programming errors related to concurrency. This is because traditional aspect-oriented programming languages do not allow a programmer to detect lock and unlock events in their programs. Therefore, many bug detection tools for concurrent programming resort to low level bytecode instrumentation libraries that are relatively cumbersome to use. In this section we describe how we extended AspectJ to eliminate this shortcoming. Further, we describe how to implement and use another language extension, the `maybeShared()` pointcut. This pointcut allows programmers to match on accesses to fields that are potentially shared amongst threads. This may in most cases be more efficient than monitoring accesses to *all* fields in a program.

3.1 The pointcuts lock() and unlock()

When writing a concurrent Java program, a programmer has nowadays multiple ways to implement a locking policy. For example with Java 5, Sun introduced the new library `java.util.concurrent`, which exposes classes like `ReentrantLock`. A programmer locks and unlocks instances of `ReentrantLock` by invoking methods on the lock itself. The programmer can easily intercept such events using AspectJ's method call pointcuts.

In general, another style of locking is however much more pervasive: the use of `synchronized` blocks and methods. As we showed in lines 9-11 of Figure 1, programmers can use synchronized blocks to protect a region of code with a certain object that serves as a lock. Program control only enters this region when it can successfully acquire a lock on the given object (in Figure 1 on `Task.class`). The lock is automatically released when control leaves the block (either by throwing an exception or by normal flow). This way, locks are automatically ensured to be properly nested. For convenience, programmers can also flag methods with the `synchronized` modifier.

Using regular AspectJ, programmers can write pointcuts to match on method modifiers, and therefore can pick out calls to synchronized *methods*. However, it is not possible to match on the acquisition and release of locks using synchronized *blocks*. This prevents researchers from using AspectJ to implement bug detection algorithms for concurrent programs. Our proposed `lock()` and `unlock()` pointcuts overcome this shortcoming.

Syntax and semantics.

The pointcut `lock()` matches whenever the program acquires a lock, by entering either a synchronized block or method. The pointcut `unlock()` matches whenever control flow leaves such a block or method. A programmer can expose the object that is locked, respectively unlocked by conjoining the `lock()`, respectively `unlock()` pointcut with an `args(..)` pointcut. Figure 2 shows two example advice attached to `lock()` pointcuts that execute before, respectively after successful acquisition of a lock. The pointcut `args(l)` binds the variable `l` to the object that is locked on. Note that the declared type of `l` is `TaskQueue`. Because of that, the pieces of advice do not execute if a lock is claimed that is not of type `TaskQueue`. Our implementation does not even insert instrumentation into those places of the program at all. If the programmer instead wishes to match on any lock that is acquired or released, regardless of the lock's type, she can use the declared type `Object`, as this is the super-type of all reference types.

The Technical Report version of this paper [11] gives more details about our concrete implementation of both `lock()` and `unlock()` pointcuts.

```
before(TaskQueue l): lock() && args(l) {
    System.out.println("About to acquire a lock on "+l);
}

after(TaskQueue l): lock() && args(l) {
    System.out.println("Successfully acquired a lock on "+l);
}
```

Figure 2: Logging lock acquisition with our AspectJ language extension

3.2 The pointcut maybeShared()

We named the third and last pointcut of our AspectJ extension `maybeShared()`. This is because it matches all field accesses (reading or writing) that *may be shared*, i.e. whose field may be read from or written to by multiple threads. The word "may" here suggests that the semantic definition of this pointcut is somewhat fuzzy. This is however not the case. We can rigorously define the semantics of this pointcut through the following two invariants.

1. The pointcut `maybeShared()` matches only field read or write statements.

2. If a statement reads from a field or writes to a field and this field *is* read from or written to by multiple threads (through this and/or other statements), then `maybeShared()` matches this statement.

Note that the second invariant is unidirectional. In other words, `maybeShared()` is required to match accesses that are indeed shared, but it *may* also match other field accesses. The crucial point is that this over-approximating definition enables sound optimizations for many algorithms that attempt to find programming errors in concurrent programs at runtime.

By the definition of `maybeShared()`, one sound implementation of this pointcut would be to match all field read or write statements in the entire program. Algorithms making use of the `maybeShared()` should take this into account and therefore not rely on certain statements *not* being matched. Our RACER algorithm for example (Section 4) works correctly with such an implementation. However, the purpose of the `maybeShared()` pointcut is of course to make it match only as many statements as necessary but as few statements as possible. For instance, in our running example (Figure 1), we would like to match the field accesses in lines 8 and 10 but not 12, because the field in line 12 is not shared among different threads. With such an implementation, a programmer can conjoin `maybeShared()` with other pointcuts to gain an implementation that is automatically optimized by focusing on shared field accesses. For instance, the following pointcut, taken from our RACER implementation, is guaranteed to match all statements where a shared static field is set. It *could* further match some write accesses to static fields that are not shared, i.e. which only one thread accesses.

```
pointcut staticFieldSet():
    set(static * *) && maybeShared();
```

In the following we expose an efficient implementation of the `maybeShared()` pointcut that uses a static whole-program analysis to make it match fewer unshared field accesses than the unoptimized implementation.

3.2.1 An effective implementation

The implementation of `maybeShared()` that we expose in our *abc* compiler uses a compiler feature called *reweaving*. First we use the compiler to weave our RACER implementation (and/or any other aspects present), containing the `maybeShared()` pointcuts, into the program under test. To prepare the weaving, *abc* first matches all pointcuts against all statements in the program and so generates a "weaving plan", containing instructions about which aspect code to weave where. Then *abc* performs the actual weaving according to this plan. In a next step, we analyze all field access statements in this weaving plan using a thread-local objects analysis of the entire woven program. The thread-local objects analysis tells us which objects are definitely thread-local, i.e. not read from or written to by multiple threads. We then alter the weaving plan to not match the `maybeShared()` pointcut at statements which read from or write to such thread-local objects. In a last step we undo the initial weaving procedure, i.e. we un-weave the woven program to restore its original code, and then reweave the program using the optimized weaving plan. In result, `maybeShared()` does not match any field access where our thread-local objects analysis was powerful enough to prove thread-locality of the field being accessed.

Thread-local objects analysis.

The thread-local objects analysis we use is not a contribution of this paper. It was developed by Halpert et al. for the purpose of component-based lock allocation. In their paper [27], the authors describe the approach in detail (Section 3 there). We here only outline the analysis process.

The thread-local objects analysis runs in different stages. First, the analysis builds a call graph for the entire program. It also uses the flow-insensitive points-to analysis in Spark [33] to build points-to sets. An analysis can use points-to sets to statically estimate whether two variables may point to the same objects.

In a second stage, Halpert et al. create information-flow summaries for every reachable method in the program. The summaries describe how data, in particular objects, may flow from a method's parameters to its return value or to other methods.

The actual thread-local objects analysis (TLO) then executes as a third stage. To quote Halpert et al., "*TLO classifies all fields as either thread-local or thread-shared, where any field that may be accessed by more than one thread is thread-shared and all others are thread-local.*" The analysis inspects one thread creation site t at a time. First, the analysis enumerates all methods that may be executed through t. Then the analysis flags every field accessed by a method external to t as thread-shared. All other fields are classified as thread-local. A similar classification applies to method parameters. If a method outside t calls a method m, then m's parameters are considered thread-shared, otherwise thread-local.

In a next step, the thread-local objects analysis makes use of the information-flow analysis to propagate information about shared fields through methods. Whenever the information-flow analysis indicates that a shared value may flow to a field that, until now, was classified as thread-local, the analysis changes this classification to thread-shared. The process is then repeated with the new classification until the analysis reaches a fixed point.

Lastly, an interprocedural stage propagates this information along method calls, again until a fixed point is reached. This stage also combines the information for all the different threads to a common data structure. In result, when the programmer queries the thread-local objects analysis for a field f, the analysis reports this field as thread-local only if it has not been classified as thread-shared for any thread.

The thread-local objects analysis is demand-driven; querying it on fewer variables will decrease the analysis time. We query the analysis for any field access that is matched by a `maybeShared()` pointcut, but only after the rest of the pointcut matching has completed. For instance, the pointcut

set(**static** $*$ $*$) && **maybeShared**()

matches only writes to static fields. If the programmer applies this pointcut to the example program from Figure 1, then we only query the thread-local objects analysis for the fields `shared` and `shared_protected` because `not_shared` is non-static and therefore the value of `maybeShared()` does not matter. This "lazy querying" makes the approach relatively efficient, as the thread-local objects analysis may be queried comparatively sparsely.

4. RACER ALGORITHM

To demonstrate how to use our AspectJ language extension, we implemented a novel algorithm called RACER, a variant of the ERASER algorithm for data-race detection by Savage et al. [36]. The goal of RACER is to detect the potential for data races at runtime, just as in ERASER. Therefore RACER has some parts in common with the ERASER algorithm. However, its semantics is closer to Java's memory model [26] and, as we will see, RACER therefore can detect data races in Java programs that ERASER would miss.

4.1 Lock sets

Both algorithms keep lock sets, as follows. The idea is to maintain for each field f a set of candidate locks $L(f)$. At each point of a program execution, the set $L(f)$ contains the lock objects that all threads could agree on using when accessing the field f so far. We qualify a field by its owner. For a static field f of a class C we maintain a lock set $L(C.f)$, for an instance field f of an object o we maintain the set $L(o.f)$. Our implementation uses weak identity hash maps to implement this mapping. Such maps compare keys on object reference identity (opposed to equality). This is necessary because we wish to store different lock sets for different objects, regardless of whether these objects are equal according to the `equals(..)` method. Further, these maps automatically dispose of entries whose key got garbage collected. This is to prevent our implementation from causing memory leaks. Note that this form of memory management is sound. If an object o gets garbage collected, no thread can access its fields any more. Therefore, no field of o can be part of a race on the remainder of the execution.

As the program under test starts up, lock sets are assumed to hold "all possible objects". As there is no way to enumerate all those objects, we use a special marker set to implement this semantics. Furthermore we maintain one lock set $L_T(t)$ for each thread t. At any time, it holds the locks currently owned by t. One AspectJ aspect, `Locking`, keeps track of these lock sets using a thread-local variable, as we show in Figure 3. Because Java's locks are reentrant, we use a bag instead of a set. In lines 3-6 we declare the thread-local variable `locksHeld` and initialize it to an empty bag. Then, whenever the program claims a lock `l`, we add this lock to the bag of the current thread (lines 8-11). Whenever the program releases a lock `l`, we remove it from the bag (lines 13-17). (The conjunct "`&& Racer.scope()`" avoids the pointcuts from matching within our own RACER implementation and therefore avoids potential infinite recursion.) As the reader can see, this way of implementation is very direct. No additional instrumentation phase is necessary, as the AspectJ weaver takes care of the entire weaving process.

```
1  public aspect Locking {
2
3      ThreadLocal locksHeld = new ThreadLocal() {
4          protected synchronized Object initialValue() {
5              return new HashBag();
6          } };
7
8      before(Object l): lock() && args(l) && Racer.scope() {
9          Bag locks = (Bag)locksHeld.get(); //thread−local copy
10         locks.add(l);
11     }
12
13     after(Object l): unlock() && args(l) && Racer.scope() {
14         Bag locks = (Bag)locksHeld.get(); //thread−local copy
15         assert locks.contains(l);
16         locks.remove(l);
17     }
18 }
```

Figure 3: Aspect bookkeeping thread-local lock sets

An additional advantage of using AspectJ is that we could easily modify the `Locking` aspect to take other kinds of locking into account. For instance, if `ReentrantLock`s were used (see Section 3), we could just extend the pointcuts in Figure 3 with an additional conjunct, e.g. replacing line 8 by:

```
before(Object l):
( lock() && args(l) ||
  call(void ReentrantLock.lock()) && target(l) )
&& Racer.scope() { ...
```

This allows researchers and programmers to be very flexible in the choice of locks and how they are required.

Whenever the program under test accesses a field, a second aspect, `Racer`, is notified. We show the essential parts of this aspect in Figure 4. The aspect first declares four different pointcuts that match reads and writes from, respectively to, static fields and instance fields (lines 3-10). Note that we use the `maybeShared()` pointcut because we are not interested in accesses to fields that cannot possibly be shared amongst threads.

Two pieces of advice follow. The first, in lines 12-17, executes right before a static field is written to. The advice first extracts the field's name, the declaring class and the source location from the special constant `thisJoinPointStaticPart`. The combination of declaring class and field name makes up our qualified field name $C.f$. We use the source location to be able to tell the programmer where a field was accessed, if this access is part of a race.

The constant `thisJoinPointStaticPart` is generated by the AspectJ compiler and holds all statically available information about the intercepted point in program execution (the joinpoint). Because this information is statically available, the compiler implements an optimized compilation strategy to generate this constant. Any use of `thisJoinPointStaticPart` is therefore very efficient. In addition we wish to note that, although we access information *about* the monitored field, we never access the field itself. Therefore, our own implementation cannot itself cause a data race in the base program[1].

[1]Our RACER implementation could however have data races within its own code. We therefore applied a copy of RACER to RACER itself [10] to validate that there were no races in our implementation. We found none.

```
1  aspect Racer {
2
3    pointcut staticFieldSet():
4      set(static * *) && maybeShared();
5    pointcut fieldSet(Object owner):
6      set(!static * *) && target(owner) && maybeShared();
7    pointcut staticFieldGet():
8      get(static * *) && maybeShared();
9    pointcut fieldGet(Object owner):
10     get(!static * *) && target(owner) && maybeShared();
11
12   before(): staticFieldSet () && scope() {
13     String id = getId(thisJoinPointStaticPart);
14     Class owner = getDeclaringClass(thisJoinPointStaticPart);
15     SourceLocation loc = getLocation(thisJoinPointStaticPart);
16     fieldSet (owner,id,loc );
17   }
18
19   before(Object owner): fieldSet(owner) && scope() {
20     String id = getId(thisJoinPointStaticPart);
21     SourceLocation loc = getLocation(thisJoinPointStaticPart);
22     fieldSet (owner,id,loc );
23   }
24   ...
25 }
```

Figure 4: Aspect updating per-field lock sets on field access

We then call the method `fieldSet(..)` to actually register the field write event, as follows. The `Racer` aspect asks the `Locking` aspect for the lock set $L_T(t)$ of the currently executing thread. Then, `Racer` refines the lock set $L(C.f)$ of the field with $L_T(t)$:

$$L(C.f) := L(C.f) \cap L_T(t)$$

This is because, if the programmer uses a lock *consistently* to protect the field $C.f$, meaning it will remain in $L(C.f)$ during all refinements, then this lock in fact protects all accesses to $C.f$.

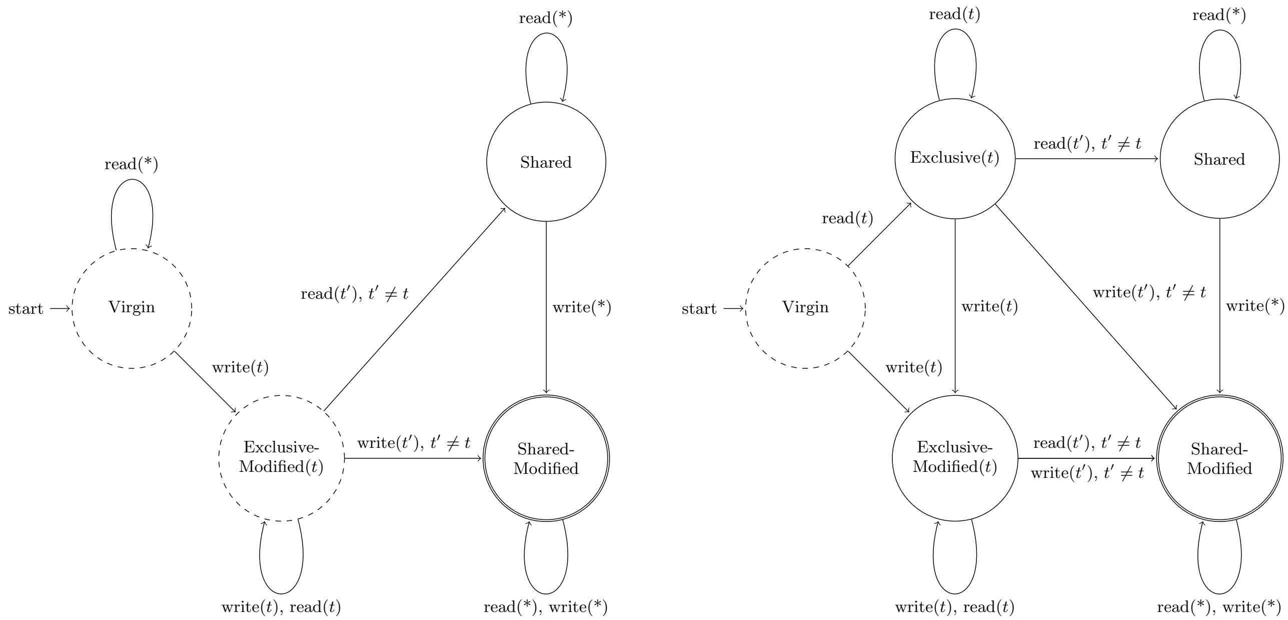
The second piece of advice in lines 19-23 of Figure 4 performs the same update for instance fields, this time with the `owner` as the field's qualifier instead of the declaring class. The aspect furthermore contains two other pieces of advice that register reading field accesses in the very same manner, with the same updates to the fields' lock sets (not shown).

Once a lock set for a field becomes empty, this means that the programmer used no consistent lock for this field over the entire execution of the program. If the field is shared among threads, this indicates a high potential for a data race.

4.2 State machine

The updates to lock sets presented so far are identical to the updates that Savage et al. described in the ERASER algorithm [36]. As Savage et al. point out however, this simple locking discipline is too strict. For instance (1) it should be okay for a variable v's lock set to become empty if this variable is only ever accessed by one thread. Furthermore (2) one should not report potential for read-read races, as such races can never lead to inconsistent visible data. Because the ERASER algorithm was originally developed for C programs it even dealt with another idiom, (3) where variables are frequently initialized without holding a lock (and in C it is commonly safe to do so).

Savage et al. took care of these constraints by framing the state machine shown in Figure 5a around the lock set



(a) Eraser state machine

(b) Racer state machine

Figure 5: Eraser and Racer state machines; dashed states do no lock refinement; double-lined states report race potential

refinement algorithm from Sub-section 4.1. One stores one instance of this state machine for every monitored variable. Each variable initially starts in a Virgin state. Once the variable is initialized by a thread t, it is confined to t by moving into state Exclusive-Modified(t)[2]. While in this state, the variable is considered in (3) its initialization phase—t may write to and read from the variable. Lock sets only begin to be refined when another thread t' accesses the variable, entering the state Shared respectively SharedModified, an indication that property (1), single-threaded access, does not hold. To avoid (2) reporting potential for read-read races Eraser only signals potential for a race if a lock set becomes empty while in state SharedModified, not in Shared.

As noted above, Eraser grants an explicit initialization phase for each variable—a phase which is assumed safe. This approach is very reasonable in C. In Java the situation is different, however. In Figure 6 we give a subtle example of unsafe un-synchronized field accesses even during object initialization. We adapted the example from Brian Goetz's book on Java concurrency [24]. The class constructor creates an object of an anonymous inner class which it then registers as an event listener. The problem is that as soon as this registration has happened, the `doSomething()` method may be executed by the event dispatch thread of Sun's Abstract Window Toolkit (AWT), as this thread takes care of notifying event handlers. Furthermore, the method has access to its parent `ThisEscape` object, via an implicit `this` reference. However, this object has not yet been fully constructed. Therefore, according to the revised Java Memory Model [26], if this listener were to access for instance the field `i`, it would not be clear, which value it would read. We have just witnessed a very subtle data race. We believe that in particular such subtle data races are very hard to find and hence should be reported by our tool, just as any other race. As our experiments in Section 5 will show, quite many data races may fall into this category. Because many tools

[2]This state was called Exclusive in [36] but the name Exclusive-Modified(t) suits our comparison better.

```
class ThisEscape {
    int i;

    ThisEscape(EventSource source) {
        source.registerListener(
            new EventListener() {
                public void onEvent(Event e) {
                    doSomething(e);
                }
            }
        );
        i = 42;
    }
    ...
}
```

Figure 6: Constructor letting an implicit reference to `this` escape to the AWT event dispatch thread; c.f. [24], page 41

for Java imitate C-based algorithms like Eraser, in some Java programs such races went undetected for years.

For Racer we therefore decided that we would still like to refrain from reporting variables that are (1) accessed by a single thread only, or (2) expose only potential for read-read races. However, following the Java Memory Model, we decided to drop property (3), i.e. Racer does not assume a variable's initialization as safe. Initialization instead *has* to occur while holding a suitable lock. In Figure 5b we show a state machine that follows these design decisions. In contrast to Eraser, Racer refines lock sets immediately, not only after a second thread has accessed the variable. Furthermore, through the new state Exclusive(t), we regard sequences "read(t) write(t')" as an indication for a possible race, because without proper synchronization this sequence could just as well have been "write(t') read(t)", given a different scheduling order. In this case the write performed by t' would not necessarily be visible to t. The Exclusive and ExclusiveModified states take care of (1) single-threaded access, while the Shared state again takes care of (2) not reporting potential for read-read races, just as in Eraser.

5. CASE STUDY

We applied Racer to an experimental planetary rover controller, named the *K9 Executive*, for a rover named K9 developed at NASA Ames Research Center. In the following, we briefly introduce this application, followed by the results of applying Racer.

5.1 The K9 Rover and Executive

The K9 Rover is an experimental hardware platform for autonomous wheeled rovers, targeted for the exploration of a planetary surface such as Mars. K9 is specifically used to experiment with new autonomy software. Rovers are traditionally controlled by low-level commands uploaded from Earth. The K9 Executive, a software module, provides a more flexible means of commanding a rover through the use of high-level plans in a domain specific programming language. High-level plans can for example be generated by an on-board AI-based planner. The Executive is essentially an interpreter for the plan language.

The Executive is multi-threaded. In Figure 7 we show the threads relevant for the presentation in this paper as boxes. All threads are started from a main program in the class `Main`. The `RuntimeExecutive` thread is responsible for the overall execution of plans. The interpretation of a primitive plan element, a task, causes the `RuntimeExecutive` to ask the `ActionExecution` thread to command the vehicle to perform the task's action. The `ActionExecution` thread subsequently is responsible for commanding the vehicle and reporting back the status. The `ActionExecution` thread updates a database whenever the status of a vehicle component changes. The `ExecCondChecker` (composed of two separate threads) monitors changes in the database (`DbMonitor`), prioritizes the changes and signals back the `RuntimeExecutive` through `Filter`.

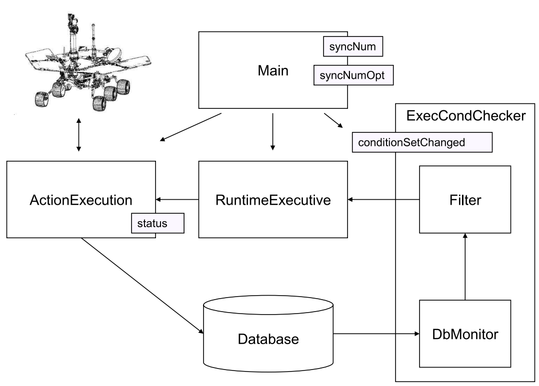


Figure 7: The K9 Executive and 4 of its 68 unprotected fields

The Rover is programmed in almost seven thousand lines of Java and is an abstracted version of 35 thousand lines of C++, originally controlling the rover, also developed at NASA Ames Research Center. Of the 35 Kloc C++ code, 9.6 Kloc are related to core functionality and the rest is for data structure manipulation (modules for specific rovers and science instruments) and research-related extensions. Here the main focus is on the core functionality. The Java version of this code was developed specifically for use in an evaluation of Java verification tools conducted at NASA Ames Research Center, as described in [14]. These tools included the Java PathExplorer [30], which contained an earlier implementation of the Eraser algorithm for Java, see Section 7. Researchers further used this code to evaluate the Java PathFinder model checker [41], which also contained a version of Eraser; a static analysis tool for C (for a C version of the code), and temporal logic specification monitoring. To evaluate these tools, a control team seeded errors in the code, and different groups of people were tasked with detecting the errors using different tools. After this experiment the code was augmented with additional code to evaluate a deadlock analysis tool. From [14] we can cite: "*A total of 12 bugs were extracted from the CVS logs, of which 5 were deadlocks, 2 were data races, and 5 were plan-related. One of the deadlock bugs was given as an example during training on the tools, and one of the data races was unreachable in the code that was eventually analyzed – thus leaving only 10 seeded errors*". This suggests that the code contains one data race, and in fact before our experiments the developers of this code were only aware of this single race.

5.2 Application of Racer

At first, we were therefore surprised to see that running Racer on the K9 Executive revealed 70 data races. These races can be categorized into three classes:

- 1 known data race on `ActionExecution.status`.
- 2 data races on variables `syncNum` and `syncNumOpt` in `Main`, which had been just recently introduced.
- 67 data races on various variables initialized in constructors, which went undetected for a long time.

The last group contains two pairs of races which occur on the same field each. This means that the K9 Executive accesses a total of 68 fields without proper protection.

5.2.1 The Race on ActionExecution.status

Racer issues the following message to indicate a data race on a variable `status` in class `ActionExectution`:

```
Race condition found!
Field 'int ActionExecution.status'
is accessed unprotected.
Owner object: 6171853
----------------------------------------
Read at ActionExecution.java:233:5
Read at ActionExecution.java:244:12
Write at ActionExecution.java:370:4
```

This is a race caused by the `ActionExecution` thread and the `RuntimeExecutive` thread both accessing this variable without both first acquiring a common lock. This is exactly the error planted in the code during the original verification experiment [14].

5.2.2 The Races on syncNum and syncNumOpt

For reasons of brevity we will not show the error messages from Racer for the remaining data races. The two data races mentioned in this section stem from an experiment performed with the K9 Executive (after the case study from [14]) in order to determine the efficiency with which a static analysis algorithm could reduce the number of locking operations needed to be monitored dynamically in order to detect a deadlock. For this purpose two integer counters

```
void initConditionChecker() {
  conditionSetChanged = false; // access variable
  checkerDBChanged = false;
  savedWakeupStruct = null;
}

public ExecConditionChecker(
  Database xdb, Executive xexec,
  ExecHasThreadedComponents xparent
)
{
  checkerThread =
    new ExecConditionCheckerThread(this,null);
  dbThread =
    new ExecDBMonitorThread(this, null);
  db = xdb;
  exec = xexec;
  initConditionChecker(); // accesses variable
  checkerThread.start();  // accesses variable
  dbThread.start();
}
```

Figure 8: Data race in `ExecCondChecker`

were introduced in the `Main` class: `numSync` (number of synchronizations executed in total) and `numSyncOpt` (number of synchronizations executed after optimization). It turned out that these two counters were updated by the different threads, not protected by any lock.

5.2.3 Constructor Initializations and Threads

Racer reports a data race on `conditionSetChanged` in the class `ExecCondChecker`. The code in Figure 8 shows the declarations of the function `initConditionChecker` and the constructor `ExecConditionChecker` to illustrate this situation. The constructor calls `initConditionChecker`, updating the variable, and then starts a `checkerThread`, which accesses `conditionSetChanged` (not shown). The problem is that the changes done to the fields within the constructor are not necessarily visible to the `checkerThread` without proper synchronization. If `conditionSetChanged` had been final, then its value would have been guaranteed to be visible at least *after* execution of the constructor (Section 17.5 of the Java Language Specification [26]). However, since the thread is started from *within* the constructor, the access to this field through the other thread would have been unsafe in this situation nevertheless. Starting threads from a constructor is very often unsafe for exactly this reason. The situation here reflects a bug pattern very similar to the one we saw in Section 4, Figure 6 under the name `ThisEscape`. There are 66 more similar races reported for this code. They are all caused by an assignment within a constructor.

False positives.

We wish to note that we could find only two false positives. All were in the third category of 67 races. In these cases, a `final` variable was initialized within an object's constructor by one thread and then accessed by another thread, without synchronization, but after execution of the constructor had finished. As noted above, such access is safe.

We could have excluded final variables from monitoring, simply by conjoining the pointcuts in our `Racer` aspect with:

$$!\textbf{set}(\textbf{final} * *) \ \&\& \ !\textbf{get}(\textbf{final} * *)$$

However, there can still be races involving final variables, for instance if a constructor itself starts a thread. We therefore opted to monitor final fields as well.

Observations

We confirmed all races that we found with the original developers of the code. Researchers analyzed this code earlier [14] but detected only the data race described in Sub-section 5.2.1. The errors described in Sub-section 5.2.2 were introduced at a later point and could therefore not be found at the time. However, the 67 data races described in Sub-section 5.2.3 all existed at the time. They were not detected due to way the initialization phase of the Eraser algorithm was implemented, regarding constructor execution as safe and not taking into account the Java memory model, as we explained in Section 4. Because Eraser does not refine lock sets in Exclusive states the races went unnoticed. This was the case also for the version of the Eraser algorithm implemented in the Java PathFinder model checker [41] and in Java PathExplorer [30] which were used in the earlier experiments. We conclude that the Racer algorithm or variants of it might be better suited for Java programs than Eraser because it follows Java's memory model more closely.

6. FURTHER EXPERIMENTS

In addition to our in-depth case study, we further applied Racer to two more, smaller benchmarks taken from [27]. This was to find out whether our results gained from the case study could be generalized to other programs as well. The two benchmarks are *roller* and *bank*. The benchmark *roller* simulates a roller coaster where "*7 passenger threads compete for 7 seats in 1 roller coaster thread*" [27]. This benchmark exposes very high contention. *bank* is a little banking application by Doug Lea [32]. It starts eight threads which each make a random transaction from one account to another and then call `Thread.yield()`. We note that both benchmarks were written by researchers in the field of concurrent programming. Nevertheless, using our Racer implementation we could find races in these programs as well.

Table 1 shows our experimental results for these two benchmarks and, to make the picture complete, additional numbers for our rover case study. Of each benchmark we present two versions: one without optimization of the `maybeShared()` pointcut and one where these optimizations are enabled.

We compiled the benchmarks on a Java HotSpot(TM) 64-Bit Server VM (build 1.6.0-rc-b104, mixed mode), but linked the benchmarks to Sun's JDK version 1.4.2_12, which we also used to run the benchmarks (with default heap space). Our machine used an AMD Athlon 64 X2 Dual Core Processor 3800+.

Compilation time.

The compilation time is low without optimization, generally below two minutes. The static whole-program optimization adds about one and a half minutes of compilation time to our smaller benchmarks. In case of the K9 rover however, compilation takes almost an hour to complete with optimizations enabled. We conjecture that this is partly due to Halpert et al.'s unoptimized implementation of the thread-local objects analysis and partly due to the programming style present in the rover code.

	roller	roller-opt	bank	bank-opt	rover	rover-opt	unit
compilation time	0:36	2:10	0:32	2:14	1:24	59:28	m:ss
no instrumentation	31	31	22	22	2	2	s
with instrumentation	334	332	535	538	2	2	s
last race detected after	0.07	0.07	0.07	0.07	0.78	0.78	s
instrumented fields	9	8	16	15	260	169	
with reported races	6	6	7	7	68	68	
with actual races	6	6	7	7	66	66	
reported races	6	6	7	7	70	70	
actual races	6	6	7	7	68	68	
races due to initialization	5	5	5	5	65	65	
regular un-synchronized access	1	1	0	0	3	3	
un-synchronized use of mutators	0	0	2	2	0	0	
false positives	0	0	0	0	2	2	

Table 1: Experimental results

Runtime.

The next section of Table 1 shows the runtimes for the different configurations. In case of the two small benchmarks, our instrumentation adds around 11-fold (*roller*) and 24-fold (*bank*) overhead. Through profiling we determined that much of this slowdown is caused by contention. Both benchmarks spend around 70% of their time waiting on a lock. When our instrumentation monitors a field access through a thread t, and this field access is within a synchronized region, then this forces t to reside longer in this region, to execute the instrumentation code. All other threads have to wait for t to finish in the meantime. This naturally increases the overall wait time. The code of the K9 rover does not show such high contention and indeed, in this benchmark we could perceive no overhead. The table further shows that in all three benchmarks, Racer reported all races within the first second of execution (we only report each race once). This suggests that even when the runtime overhead is quite high, this overhead might not actually cause any problems in practice. In addition, the programmer can opt to restrict instrumentation caused by Racer, simply by modifying the `scope()` pointcut used in Figures 3 and 4, e.g. to:

pointcut scope(): !**within**(package.with.no.monitoring.∗);

Instrumented fields.

Next we comment on the number of fields instrumented. The purpose of optimizing the `maybeShared()` pointcut was to reduce the number of instrumented fields by restricting the instrumentation only to fields that may be shared among threads. In *roller* and *bank* this was not very effective, since in both benchmarks all but one field indeed are shared. Therefore, the optimization was ineffective and the runtime was not improved. In case of the rover code, about one third of the 260 fields were detected as thread-local and not instrumented in the optimized version. However, since the rover code finished execution after two seconds already, there was no perceivable improvement in runtime either.

Detected races.

In *roller*, the Racer algorithm reported six races on six fields. There were no false positives. Five of the six races were due to the unprotected initialization anti-pattern which we saw already in the rover code. The remaining race occurred because of a field accidentally being accessed just before a `synchronized` block instead from within. In the *bank* benchmark seven races were reported, all valid, with five occurring through unprotected initialization and two others through a similar pattern. In these two cases, a setter sets a field in one thread and then another thread uses this field without synchronization. In Section 5 we already commented on the races in the Rover Executive.

7. RELATED WORK

The original Eraser data race algorithm [36] was implemented in Compaq's Visual Threads tool [28], now offered and maintained by HP. Programmers can use Visual Threads with any application that uses a POSIX threads library, which includes common implementations of Java. Visual Threads analyzes multi-threaded applications for potential logic and performance problems. The tool visualizes state changes and provides automated dynamic analysis algorithms to diagnose common problems associated with multi-threading, including deadlock, data protection, performance, and programming errors. Visual Threads uses the object code instrumentation tool ATOM [21]. Attempts have been made to improve the accuracy of the Eraser algorithm [35,42]. In [29] we describe our first implementation of the Eraser algorithm for Java, guiding the Java PathFinder (JPF) model checker [41] to confirm the warnings discovered by the much faster potential-analysis. We instrumented the programs under test by modifying the Java Virtual Machine of JPF. The algorithm was later re-implemented and elaborated in the Java PathExplorer (JpaX) tool [30], which used the Jtrek bytecode instrumentation tool [17] and later the BCEL bytecode instrumentation tool [18]. With the AspectJ language extension proposed in this paper, the use of Jtrek or BCEL becomes obsolete.

Other kinds of dynamic race analysis tools have been developed, which are characterized by detecting potentials for errors, like the Eraser algorithm, rather than directly detecting the occurrence of errors. Artho et al. proposed a high-level data race algorithm [4] which detects inconsistencies in which collections of variables are access protected by locks. If for example two variables x and y are accessed in one single synchronized block in one part of the program and in separate synchronized blocks in another part of the program, the algorithm considers this an inconsistent use, and issues a warning suggesting that the latter use is potentially unsafe. The algorithm is also called the view consistency algorithm, since it attempts to detect view inconsistencies during runtime. The absence of low-level and high-level data races still allows for other concurrency errors. Related to high-level data races are atomicity violations as detected by the tools in [5, 23, 43]. An example is a thread that reads a shared variable into a local variable, updates the local variable, and then writes back to the shared variable. The local variable may at some point become *stale* (out of date) if some other thread updates the shared variable. jPredictor [16] extracts a causality relation from the execution trace, sliced using static analysis and refined with lock-atomicity information. Two common types of errors are investigated in [16]: data races and atomicity violations. jPredictor's program instrumentor is built on top of the Soot [37] Java bytecode engineering package. The AspectBench Compiler used for our language extension uses Soot internally, to conduct the weaving process. However, the language extension hides these internals from the programmer behind a visually appealing syntax.

Programmers can also effectively use dynamic analyses to find potential for deadlocks. As mentioned, the Visual Threads tool detects deadlock potentials, essentially by detecting cycles in a lock graph. Bensalem and Havelund [9], and Agrawal et al. [1] improved this algorithm to reduce false positives. Agrawal et al. further suggest the use of deadlock types during a static analysis phase to reduce overhead during dynamic deadlock analysis by identifying synchronizations that can be regarded safe, and hence do not need to be monitored/recorded. This is similar to our static optimization of `maybeShared()` in that it tries to remove unnecessary monitoring overhead through an analysis at compile time. Concurrent programs may be modified by inclusion of wait statements or modifications to schedulers, so that a fuller range of non-deterministic behaviours are exhibited during testing. Such modifications can be combined with predictive analysis [8, 22].

All mentioned algorithms work without the need for the user to provide a specification. Several systems have been developed to monitor program executions against user provided formal specifications. The runtime verification community is concerned with program correctness. An example of such a system is Eagle [7]. Tracematches's [2] answer provides an efficient implementation of runtime monitoring with object bindings as a language extension to AspectJ. Bodden et al. [12, 13] used tracematches to prove Java and AspectJ programs partially correct. Tracematches can directly use the three novel pointcuts proposed here.

Apart from tracematches however, the typical scenario for the building of the above tools in the case of Java is the use of bytecode instrumentation tools. Examples are Jtrek [17], BCEL [18], Soot [37], and ASM [15]. Similar tools for other languages include Valgrind [40], ATOM [21], and the C source code instrumentation and analysis tool CIL [34]. Programmers can however further instrument programs through debugging interfaces, modification of the runtime system or virtual machine (as in the case of the Java PathFinder), or through operating system or middle-ware services. Attempts have been made to develop higher level libraries on top of the low level instrumentation packages. In previous work, for example, we developed the jSpy tool [25], which instruments Java byte-code, but using a higher level of primitives compared to what is offered by the low-level bytecode instrumentation tools. A jSpy instrumentation specification consists of a set of rules, each of which consists of a condition on byte-code and an instrumentation action stating what to report when byte-codes satisfying the condition are executed. The reported events are then picked up by monitors that in turn check for various user provided properties. The tool is oriented towards monitoring rather than functionality modification. Another high-level instrumentation tool is Sofya [31].

The main observation is that most, if not all, of the dynamic analysis tools described above all use low level instrumentation tools that are more or less difficult to use. An aspect oriented programming language with synchronization pointcuts makes this part of the work much simpler.

Since quite a while now, the community around aspect-oriented programming has been calling for more "semantic pointcuts" (e.g. [3, 20]), which allow programmers not to match on a program's structure like a call to a method `foo()`, but on more semantic properties. Generally we agree with this point of view. We therefore implemented the `maybeShared()` pointcut as an answer to that call. However, an implementation of such pointcuts that is efficient for arbitrary base programs is still out of sight and therefore we encourage further research in this area.

8. CONCLUSION AND FUTURE WORK

In this work we have proposed a language extension to the aspect-oriented programming language AspectJ. We extend AspectJ with three new pointcuts `lock()`, `unlock()` and `maybeShared()`. These pointcuts allow researchers to easily implement bug finding algorithms for concurrent programs. As an example, we have implemented RACER, an adaption of the ERASER race detection algorithm to the Java memory model. We found that using our AspectJ extension we were able to implement RACER very easily, in just two aspects with a small set of supporting classes.

The RACER algorithm is different from C-based race detection algorithms like ERASER in the way that it treats object initialization. ERASER is very forgivable to programmers in an object's initialization phase. RACER on the other hand detects and reports also races that comprise the initialization of an object. This revealed 70 data races in program code of the NASA K9 Rover Executive, 69 of which went previously undetected, although extensive studies of this code had already been performed at a time where 67 of these undetected races were already present.

In future work we plan to reduce the compile time overhead caused by the thread-local objects analysis that we use to implement the `maybeShared()` pointcut. Furthermore it would be interesting to see if the RACER algorithm could be extended to take more synchronization primitives into account, for instance the method `Thread.join()`. The use of our AspectJ language extension makes our implementation very flexible in that respect.

Acknowledgements & Program download.

We thank Clark Verbrugge for helping us validate some of the data races we found in the K9 rover executive. Also we are grateful to him, Richard Halpert, and Chris Pickett for making their thread-local objects analysis and their benchmarks available to us. We thank Stefan Savage for providing clarifications on ERASER. Part of the work described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Our RACER implementation and compiler are available at:

`http://www.aspectbench.org/`

9. REFERENCES

[1] R. Agarwal, L. Wang, and S. D. Stoller. Detecting potential deadlocks with static analysis and run-time monitoring. In Ur et al. [39], pages 191–207.

[2] C. Allan, P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In R. Johnson and R. P. Gabriel, editors, *OOPSLA*, pages 345–364. ACM, 2005.

[3] T. Aotani and H. Masuhara. Compiling conditional pointcuts for user-level semantic pointcuts. In *Software-Engineering Properties of Languages and Aspect Technologies (SPLAT)*, March 2005.

[4] C. Artho, K. Havelund, and A. Biere. High-level data races. *Software Testing, Verification and Reliability*, 13(4):207–227, 2003.

[5] C. Artho, K. Havelund, and A. Biere. Using block-local atomicity to detect stale-value concurrency errors. In F. Wang, editor, *ATVA*, volume 3299 of *LNCS*, pages 150–164. Springer, 2004.

[6] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. *abc*: An extensible AspectJ compiler. In *AOSD conference*, pages 87–98. ACM Press, 2005.

[7] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In B. Steffen and G. Levi, editors, *VMCAI*, volume 2937 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2004.

[8] S. Bensalem, J.-C. Fernandez, K. Havelund, and L. Mounier. Confirmation of deadlock potentials detected by runtime analysis. In *PADTAD '06: Proceeding of the 2006 workshop on Parallel and distributed systems: testing and debugging*, pages 41–50, New York, NY, USA, 2006. ACM.

[9] S. Bensalem and K. Havelund. Dynamic deadlock analysis of multi-threaded programs. In Ur et al. [39], pages 208–223.

[10] E. Bodden, F. Forster, and F. Steimann. Avoiding infinite recursion with stratified aspects. In R. Hirschfeld, A. Polze, and R. Kowalczyk, editors, *GI-Edition Lecture Notes in Informatics "NODe 2006 GSEM 2006"*, volume P-88, pages 49–64. Gesellschaft für Informatik, Bonner Köllen Verlag, 2006.

[11] E. Bodden and K. Havelund. Racer: Effective race detection using AspectJ (extended version). Technical Report abc-2008-1, `http://www.aspectbench.org/`, 05 2008.

[12] E. Bodden, L. J. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In E. Ernst, editor, *ECOOP*, volume 4609 of *Lecture Notes in Computer Science*, pages 525–549. Springer, 2007.

[13] E. Bodden, P. Lam, and L. Hendren. Static analysis techniques for evaluating runtime monitoring properties ahead-of-time. Technical Report abc-2007-6, `http://www.aspectbench.org/`, 11 2007.

[14] G. P. Brat, D. Drusinsky, D. Giannakopoulou, A. Goldberg, K. Havelund, M. R. Lowry, C. S. Pasareanu, A. Venet, W. Visser, and R. Washington. Experimental evaluation of verification and validation tools on martian rover software. *Formal Methods in System Design*, 25(2-3):167–198, 2004.

[15] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *Adaptable and Extensible Component Systems*, Grenoble, France, November 2002. `http://asm.objectweb.org`.

[16] F. Chen, T. F. Şerbănuţă, and G. Roşu. jPredictor: A predictive runtime analysis tool for Java. In *International Conference on Software Engineering (ICSE'08)*. ACM press, 2008. To appear.

[17] S. Cohen. Jtrek. Compaq. No longer maintained.

[18] M. Dahm. BCEL. `http://jakarta.apache.org/bcel`.

[19] M. d'Amorim and K. Havelund. Event-based runtime verification of Java programs. In *WODA '05: Proceedings of the third international workshop on Dynamic analysis*, pages 1–7, New York, NY, USA, 2005. ACM Press.

[20] M. Eichberg, M. Mezini, and K. Ostermann. Pointcuts as functional queries. In W.-N. Chin, editor, *APLAS*, volume 3302 of *Lecture Notes in Computer Science*, pages 366–381. Springer, 2004.

[21] A. Eustace and A. Srivastava. ATOM: a flexible interface for building high performance program analysis tools. In *Technical Conference Proceedings on USENIX 1995*, pages 25–25, Berkeley, CA, USA, 1995. USENIX Association.

[22] Y. Eytani, K. Havelund, S. D. Stoller, and S. Ur. Towards a framework and a benchmark for testing tools for multi-threaded programs: Research articles. *Concurrency and Computation: Practice and Experience*, 19(3):267–279, 2007.

[23] C. Flanagan and S. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. *SIGPLAN Notices*, 39(1):256–267, 2004.

[24] B. Goetz. *Java concurrency in practice*. Addison Wesley, 2006.

[25] A. Goldberg and K. Havelund. Instrumentation of Java bytecode for runtime analysis. In *Fifth ECOOP Workshop on Formal Techniques for Java-like Programs (FTfJP'03)*, July 2003. Darmstadt, Germany.

[26] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java(TM) Language Specification*. Addison-Wesley Professional, 2005.

[27] R. L. Halpert, C. J. F. Pickett, and C. Verbrugge. Component-based lock allocation. In *PACT'07: Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques*, pages 353–364, Sept. 2007.

[28] J. Harrow. Runtime checking of multithreaded applications with visual threads. In K. Havelund, J. Penix, and W. Visser, editors, *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 331–342. Springer, 2000. `http://h30097.www3.hp.com/dtk/visualthreads_ov.html`.

[29] K. Havelund. Using runtime analysis to guide model checking of Java programs. In *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 245–264. Springer, 2000.

[30] K. Havelund and G. Rosu. An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design*, 24(2):189–215, 2004.

[31] A. Kinneer, M. B. Dwyer, and G. Rothermel. Sofya: Supporting rapid development of dynamic program analyses for java. In *ICSE COMPANION '07: Companion to the proceedings of the 29th International Conference on Software Engineering*, pages 51–52, Washington, DC, USA, 2007. IEEE Computer Society.

[32] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[33] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In G. Hedin, editor, *Compiler Construction, 12th International Conference*, volume 2622 of *LNCS*, pages 153–169, Warsaw, Poland, April 2003. Springer.

[34] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In R. N. Horspool, editor, *CC*, volume 2304 of *Lecture Notes in Computer Science*, pages 213–228. Springer, 2002.

[35] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*, pages 167–178, 2003.

[36] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[37] Soot website. `http://www.sable.mcgill.ca/soot/`.

[38] V. Stolz and E. Bodden. Temporal assertions using AspectJ. *Electr. Notes in Theor. Computer Science*, 144(4):109–124, 2006.

[39] S. Ur, E. Bin, and Y. Wolfsthal, editors. *Hardware and Software Verification and Testing, First International Haifa Verification Conference, Haifa, Israel, November 13-16, 2005*, volume 3875 of *Lecture Notes in Computer Science*. Springer, 2006.

[40] Valgrind. `http://valgrind.org`.

[41] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. In *15th IEEE International Conference on Automated Software Engineering*, volume 10, pages 203–232, 2003.

[42] C. von Praun and T. R. Gross. Object race detection. In *OOPSLA*, pages 70–82, 2001.

[43] L. Wang and S. D. Stoller. Run-time analysis for atomicity. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.