Aspect-Oriented Race Detection in Java

Eric Bodden and Klaus Havelund

Abstract—In the past, researchers have developed specialized programs to aid programmers in 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 among 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 describe a new algorithm which we call RACER, an adaption 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 and two smaller programs. Our experiments demonstrate that our implementation is effective in finding subtle data races. In the Rover Executive, RACER finds 12 data races, with no false warnings. Only one of these races was previously known.

Index Terms—Race detection, runtime verification, aspect-oriented programming, semantic pointcuts, static analysis.

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 [1], [2], [3], [4], [5], [6], [7], [8], [9], [10].

All of 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 the program source, while dynamic approaches analyze a trace or abstract-state representation generated by executing the program. Up to now, most existing dynamic approaches have used some form of low-level bytecode instrumentation library to transform the analyzed program into one that generates those events. However, such libraries, for example, BCEL [11], are difficult to use and distract efforts from focusing on the more interesting algorithmic aspects of the analyses.

Researchers have recognized aspect-oriented programming as a convenient tool to declare instrumentation at a high level of abstraction [12], [13], [14], [15], [16]. Aspect-oriented programming allows programmers to use predicates, called pointcuts, to intercept certain events of interest at runtime. Unfortunately, in all of the current Java-based aspect-oriented programming languages, programmers can only intercept events such as method calls, field accesses, and exception handling. In particular, none of these languages allows programmers to intercept events that regard the acquisition and release of locks. This precludes programmers from implementing algorithms in AspectJ that are meant to find concurrency-related programming errors such as data races. In this work, we hence propose a novel extension to the aspect-oriented programming language AspectJ. The language extension that 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 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 overapproximation using a static thread-local-objects analysis [17]. The approximation assures that the pointcut matches every access to a field that is indeed shared. Because of the overapproximation, the pointcut may, however, also match accesses to fields that are not actually shared, i.e., fields that only a single thread accesses.

Using these three novel pointcuts, programmers can easily implement bug-finding algorithms that detect errors related to concurrency. The **lock**() and **unlock**() pointcuts allow a programmer to uniformly act on any acquisition and release

- *E. Bodden is with the Software Technology Group, Technical University Darmstadt, Hochschulstr. 10, S2|02 A209, 64289 Darmstadt, Germany. E-mail: eric@bodden.de.*
- *K. Havelund is with the Jet Propulsion Laboratory, California Institute of Technology, M/S 301-285, 4800 Oak Grove Dr., Pasadena, CA 91109. E-mail: klaus.havelund@jpl.nasa.gov.*

Manuscript received 30 Dec. 2008; revised 12 May 2009; accepted 13 July 2009; published online 27 Jan. 2010.
Recommended for acceptance by B. Ryder and A. Zeller.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSESI-2008-12-0412.
Digital Object Identifier no. 10.1109/TSE.2010.25.

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 [18]. To show how programmers can use this language extension, we adapted the ERASER race-detection algorithm [19] to Java, and implemented it using the new pointcuts. The new algorithm is named RACER. Both ERASER and RACER detect program executions which reveal potential for data races in the executed application.

We presented [20] a first version of the RACER algorithm at ISSTA 2008. However, we subsequently noted[1] that a large number of potential data races that this version of RACER reported were unfortunately false warnings. The initial version of RACER reported these false warnings because it ignored calls to `Thread.start()`. The improved version of RACER that we present in this paper takes such calls into account and therefore avoids reporting these false positives.

We applied the aspects implementing the RACER algorithm to a plan execution program for the NASA K9 rover and two other multithreaded programs written by computer science researchers. Our results show that the algorithm is effective in finding data races. In the NASA code, RACER found 12 races, 11 of which were previously unknown, although extensive studies had been performed on the K9 rover code before. In the other two programs, we found no races, which was expected because we strongly believe that these programs are race-free. RACER reported only one false warning on these three benchmarks.

As we will show, beyond data race detection, the extension is able to support most (if not all) other concurrency analysis algorithms, which typically analyze properties of synchronization and field accesses. Our extension of AspectJ can capture all of Java's synchronization primitives, and AspectJ already provides pointcuts for accessing those entities intended to be protected by synchronization, namely, field reads and writes. The extension, for example, will be able to support algorithms for deadlock detection [21], high-level data race detection [6], and stale-value detection [7]. Researchers have previously implemented all of these algorithms using the low-level BCEL bytecode instrumentation library [11]. Programmers could implement such bug-detection tools much easier using AspectJ. 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 the case of the **maybeShared**() pointcut through a static whole-program analysis,
- an algorithm for race detection in Java, coined RACER, that improves on ERASER, and an implementation using the three novel AspectJ pointcuts, and
- an experiment showing that our implementation is effective in finding data races in a plan execution program for the NASA K9 rover.

1. We wish to thank Bill Pugh (University of Maryland) for pointing out these false positives at the ISSTA conference.

```
1 class Task implements Runnable {
2
3   static int shared;
4   static int shared_protected;
5   int not_shared;
6
7   public void run() {
8     System.out.println(shared++);
9     synchronized(Task.class) {
10      System.out.println(
11        shared_protected++);
12    }
13    System.out.println(not_shared++);
14  }
15
16  public static void main(String[] args) {
17    Task task1 = new Task();
18    Task task2 = new Task();
19    Thread thread1 = new Thread(task1);
20    Thread thread2 = new Thread(task2);
21    thread1.start();
22    thread2.start();
23  }
24}
```

Fig. 1. Example program containing a data race.

2 Example Program

In Fig. 1, we show an example program that contains 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 that the other thread accesses. 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 Fig. 1) with a bytecode instrumentation package such as BCEL [11] to emit these events at runtime. A special runtime environment would then monitor the events and report a programming error as the error is detected. Programming the bytecode instrumentation packages is a tedious and time-consuming task. We propose using aspect-oriented programming instead.

```
1 aspect Racer {
2   pointcut scope():
3     !within(ca.mcgill.sable.racer.*);
4
5   pointcut fieldSet(Object owner):
6     set(!static * *) && target(owner);
7
8   ...
9   before(Object owner): fieldSet(owner)
10    && scope() {
11    ...
12  }
13  ...
14}
```

Fig. 2. Example pointcut and advice.

3 AspectJ Language Extension

Aspect-oriented programming is a programming style that allows programmers to implement special "cross-cutting concerns" in a modular way and then combine these concerns with a base program through a process called weaving.

One particularly popular aspect-oriented programming language is AspectJ [22]. AspectJ is a backward compatible language extension to Java. It allows programmers to define a set of aspects, where an aspect itself is similar to a normal Java class. However, unlike the case for normal Java classes, programmers do not usually invoke the methods of an aspect explicitly. Instead, the AspectJ runtime invokes these methods (programmers frequently call the methods "pieces of advice" or just "advice") implicitly, at a set of well-defined points in the program's execution, so-called joinpoints.

In AspectJ, programmers can define a piece of advice to execute at one of the following program events (see [23] for details):

- method call,
- method execution,
- constructor call,
- constructor execution,
- static-initializer execution,
- object preinitialization,
- field read,
- field write,
- exception handler execution,
- advice execution.

Programmers can use special predicates, called pointcuts, to define the set of joinpoints at which each piece of advice should execute. Fig. 2 shows part of an aspect that we use in our own implementation of the RACER algorithm. In lines 2-3, the aspect defines a pointcut `scope`. This pointcut selects from all joinpoints that we mentioned in the list above those that do not occur within the lexical scope of any class within the package `ca.mcgill.sable.racer`. Programmers frequently use such pointcuts to prevent their aspects from applying to themselves. The pointcut `fieldSet` in lines 5-6 matches any joinpoint at which the program assigns a value to a nonstatic field of any type and name (as denoted by the wildcard *). At the same time, the pointcut binds the variable `owner` to the target object, i.e., the object whose field it is assigned to. The advice in lines 9-12 declares that it executes `before` any joinpoint described by the pointcut `fieldSet` and by the pointcut `scope`, i.e., before any assignment to a nonstatic field that happens within a method body that is outside of the lexical scope of any class in the package `ca.mcgill.sable.racer`.

```
private final ReentrantLock lock =
    new ReentrantLock();

public void m() {
  lock.lock();  // block until condition holds
  try {
    // ... method body
  } finally {
    lock.unlock()
  }
}
```

Fig. 3. Use of class `ReentrantLock`.

Researchers have identified [12], [13], [14], [15], [16] long ago that runtime monitoring for bug detection is a crosscutting concern, and aspects resemble a convenient abstraction for implementing runtime monitors. 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 in Java programs. This is because, traditionally, AspectJ did not allow a programmer to detect lock and unlock events caused by synchronized regions 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 among 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 nowadays has multiple ways to implement a locking policy. For example, with Java 5, Sun introduced the new library `java.util.concurrent`, which offers classes like `ReentrantLock`. Fig. 3 shows a code stub taken from Sun's documentation of this class. As one can see, programmers acquire locks explicitly using this class, by calling the `lock()` method. They can release a lock by calling `unlock()`.

In Fig. 4, we show standard AspectJ pointcuts that programmers can use to capture these events. The pointcut definition in lines 1-2 matches all calls to the `lock()` method and binds the target object of the call to variable `l`. The pointcut definition in lines 3-4 does the same for `unlock()`. Researchers can easily construct bug-detection

```
1 pointcut lock(ReentrantLock l):
2   call(* ReentrantLock.lock()) && target(l);
3 pointcut unlock(ReentrantLock l):
4   call(* ReentrantLock.unlock()) && target(l);
```

Fig. 4. Pointcuts matching on calls to `ReentrantLock`.

algorithms using such pointcuts if the concurrent program under test only uses the class `ReentrantLock` for locking.

In general, another locking style is, however, much more pervasive: the use of `synchronized` blocks and methods. As we showed in lines 9-12 of Fig. 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 Fig. 1 on `Task.class`). The virtual machine automatically releases the lock when control leaves the block (either by throwing an exception or by normal flow). This way, lock and unlock operations are balanced at runtime.

For convenience, programmers can also flag methods with the `synchronized` modifier. A method declaration

synchronized void foo() { /* *code* */ }

is semantically equivalent to the declaration

void foo(){ **synchronized(this)** { /* *code* */ } }

and a declaration

static synchronized void foo() { /* *code* */ }

within a class `C` is semantically equivalent to:

static void foo() {
  synchronized(C.class) { /* *code* */ } }

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.

3.1.1 Syntax and Semantics

The programmer can use both pointcuts directly within any pointcut declaration. 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 access the object that is locked, respectively, unlocked, by conjoining the `lock()`, respectively, `unlock()` pointcut with an `args(..)` pointcut. Further, the programmer can attach pieces of advice to these pointcuts. An advice then executes whenever its pointcut matches. Fig. 5 shows two example advices 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 it 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

```
1 before(TaskQueue l): lock() && args(l) {
2   System.out.println(
3     "About to acquire a lock on "+l);
4 }
5
6 after(TaskQueue l): lock() && args(l) {
7   System.out.println(
8     "Successfully acquired a lock on "+l);
9 }
```

Fig. 5. Logging lock acquisition with our AspectJ language extension.

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 supertype of all reference types.

3.1.2 Implementation for Synchronized Blocks

For synchronized blocks, the implementation of `lock()` and `unlock()` pointcuts is relatively straightforward, by the way that a Java compiler generates bytecode for synchronized blocks. The compiler translates a synchronized block **synchronized**(x) {/* *code* */} to Java bytecode of the following form:

1: monitorenter(x);
2: /* *code* */
...
n: monitorexit(x);
...
m: monitorexit(x);

trap Throwable from 1 to n with m

In other words, `monitorenter` and `monitorexit` bytecodes surround the protected region. A virtual machine trap handles the case where the protected region throws an exception. On any exception (`Throwable` is the common ancestor of any exception type in Java) occurring between lines 1 and n, the trap jumps to line m, where the program then releases the lock on `x` by executing `monitorexit(x)`.

We implemented `lock()` and `unlock()` pointcuts using the AspectBench Compiler (*abc*) [18]. *abc* performs the weaving process on an internal three-address-code representation. This representation exposes `monitorenter` and `monitorexit` bytecodes as shown above. We therefore implemented the `lock()` pointcut by matching on `monitorenter` and the `unlock()` pointcut by matching on `monitorexit`, respectively. The programmer can conjoin any of the two pointcuts with an `args(x)` pointcut. The compiler then extracts the locked object from the `monitorenter` or `monitorexit` bytecode and binds this object to `x`.

3.1.3 Implementation for Synchronized Methods

The approach for synchronized methods is not quite as straightforward. The crucial question to answer is where instrumentation code within synchronized methods should be woven. As an example consider the first advice definition in Fig. 5 (lines 1-4). The programmer stated that lines 2-3 are to be executed *before* the lock is acquired. Assume that our implementation simply wove this advice by inserting lines 2-3 at the beginning of the declaration of each synchronized

method. This would give us the wrong semantics. According to the Java language specification [24], the method body is executed *after* the lock has been acquired.

To work around this problem, we transform synchronized methods to unsynchronized methods holding synchronized blocks, as we showed in Section 3.1. We implemented this transformation in the AspectBench Compiler. After the compiler has applied the transformation, we know that synchronization can only occur through synchronized blocks, not methods, and we can, therefore, apply the weaving strategy from Section 3.1.2.

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., accesses to fields which more than one thread could potentially write to. 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 multiple threads *do* write to or read from this field (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 overapproximating 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 that use `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 (Fig. 1), we would like to match the field accesses in lines 8 and 11 but not 13, because the field in line 13 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 describe 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.

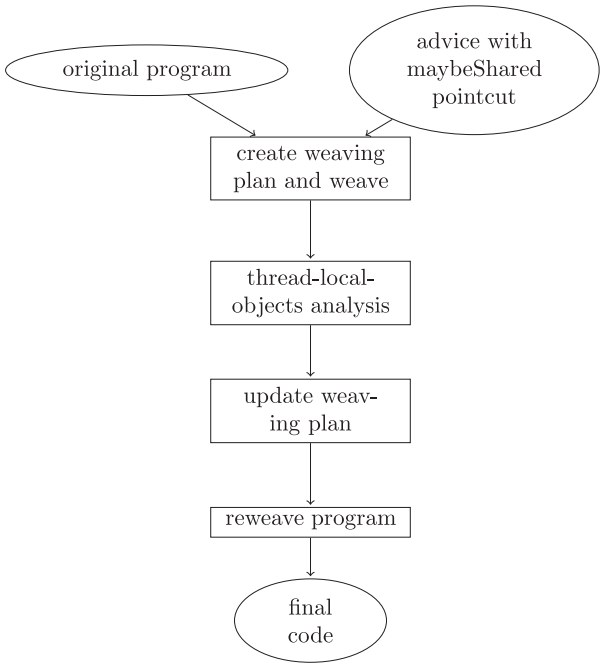

Fig. 6. Weaving process for maybeShared pointcut.

3.2.1 Implementation Overview

Our implementation of `maybeShared()` uses a compiler feature called reweaving. Fig. 6 explains this feature. 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.[2] 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 unweave the woven program to restore its original code, and then reweave the program using the optimized weaving plan. As a result, `maybeShared()` does not match any field access where the thread-local-objects analysis was precise enough to prove the accessed field thread-local.

3.2.2 Thread-Local-Objects Analysis

We use a thread-local-objects analysis that Halpert et al. originally developed for component-based lock allocation. In their paper [17], the authors describe the approach in detail (Section 3 there). We here only outline the analysis process.

2. In our particular setting, it would be equally possible to apply the thread-local-objects analysis to the unwoven Java program. However, by applying it to the woven program instead, we allow programmers to apply their bug-detection algorithms not only to Java programs but to AspectJ programs as well: After weaving, the compiler has reduced the AspectJ program to a plain Java program, which our analysis can handle.

The thread-local-objects analysis is a flow-insensitive context-sensitive whole-program analysis, but unlike many other context-sensitive static analyses, it considers thread-creation sites as context, not method-call sites. 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 [25] 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, the analysis creates 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 (TLOs) 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 $methods(t)$ that may be executed through t. Then, the analysis flags as thread-shared every field accessed by a method not in $methods(t)$. The analysis classifies all other fields as thread-local. A similar classification applies to method parameters. If a method outside t calls a method m, then the analysis considers m's parameters as thread-shared; otherwise, it considers the parameters as thread-local.

In a next step, the thread-local-objects analysis uses 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.

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

The thread-local-objects analysis is demand-driven, i.e., it computes for every field f separately whether or not f may be shared. Hence, we can decrease the analysis time by asking the analysis for information only about exactly those fields f for which this piece of information (whether or not f may be shared) actually matters. We ask the analysis for information about 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 Fig. 1, then we only query the thread-local-objects analysis for the fields `shared` and `shared_protected` because `not_ shared` is nonstatic, 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. [19]. RACER aims at detecting potential data races at runtime, just as ERASER does. Therefore, RACER has some parts in common with the ERASER algorithm. However, RACER's semantics is closer to Java's memory model [26] and, as we will see, RACER can, therefore, detect data races in Java programs that ERASER would miss.

When we presented RACER at the 2008 International Symposium on Software Testing and Analysis (ISSTA), Bill Pugh, one of the creators of the current version of the Java Memory Model [26], pointed out to us that some of the race warnings that RACER issued were actually false warnings. The initial version of RACER caused these false positives because it ignored calls to `Thread.start()`. According to the Java Memory Model, such calls create a happens-before edge, and therefore, allow for a safe handover of values from the caller thread to the started thread. For this paper, we therefore created an enhanced version of RACER, which takes calls to `Thread.start()` into account in order to avoid false positives.

We next explain the basic principles underlying both the ERASER and our RACER algorithm, and then explain how RACER differs from ERASER.

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 $L(o.f)$. We maintain the set using a chain of maps. A map `ownerToFieldToLocks` associates an owner o with a map `fieldToLocks`, which then maps from f to $L(o.f)$. For the map `ownerToFieldToLocks`, we use a weak identity hash map. Such a map compares keys on object-reference identity (as opposed to equality). Further, weak maps automatically dispose of entries whose key collected garbage. This practice prevents our implementation from causing memory leaks. Weak maps register with a special reference queue that the garbage collector maintains. When collecting an object o, the garbage collector notifies the map, and the map, in turn, discards its mapping for o (if any such mapping is present). The same holds for class objects C. However, the garbage collector does not collect a class C before it can also collect C's class loader. This, in turn, is commonly the case only when the virtual machine shuts down. Therefore, using weak hash maps will normally not save any memory for mappings for class objects C. Note that this form of memory management is sound. After the garbage collector has collected object o (or class C), no thread can access its fields

```
1 public aspect Locking {
2
3   ThreadLocal locksHeld = new ThreadLocal() {
4     protected synchronized
5       Object initialValue() {
6       return new HashBag();
7     } };
8
9   before(Object l): lock() &&
10   args(l) && Racer.scope() {
11    Bag locks = (Bag)locksHeld.get();
12    locks.add(l);
13  }
14
15  after(Object l): unlock() &&
16   args(l) && Racer.scope() {
17    Bag locks = (Bag)locksHeld.get();
18    assert locks.contains(l);
19    locks.remove(l);
20  }
21}
```

Fig. 7. Aspect bookkeeping thread local lock sets.

any more. Therefore, no field of o (or C) can be part of a race on the remainder of the execution.

As the program under test starts up, we assume that lock sets hold the locks of all possible objects. As there is no way to enumerate all of 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 shown in Fig. 7. Because Java's locks are reentrant, we use a bag instead of a set. In lines 3-7, 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 9-13). Whenever the program releases a lock `l`, we remove it from the bag (lines 15-20). (The conjunction "`&& Racer.scope()`" prevents the pointcuts from matching within our own RACER implementation, and therefore avoids potentially 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.

An additional advantage of using AspectJ is that we could easily modify the `Locking` aspect to take other locking styles into account. For instance, if `ReentrantLocks` were used (see Section 3), we could just extend the pointcuts in Fig. 7 with an additional disjunct, e.g., replacing lines 9 and 10 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 acquired.

Whenever the program under test accesses a field, the AspectJ runtime notifies a second aspect `Racer`. We show the essential parts of this aspect in Fig. 8.

```
1 aspect Racer {
2
3   pointcut staticFieldSet():
4     set(static * *) && maybeShared();
5   pointcut fieldSet(Object owner):
6     set(!static * *) && target(owner)
7       && maybeShared();
8   pointcut staticFieldGet():
9     get(static * *) && maybeShared();
10  pointcut fieldGet(Object owner):
11    get(!static * *) && target(owner)
12      && maybeShared();
13
14  before(): staticFieldSet() && scope() {
15    String id =
16      getId(thisJoinPointStaticPart);
17    Class owner =
18      getClass(thisJoinPointStaticPart);
19    SourceLocation loc =
20      getLocation(thisJoinPointStaticPart);
21    fieldSet(owner,id,loc);
22  }
23
24  before(Object owner): fieldSet(owner)
25    && scope() {
26    String id =
27      getId(thisJoinPointStaticPart);
28    SourceLocation loc =
29      getLocation(thisJoinPointStaticPart);
30    fieldSet(owner,id,loc);
31  }
32  ...
33}
```

Fig. 8. Aspect updating per-field lock sets on field access.

The aspect first declares four different pointcuts (lines 3-12) that match writes to static fields and instance fields (the first two pointcuts) and reads from static fields and instance fields (the last two pointcuts). Note that we use the `maybeShared()` pointcut because we are not interested in accesses to fields that cannot possibly be shared among threads.

Two pieces of advice follow. The first, in lines 14-22, 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. Note how convenient it is to get access to an event's source location using `thisJoinPointStaticPart`. We believe that this particular AspectJ feature can be very helpful for may online bug-detection algorithms.

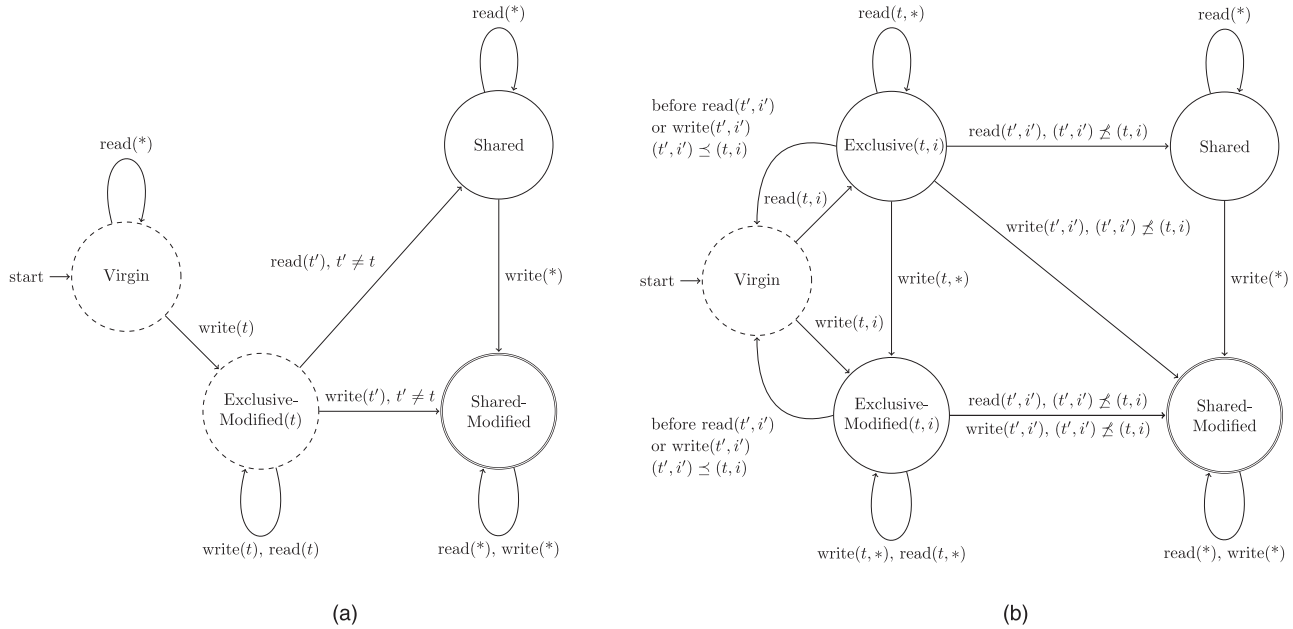

Fig. 9. ERASER and RACER state machines: Dashed states do no lock refinement; double-lined states report race potential when the lock set becomes empty. (a) ERASER state machine. (b) RACER state machine.

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. Our RACER implementation could, however, have data races within its own code. We therefore used stratified aspects [27] to validate that there were no races in our implementation. Stratified aspects allow programmers to declare aspects on a metalevel. Aspects on a metalevel cannot apply to aspects on the same level (and hence, no to themselves), but they can apply to aspects on a lower level. To check for races within RACER, we declared two copies of RACER: 1) one copy on the metalevel 1 (directly above the base code) and 2) one copy on the metalevel 2, directly above level 1. We then ran the resulting woven program using our benchmark set (see Section 5). In this setting, the aspects "(1)" would report data races in metalevel 0 (the base program), while the aspects "(2)" would report data races in metalevel 1, which includes the aspects "(1)." In all cases, both copies reported the same races, i.e., the aspects "(2)" reported no races within our RACER implementation.

To conclude our description in Fig. 8, the advice then calls 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 $L_T(t)$:

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

The variable is regarded as safely protected as long as $L(C.f)$ never goes empty. This is because, if the programmer uses a lock consistently to protect the field $C.f$, this lock will remain in $L(C.f)$ during all refinements. The potential for a data race may, on the other hand, exists if $L(C.f)$ becomes empty at some point (and $C.f$ is shared among threads).

The second piece of advice in lines 24-31 of Fig. 8 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).

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 [19]. 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 Fig. 9a around the lock-set refinement algorithm from 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, the variable's state changes to Exclusive-Modified(t).[3] This signifies that t is initializing the variable, and thus only t should access the variable, while it is in this state. While in this state, the variable is considered in 3) its initialization phase—t may write to and read from the variable. RACER does not refine lock sets before another thread t' accesses the variable, entering the state Shared, respectively, SharedModified, an indication that property 1), single-threaded access, does not

3. This state was called Exclusive in [19] but the name Exclusive-Modified(t) suits our comparison to RACER better.

```
class ThisEscape {
  int i;

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

Fig. 10. Constructor letting an implicit reference to `this` escape to an event-processing thread; c.f. [28, page 41].

```
1 class Thread1 extends Thread {
2   int var;
3
4   Thread1(int i){ var = i; }
5
6   public static void main(String[] args) {
7     Thread1 t1 = new Thread1(42);
8     t1.start();
9     //t1.var = 23;
10  }
11
12  public void run() {
13    System.out.println(var);
14  }
15}
```

Fig. 11. Example code starting a new thread.

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.

4.3 Safe versus Unsafe Initialization in Java

As noted above, ERASER grants an explicit initialization phase for each variable—a phase which is assumed safe. This may cause Eraser to miss data races when they occur during an object's initialization phase. The same holds in a Java-based setting. In Fig. 10, we give a subtle example of unsafe unsynchronized field accesses even during object initialization. We adapted the example from Goetz's book on Java concurrency [28]. Assume a system in which different threads, for instance, the main thread, add events to an event queue. Also, the main thread starts an event-processing thread at the beginning of the program, which removes these events from the queue and sends the events to registered event listeners. This causes these listeners to process the event in this event-processing thread. The constructor in Fig. 10 creates an object of an anonymous inner class which it then registers as an event listener. The problem is that the event-processing thread may execute the `doSomething()` method as soon as this registration has happened. The method has access to its parent `ThisEscape` object, via an implicit `this` reference. The event-processing thread runs concurrently to the constructor of `ThisEscape`, i.e., according to the revised Java Memory Model [26], the constructor execution neither happens before nor happens after the execution of `onEvent(..)`. As a result, the value of 42 for the field `i` is not visible to this listener, and if the listener were to access the field `i`, it would not be clear which value it would read, 0 or 42. We have just witnessed a very subtle data race. Mature listener-based frameworks like Sun's Abstract Window Toolkit (AWT) therefore safeguard against such races through additional synchronization code. We believe that such subtle races are particularly hard to find, and therefore, our tool should report such races just as any other race. Because many tools for Java imitate C-based algorithms like ERASER in some Java programs such races went undetected for years. The ERASER algorithm would miss the data race in Fig. 10 because the write to `i` followed by the read from `i` lead to the state Shared. ERASER does not report a race in this state even when the lock set becomes empty.

For RACER, although 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; we would, following the Java Memory Model, want to weaken property 3): RACER only assumes a variable's initialization as safe in one certain special situation.

We show an example of this "safe" situation in Fig. 11. The main thread assigns the value 42 to variable `t1.var`, and then starts the thread `t1`. From there onward, only `t1` accesses the variable `var`. In this case, `t1` may access `var` without locking because the Java Memory Model assures that the initialization in the main thread happens before any potential access by `t1`: According to Section 17.4.4 of the Java Language specification [24], it holds that "[an] action that starts a thread synchronizes with the first action in the thread it starts."[4] Fig. 12 shows a sequence diagram for this program.

Note how this situation is different from the one that is shown in Fig. 13. This figure is based on a modified version of the code in Fig. 11 in which the assignment that is commented out at line 9 gets executed. In this case, there is still no race between the first assignment by the Main thread and the read in Thread1, but there is a race between the read in Thread1 and the second assignment in the main thread. This is because there is not a happens-before edge between any code of the main thread that follows the spawn of Thread1 and the code of Thread1 itself. In order to make the assignment in line 9 visible to Thread1, both threads would have to agree on a common lock which they would then have to synchronize on.

When we compare the situation in Fig. 12 with the one in Fig. 13, it becomes clear that while the updates performed by the Main thread that happened in the period marked as (Main, 0) are visible to Thread1, any accesses in the period marked as (Main, 1) are not visible to Thread1. This is because of a data race, as indicated by a missing happens-before relationship. To make visible to Thread1 all writes performed by the Main thread that happened after spawning Thread1, the programmer must extend Thread1 to use a locking discipline that is consistent with the locking

4. Our initial RACER implementation presented at ISSTA did not treat these kinds of situations precisely.

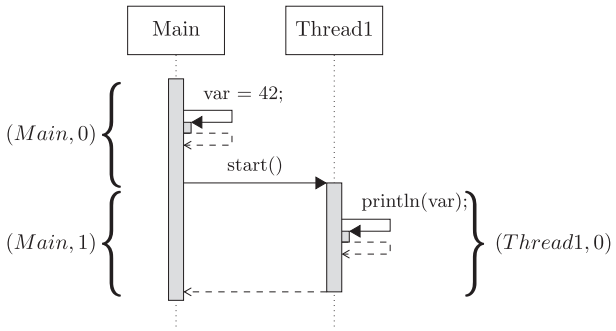

Fig. 12. Access periods in code of Fig. 11.

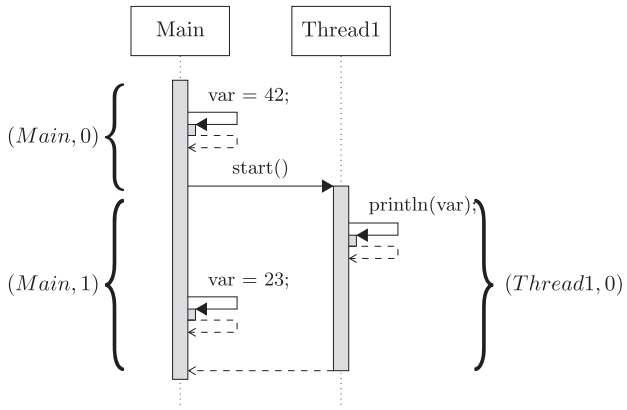

Fig. 13. Access periods in case with data race.

discipline that the Main thread uses. In a nutshell, we can therefore conclude that in Java, a thread t may access an object without locking only if it initialized this object itself, or if a parent thread initialized the object *prior* to starting t.

In order to decide whether an object's initialization took place prior to starting the child thread or not, RACER divides each thread's lifetime into multiple *access periods*, just as we see in Figs. 12 and 13.

4.4 Access Periods

When a thread t spawns a thread t', then t' can safely access all values that t wrote *before* the event of spawning t' (without holding a lock); however, t' may not access such values that t wrote after spawning t' (without holding a lock). RACER hence has to have a means to distinguish writes before a call to `Thread.start()` from writes after such a call.

RACER therefore associates with every field f in its state machine not just the last Thread t that accessed f, but instead an *access period* (t, i_t), where $i_t \in \mathbb{N}$ identifies a period in time. This period is nothing other than a thread-local counter. When a thread t is started, its counter i_t has value 0, and we say that "t is in period 0." We increment i_t every time t spawns another thread. Whenever t accesses a field f, then we associate with f the access period (t, i_t). This then allows us to identify later on whether this access happened before the spawning of a particular thread or after. To keep track of which access periods are visible to other access periods, RACER needs to keep track of spawned threads and *when* they are spawned.

4.5 Keeping Track of Spawned Threads

In general, RACER needs to keep track of which thread t spawned a thread t' and during which period. Let $p = (t, i_t)$ be an access period which is terminated by t spawning a thread t'. As this call to `t'.start()` occurs, RACER stores a *spawning relationship* between p and the period $(t', 0)$, effectively expressing the information that thread t' can (in period 0 and onward) safely access fields that were assigned by thread t in its period i_t or earlier. Based on this information, we next define a visibility relation that expresses which values a spawned thread t' can safely access from its parent thread t.

4.5.1 Visibility Relation

The visibility relation $\prec$ is the smallest relation among access periods such that for all periods (t, i_t), $(t', i_{t'})$, and $(t'', i_{t''})$, and for all $i, j \in \mathbb{N}$, it holds that

$$t \text{ spawned } t' \text{ in period } i_t \rightarrow (t', 0) \prec (t, i_t),$$
$$j > i \rightarrow (t, j) \prec (t, i).$$

We define $\preceq$ to be the reflexive and transitive closure of $\prec$. Based on this relationship, a thread t' may safely access a value v in period $i_{t'}$ without locking if v is associated with a last access period (t, i_t) for which $(t', i_{t'}) \preceq (t, i_t)$ holds.

Note that this definition of the visibility relation only takes into account the spawning of threads. In particular, it ignores other Java synchronization primitives, such as calls to `Thread.join()` and accesses to volatile fields, or uses of specialized atomic classes as they exist in the package `java.util.concurrent.atomic`. Generally, this can lead to false positives when analyzing a program with RACER because RACER may miss happens-before relationships that these other synchronization primitives establish. Researchers could extend RACER to handle such primitives as well. For instance, Harrow [1] handles the C-equivalent to a call to `t.join()` in Java by transferring `t`'s lock set to the thread that performs the `join()` call. For now, we decided not to implement support for such additional primitives because the false-positive rate that we observed for RACER was already low enough on our benchmarks and would not have improved for this benchmark set even if we had implemented such support. Also, note that calls to `Object.wait()` and `Object.notify()` are not additional synchronization primitives: A thread may only call either of these methods on an object o when this thread already owns the lock of o. Hence, the lock refinement that RACER implements handles such cases implicitly and precisely.

Fig. 9b shows our state machine for the RACER algorithm. As can be seen, the right-hand side of this state machine is equivalent to the one that ERASER uses (Fig. 9a). However, RACER offers a more fine-grained treatment of initialization. In particular, ERASER only contains a single ExclusiveModified state: Initial read operations on a variable let this variable reside in the Virgin state. RACER, on the other hand, distinguishes reads and writes right from the beginning. This difference will prove crucial in the example that follows below.

A second difference is that ERASER parameterizes the ExclusiveModified state with a thread t only. In RACER, on the other hand, the Exclusive and ExclusiveModified states are parameterized with an access period. This additional information lets RACER compute different outgoing

transitions for these states, based on whether or not the previous read or write that lead into this state is visible to the current read or write.

Let us consider both cases, starting with the case where the read or write that leads into the Exclusive(Modified) state is invisible to the currently observed read or write, i.e., where $(t', i') \not\preceq (t, i)$ holds. In this case, we transition just as in ERASER: A write will lead to SharedModified and a read will lead to Shared. If we reach SharedModified due to a write and the refined lock set happens to be empty, a race is reported between this write and the preceding operation on this variable.

In the second case, the read or write that leads into the Exclusive(Modified) state is visible to the observed read or write, i.e., where $(t', i') \preceq (t, i)$ holds. In this case, thread t' is allowed to access the variable without a lock. Moreover, RACER assumes that the programmer meant to hand over the variable's value to t': Just before processing the read or write by t', RACER resets the variable's state to Virgin. This reassigns the full lock set to the variable. Then, RACER immediately and atomically processes the read or write, leading back into either Exclusive or ExclusiveModified (depending on whether a read or write was observed), but this time parameterized with the new access period (t', i'). From this point on, t' "owns" the variable. Another thread t'' can only access its value if it is started by t' (which again allows a safe handover to t'') or if it uses a locking discipline that is consistent with the one that t' uses.

4.6 Detailed Example

As an example, consider again the sequence diagram in Fig. 13. As noted earlier, this example does contain a data race. For this sequence of events, RACER will perform the following transitions for the field `var`:

1. On the write `var` = 42, the state will switch from Virgin to ExclusiveModified(Main,0). The lock set for `var` will be refined from the full set to the empty set because Main holds no lock during the write.
2. On spawning Thread1 from Main, RACER will record the visibility relationship between (Thread1,0) (and following) and (Main,0). Just afterward, it will then increment the access period for thread Main from (Main,0) to (Main,1).
3. Just before the read in `println(var)`, the implementation will switch from state ExclusiveModified(Main,0) to Virgin, because the read occurs in period (Thread1,0) and (Thread1,0) $\preceq$ (Main,0) holds. RACER resets the lock set for `var` to the full set. Then, instantly (and atomically), the implementation will process the read and switch to a new state Exclusive(Thread1,0). RACER refines the lock set during this process, yielding the empty set again. The value `var` was effectively handed over from thread Main to thread Thread1.
4. On the write `var` = 23, we move from Exclusive (Thread1,0) to SharedModified because the write occurs in period (Main,1) and (Main,1) $\not\preceq$ (Thread1,0) holds. When we reach the state SharedModified, we instantly report a race because the lock set is empty.

Note that resetting a value to its Virgin state also allows us to report precisely the right line number information for data races. In the above example, we only report the accesses (and their line numbers) that occurred *after* the Virgin state has been left again: Moving back to Virgin not only resets the lock sets but also the access information. In particular, we do not report the original write of value 42 as part of the race.

4.7 No Missed Races

It is important to note that unlike the ERASER algorithm, the RACER algorithm cannot miss any races: If a program run causes a data race, then the RACER algorithm will detect this race. Consider again the example from Fig. 10. As we explained, this figure shows a situation in which a data race can arise that ERASER would miss because it would only move to state Shared for this example. The RACER algorithm would instead move to SharedModified for the same example, hence reporting the data race. In general, when a thread t reads a variable v, and a thread t' then writes to v, or the other way around, and the first operation is not visible to the second one (according to the visibility relation), and t and t' do not hold a common lock during these two operations, then RACER will report a race. While the conditions mentioned in the last sentence are not sufficient for proving that a data race has really occurred (e.g., the threads could have synchronized by other means that RACER does not capture), the conditions are necessary for a data race to occur.

4.8 Implementation

The implementation of RACER is essentially as described in Section 4.1. It contains two aspects: `Locking` and `Racer`. While `Locking` keeps track of lock sets (or rather bags), the `Racer` aspect maintains a mapping from fields (qualified by their owning object) to a state and updates the state according to Fig. 9b. The transition logic in Fig. 9b is implemented within two methods, `fieldSet` and `fieldGet`, in the `Racer` aspect (see Fig. 8). In addition, the `Racer` aspect keeps track of invocations of `Thread.start()`, storing a mapping between the started and the starting thread. This allows us to decide the visibility relationship $\preceq$. (One cannot simply store a thread t's parent thread in a thread-local variable of t because the information about the parent thread may need to be accessed by a thread different from t, but only t itself can get access to its own thread-local variables.)

4.9 Implementing Other Algorithms for Finding Concurrency-Related Bugs at Runtime

We are confident that our AspectJ language extension is able to support, beyond algorithms for data race detection, most (if not all) other dynamic concurrency-related analysis algorithms, which typically analyze properties of synchronization and field accesses. In particular, our AspectJ extension can capture all of Java's synchronization primitives. RACER only uses events acquiring and releasing locks, as well as calls to `Thread.start()`. However, researchers can easily use AspectJ to also handle calls to `Thread.isAlive()` or `Thread.join()`, or accesses to volatile variables. The only other cases that introduce happens-before relationships in Java are cases in which a thread t_1 is interrupted and another thread t_2 detects the fact that t_1 was interrupted. Thread t_2 can

detect this fact by checking the results of `Thread.interrupted()` or t_1`.isInterrupted()`, or by catching an `InterruptedException`. Researchers can intercept all three classes of events using plain AspectJ. Further, AspectJ already provides pointcuts for accessing those entities intended to be protected by synchronization, namely, field reads and writes. In fact, the only frequently requested missing AspectJ pointcut that we are aware of is a pointcut that would capture assignments to local variables. It may be useful to monitor such events for bug detection in general. However, local variables play no role in concurrency-related errors, as they are thread-local.

The AspectJ language extension that we present here is therefore able to support algorithms for deadlock detection [21], high-level data race detection [6], and stale-value detection [7]. All of these algorithms have previously been implemented using the low-level BCEL bytecode instrumentation library [11]. Programmers could implement such bug-detection tools more easily using AspectJ.

In particular, we see the following advantages of using AspectJ over instrumenting a program manually, using a bytecode instrumentation package, a debugging interface or similar means. First, the AspectJ code is declarative. This has the advantage that the researcher can easily validate that the instrumented program will expose the right events, just by looking at the appropriate pointcut definitions. Due to a large user base, the available AspectJ compilers have by now become very mature, and we consider it safe to assume that, given a correct pointcut definition, the compiler will produce correctly instrumented code. It is also easy to convert existing bug-finding algorithms to use our AspectJ language extension for instrumentation. The researcher just has to define an aspect that calls the right methods of the bug-finding algorithm when the appropriate events occur at runtime. The implementation of the bug-finding algorithm itself does not need to change. Another advantage is that researchers can easily use advanced features of AspectJ, such as the exposure of context information via the `this`, `target` and `args` pointcuts or the reflective `thisJoinPoint` and `thisJoinPointStaticPart` objects. While one can also expose such information using manual instrumentation, this appears more cumbersome than just using the AspectJ primitives mentioned above.

Another advantage is that users can easily lower the runtime overhead of the bug-finding code by refining pointcuts to scope the instrumentation so that it applies only to certain packages or classes. To minimize the required changes to the aspect code, researchers can provide their implementation as a set of abstract AspectJ aspects. An abstract aspect can have abstract pointcut definitions. The researchers can then define their bug-finding algorithm completely in terms of these abstract pointcuts. Users who wish to apply this algorithm to a program then only need to instantiate a concrete subaspect that defines the abstract pointcuts. For instance, in RACER, we could define the `Racer` aspect as abstract and define the `scope()` pointcut as:

```
abstract pointcut scope();
```

Users of RACER could then restrict the bug-checking to a package`check.me` by simply providing the following concrete aspect:

```
aspect MyRacer extends Racer {
  pointcut scope(): within(check.me.*);
}
```

To the best of our knowledge, it would be hard to reach the same degree of flexibility when using a manual instrumentation approach.

As we showed in this work, even when using AspectJ, one can still have access to sophisticated program analysis techniques, for instance, techniques that determine potentially shared fields. Even better, aspects allow researchers to query and encapsulate such analyses in a declarative way, by providing pointcuts such as `maybeShared()`. The fact that the aspect references the analysis simply by the name of the pointcut decouples the analysis itself from the analysis client (the bug-finding algorithm). Therefore, in the future, researchers could implement improvements to the thread-local-objects analysis that determines `maybeShared()`, and any client code that uses the `maybeShared()` pointcut could automatically take advantage of this improved analysis without having to change any of their program code. Indeed, the idea of exposing analyses to a user in the form of pointcuts seems useful not only with respect to bug-finding algorithms for concurrency-related programming errors but also seems useful in general [29].

5 Case Study

We applied RACER to an experimental planetary rover controller, named the K9 Executive, for a rover named K9 developed at the 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 was 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 multithreaded. In Fig. 14, we show the threads relevant for the presentation in this paper as boxes drawn with full lines. Boxes with dashed lines (the lower part of the figure) represent objects that these threads access. For every object, we show in an attached box the variables on which RACER detects a data race. A main class `Main` starts all of the threads in the program. 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 commands the vehicle and reports back the status. The `RuntimeExecutive` issues tasks according to their planned execution time. For this purpose, tasks are stored in a queue. The `ExecTimer` thread then takes the tasks out of the queue and

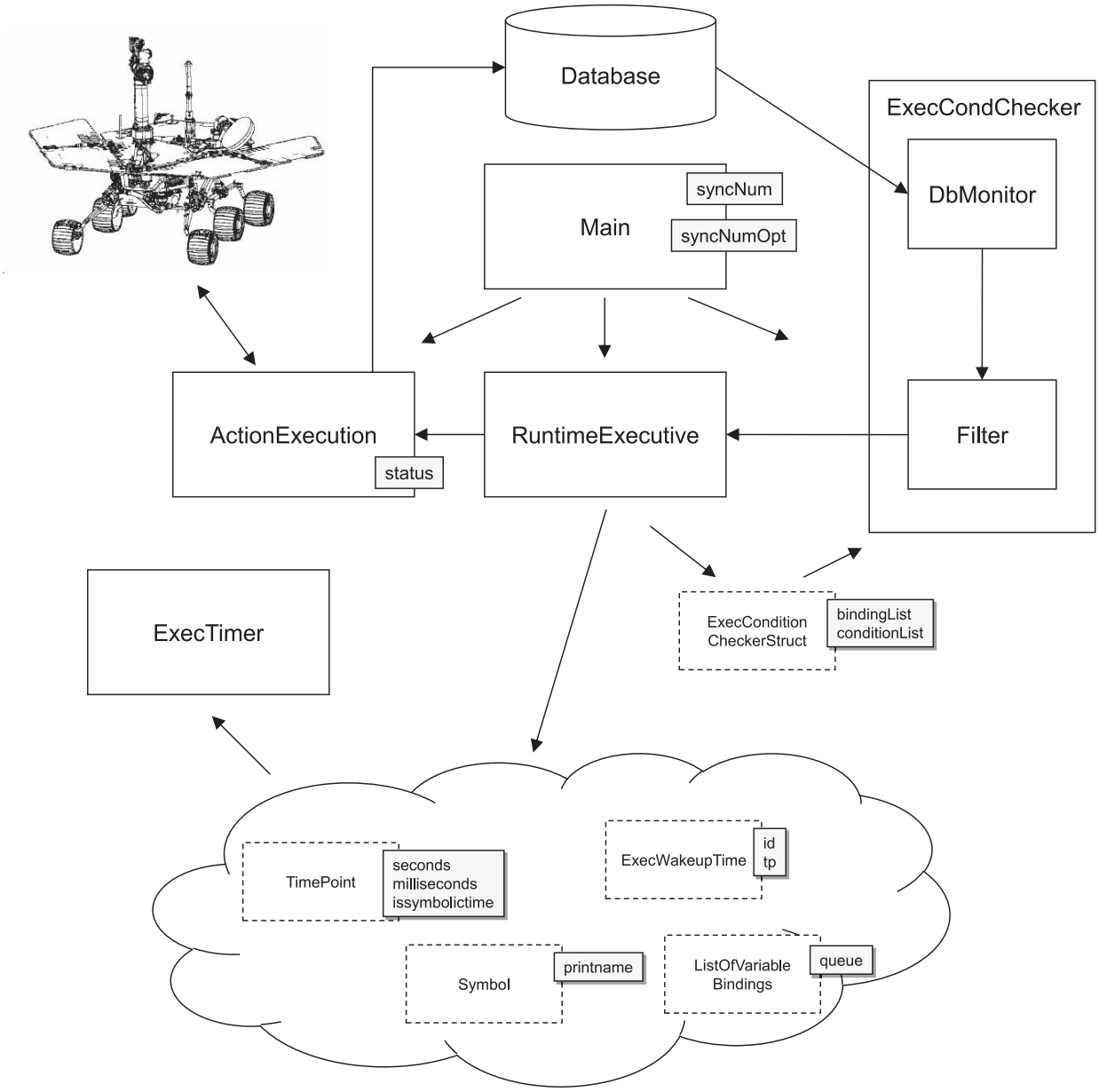

Fig. 14. The K9 Executive and its 12 unprotected fields.

executes them when their planned time is reached. The objects in the lower part of the figure represent time points, names of tasks (symbols), and variable bindings. 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`.

The K9 Executive consists of almost 7,000 lines of Java and is an abstracted version of 35,000 lines of C++ code that originally controlled the rover and were also developed at the 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. In this work, we focus on the core functionality. Researchers at the NASA Ames Research Center developed this code specifically to evaluate [30] a set of Java verification tools. These tools included the Java PathExplorer [5], which contained an earlier implementation of the ERASER algorithm for Java. Researchers further used this code to evaluate the Java PathFinder model checker [31], 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 researchers then tasked different groups of people with detecting the errors using different tools. After this experiment, researchers augmented the code with additional code to evaluate a deadlock analysis tool. From [30], we can cite: "*A total of 12 bugs were extracted from the CVS logs, of which five were deadlocks, two were data races, and five were plan-related. One of the deadlock bugs was given as an example during training on the tools, and one of the data races*

```
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
```

Fig. 15. A race reported by RACER.

was unreachable in the code that was eventually analyzed—thus leaving only 10 seeded errors." This suggests that the code contains one data race.

5.2 Application of Racer

At first, we were therefore surprised to see that running RACER on the K9 Executive revealed 12 data races. We categorized these races into three classes:

- One known data race on `ActionExecution.status`.
- Two data races on variables `syncNum` and `syncNumOpt` in `Main`, which had been just recently introduced.
- Nine data races that occurred because the programmers made wrong assumptions with respect to the initialization of variables.

5.2.1 The Race on ActionExecution.status

To indicate a data race on a variable `status` in class `ActionExectution`, RACER issues the message shown in Fig. 15. The `ActionExecution` thread and the `RuntimeExecutive` thread cause this race because they both access `status` without both first acquiring a common lock. This is exactly the error planted in the code during the original verification experiment [30].

5.2.2 The Races on syncNum and syncNumOpt

To be brief, 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 [30]) in order to determine how effectively a static analysis algorithm could reduce the number of locking operations that needed to be monitored to detect a deadlock. For this purpose, researchers added two integer counters to the `Main` class: `numSync` (number of synchronizations executed in total) and `numSyncOpt` (number of synchronizations executed after optimization). It turned out that multiple threads updated these counters without protecting the updates with a lock.

5.2.3 The Races between the RuntimeExecutive, the ExecTimer, and ExecCondChecker

In Fig. 16, we show the situation between the two threads `RuntimeExecutive` and `ExecTimer` that cause another seven data races which RACER reported on the K9 Executive. The `Main` thread starts both these threads, one right after the other. The `RuntimeExecutive` then

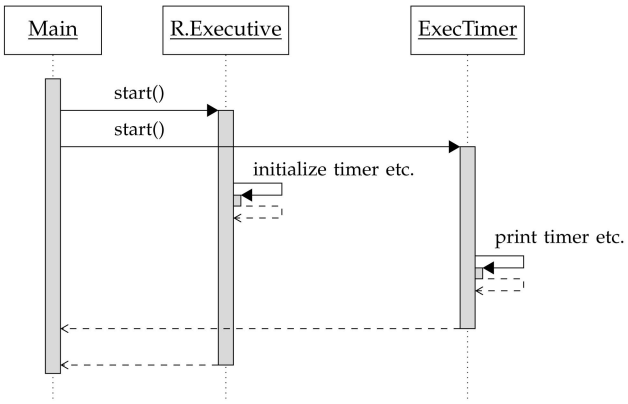

Fig. 16. Access periods in races between threads Executive and ExecTimer.

initializes certain data structures, like timers, for example, which the `ExecTimer` then accesses periodically. Because none of `RuntimeExecutive` or `ExecTimer` is a parent thread of the other, such accesses are only safe when using a consistent locking discipline. This is, however, not the case in the K9 executive: `RuntimeExecutive` locks on itself, while `ExecTimer` locks on itself too, but they never both attempt to acquire the same lock. The locking discipline is hence inconsistent. RACER found seven races on seven different fields that all fall into this same category. RACER found two additional races between the `RuntimeExecutive` and the `ExecCondChecker`. Also, here, the `RuntimeExecutive` initializes data structures, which the `ExecCondChecker` then accesses without holding an appropriate lock.

5.2.4 Observations

Researchers analyzed the code of the K9 Executive earlier [30] but detected only the seeded data race described in Section 5.2.1. The errors described in Section 5.2.2 were introduced at a later point, which explains why researchers were unable to find these races at the time of the experiment. Nevertheless, the researchers could have detected the nine other races, but missed them instead. The version of RACER presented in this paper reported no false warnings on the K9 Executive. It is important to note that our initial implementation presented at ISSTA 2008 did report 58 more potential data races that we subsequently identified as false warnings. The crucial difference between the two algorithms is that the new algorithm tracks access periods instead of just threads, and can select transitions based on the visibility relation $\preceq$. This treatment handles calls to `Thread.start()` precisely, which was not the case in the ISSTA version of this algorithm.

6 Further Experiments

In addition to our in-depth case study, we further applied RACER to two more smaller benchmarks taken from [17]. This was to find out whether the results gained from our case study can 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*

TABLE 1
Experimental Results

	roller	roller-opt	bank	bank-opt	rover	rover-opt	unit
compilation time	0:25	1:32	0:25	1:23	1:10	2:52:09	[h:]mm:ss
runtime without instrumentation	0:32	0:30	0:21	0:22	0:02	0:02	m:ss
runtime with instrumentation	6:42	6:44	10:29	10:22	0:02	0:02	m:ss
last race detected after	0.07	0.08	0.06	0.07	0.79	0.77	s
instrumented fields	9	8	16	15	260	169	
reported races	1	1	0	0	12	12	
actual races	0	0	0	0	12	12	
false positives	1	1	0	0	0	0	

compete for seven seats in one roller-coaster thread" [17]. This benchmark exposes very high contention. *bank* is a little banking application by Lea [32]. It starts eight threads which each make a random transaction from one account to another and then call `Thread.yield()`. Both benchmarks were written by researchers in the field of concurrent programming and hence we did not expect to find any data races in these benchmarks. Nevertheless, we wanted to see whether RACER would report for these benchmarks similarly few false positives as it reported for the rover case study.

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 64-Bit Server VM (version 1.6.0_05, mixed mode, 1 GB heap space), 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+.

6.1 Compilation Time

The compilation time is low without optimization, generally below 2 minutes. The static whole-program optimization adds about one and a half minutes of compilation time to our smaller benchmarks. In the case of the K9 rover, however, compilation takes almost 3 hours 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.

The next section of Table 1 shows the runtimes for the different configurations. In the case of the two small benchmarks, our instrumentation adds around 13-fold (*roller*) and 28-fold (*bank*) overhead. Through profiling, we determined that much of this slowdown is caused by contention. Both benchmarks spend around 70 percent 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. In the meantime, all other threads have to wait for t to finish. 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 Figs. 7 and 8, e.g., to:

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

Even though we believe that the slowdown caused by our `Racer` implementation is acceptable, it may still cause an effectiveness problem with respect to so-called Heisenbugs [33]. Named after the Heisenberg Uncertainty Principle, a Heisenbug is a computer bug which becomes invisible when one attempts to investigate it. In our particular setting, the instrumentation that the RACER implementation inserts has to use locks in order to maintain consistent data structures during the course of the program evaluation. These additional locks introduce additional lock contention, and generally, reduce the amount of concurrency that the uninstrumented program may have allowed. As a consequence, certain interleavings may become more unlikely than they would have been in the uninstrumented program, and hence, some bugs may be hard to detect. Nevertheless, this is a problem that the most dynamic approach to finding concurrency-related problems faces. The only way to avoid this problem would be to control the thread scheduler, which is certainly out of the scope of our current research.

Next, we comment on the number of instrumented fields. 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 are indeed shared. Therefore, the optimization was ineffective and the runtime did not improve. In the case of the rover code, the static analysis detected about one third of the 260 fields as thread-local, and hence, removed all instrumentation for these fields in the optimized version. However, since the rover code already finished execution after 2 seconds, there was no perceivable improvement in runtime either.

In *roller*, the RACER algorithm reported one data race, which turned out to be a false warning. In this benchmark, each passenger thread constantly reads a counter `rideNo` that holds the global number of rides performed so far. When performing this read on `rideNo`, the passenger thread locks on the roller-coaster thread object. The roller-coaster thread updates `rideNo` after each ride, and while doing so, it also locks on the same object as the passenger threads. However, the roller-coaster thread also reads from the field `rideNo`, and this read is performed *outside* of the locked region. This does not cause a race because the roller-coaster thread is the

only thread ever updating `rideNo`, and these updates are implicitly visible to the roller-coaster thread itself. The updates are also visible to the passenger threads because they follow a consistent locking discipline. The RACER algorithm issues a false warning for this benchmark because it reaches the SharedModified state with an empty lock set due to the fact that the roller-coaster thread reads the field without holding any lock.

In the *bank* benchmark, RACER reports no races, and indeed, we believe that this code is race-free. In Section 5, we already commented on the races in the Rover Executive.

7 Related Work

In the following, we compare RACER to other algorithms for detecting concurrency-related programming errors, compare our AspectJ-based instrumentation approach to other approaches for instrumenting Java bytecode, and discuss related work in the field of aspect-oriented programming.

7.1 Detecting Concurrent Programming Errors

7.1.1 Eraser

Savage et al. proposed the original ERASER algorithm [19] for detecting potential for data races. ERASER was an important contribution to the field; many researchers have since based their own race-detection algorithms on ERASER. As we showed, ERASER is, however, very forgiving to the programmer in an object's initialization phase, and therefore, can miss important race conditions that occur during an object's initialization. As our experimental results show, such races are not uncommon in Java programs. RACER, on the other hand, guarantees to find a data race when this race occurs on a monitored program run. Unfortunately, ERASER not only misses an important class of data races, it can also yield many false warnings. The prototype version of RACER that we published at ISSTA 2008 parameterized states with threads instead of access periods, just like ERASER does, and led to many false warnings. Access periods are important because they allow RACER to gain precision when the variables are accessed through child threads. According to the Java Memory Model, such an access is safe without locking, and a race-detection algorithm for Java should take this into account.

7.1.2 Visual Threads

The idea of using access periods is not new. Harrow implemented an extension to ERASER in Compaq's Visual Threads tool [1] that uses access periods (called "thread segments" in Harrow's paper) to model lock-free handover of objects between child and parent threads. Unlike our implementation of RACER, Harrow's implementation still uses the original ERASER state machine, and therefore, Harrow's implementation misses the same initialization-related data races that the original ERASER misses. Programmers can use Visual Threads with any application that uses a POSIX threads library, which includes common implementations of Java. Visual Threads analyzes multithreaded applications for potential logic and performance problems. The tool visualizes state changes and provides automated dynamic analysis algorithms to diagnose common problems associated with multithreading, including deadlock, data protection, performance, and programming errors. Visual Threads uses the object code instrumentation tool ATOM [34].

7.1.3 Hybrid Dynamic Race Detection

Other researchers have tried to improve the accuracy of the ERASER algorithm. O'Callahan and Choi [2] use vector clocks [35] to explicitly keep track of a subset of the happens-before relationship of the program under test. The authors then refine an Eraser-like lock-set-based race-detection algorithm using this happens-before graph to reduce the amount of false warnings that the purely lock-set-based algorithm would otherwise yield. The happens-before graph that O'Callahan and Choi use contains happens-before edges at calls to `Thread.start()`, `Thread.join()`, `Object.wait()`, and `Object.notify()`. Like our RACER algorithm, O'Callahan and Choi's algorithm does not grant any special initialization phase, and therefore, should be able to detect the initialization-based races that ERASER misses, e.g., the one from Fig. 10. As a result, O'Callahan and Choi's algorithm should be similar in precision to RACER, in fact, even more precise in case the program under test uses `join`, `wait`, or `notify`. Because O'Callahan and Choi treat `wait` and `notify`, their algorithm should also be more precise than Harrow's extended version of ERASER for programs that use these methods. The only other difference between O'Callahan and Choi's algorithm and Harrow's algorithm seems to be that Harrow treats initialization like ERASER does, while O'Callahan and Choi treat it as in RACER. O'Callahan and Choi modify a program's bytecode to add instrumentation, but they do not say whether they do so using some special bytecode instrumentation toolkit.

7.1.4 Goldilocks

Elmas et al. present the GOLDILOCKS algorithm [36]. Unlike the other race-detection algorithms that we presented above, GOLDILOCKS constructs at runtime the complete happens-before relationship of a program run. This enables the GOLDILOCKS runtime to detect a data race if and only if this race actually occurs. Keeping track of the entire happens-before graph at runtime is challenging and requires sophisticated data structures to be efficient. Lock-set-based algorithms like the ones based on ERASER can be implemented more efficiently but will usually have to allow for some amount of false warnings.

7.1.5 Object Race Detection

Von Praun and Gross [3] formulate data races as properties of objects instead of variables. This is because the authors further propose using a static escape analysis to detect objects that are accessed by multiple threads, and this escape analysis, too, reasons about objects rather than variables. The escape analysis that Praun and Gross describe therefore has similar intent to Halpert et al.'s thread-local-objects analysis [17], which we use in this paper. Unfortunately, the authors provide insufficient detail about their static analysis to determine how much exactly it differs from Halpert et al.'s analysis. Praun and Gross further describe an ownership model that allows a thread t to hand over access permissions for an object o to another thread t', thereby allowing t' to access o instead t for the remainder of the program. When t has terminated by the time of the handover, then the handover can happen implicitly. However, if t is still active, then t' has to explicitly ask t to hand over the permissions to t'. The thread t' blocks until t grants the permissions, thereby making t' the "second owner" of o. In our understanding, any object o can only have two owners. This precludes programs

from having a thread t hand over objects to a thread t', which then, in turn, hands this object to another thread t'' (unless t has terminated at this point in time). RACER always hands over variables to child threads implicitly. The child thread t' then becomes the only thread holding read and write permissions for this variable, as signified by the states Exclusive(t) and ExclusiveModified(t). In RACER, this handover can happen again when t' starts a third thread t'', which then accesses the same variable. There is, therefore, no limit to the number of handovers. Praun and Gross instrument the program under test through by transforming Java bytecode into customized X86 native code.

7.1.6 Proving That Races Exist

In [4], we describe our first implementation of the ERASER algorithm for Java, guiding the Java PathFinder (JPF) model checker [31] 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. Havelund and Roşu later reimplemented and elaborated the algorithm in the Java PathExplorer (JpaX) tool [5]. This tool used the Jtrek bytecode instrumentation tool [37] and later the BCEL bytecode instrumentation tool [11]. With the AspectJ language extension proposed in this paper, researchers can define their instrumentation in a declarative way instead, and do not have to resort to Jtrek or BCEL.

7.1.7 Detecting Potential for Other Concurrency-Related Errors

Researchers have developed other kinds of dynamic race analysis tools, all of which detect potential for errors, like the ERASER algorithm, rather than directly detecting the occurrence of errors. Artho et al. proposed a high-level data race algorithm [6] which detects inconsistencies in which collections of variables are access protected by locks. If, for example, one part of a program accesses two variables x and y in one single synchronized block and another program part accesses the same variables in separate synchronized blocks, then 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. Even in the absence of low-level and high-level data races, programs can still contain other concurrency errors. Related to high-level data, races are atomicity violations as detected by the tools in [7], [8], [9]. 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. Chen et al.'s jPredictor [10] extracts a causality relation from the execution trace, sliced using static analysis, and refined with lock-atomicity information. The authors investigate two common types of errors: data races and atomicity violations. jPredictor's program instrumentor is built on top of the Soot [38] 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 an appealing syntax.

Programmers can also effectively use dynamic analyses to find potential for deadlocks. As mentioned, the Visual Threads tool detects potential for deadlocks, essentially by detecting cycles in a lock graph. Bensalem and Havelund [21] and Agrawal et al. [39] 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 including wait statements or by modifying schedulers so that programs will exhibit a fuller range of nondeterministic behaviors during testing. Researchers have combined such modifications with predictive analysis [40], [41].

7.1.8 Specification-Based Approaches

All algorithms mentioned above work without requiring 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 [42]. Tracematches's [16] answer provides an efficient implementation of runtime monitoring with object bindings as a language extension to AspectJ. Bodden et al. [43], [44] used trace matches to prove Java and AspectJ programs partially correct. Trace matches can directly use the three novel pointcuts proposed here.

7.2 Tools for Instrumenting Bytecode

Apart from trace matches, however, in the case of Java, researchers usually use bytecode instrumentation tools when building the bug detectors that we mentioned. Examples are Jtrek [37], BCEL [11], Soot [38], and ASM [45]. Similar tools for other languages include Valgrind [46], ATOM [34], and the C source code instrumentation and analysis tool CIL [47]. 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 middleware services. Researchers have proposed higher level libraries on top of the low-level instrumentation packages. In previous work, for example, we developed the jSpy tool [48], which instruments Java bytecode, but using a higher level of primitives compared to what the low-level bytecode instrumentation tools offer. A jSpy instrumentation specification consists of a set of rules, each of which consists of a condition on bytecode and an instrumentation action stating what to report when bytecodes satisfying the condition executes. Monitors then pick up the reported events and check for various user-provided properties. The tool is oriented toward monitoring rather than modifying functionality. Another high-level instrumentation tool is Sofya [49].

The main observation is that most, if not all, of the dynamic analysis tools described above 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.

7.3 Aspect-Oriented Programming

For quite a while now, the community around aspect-oriented programming has been calling for more "semantic pointcuts" (e.g., [50], [51]), which allow programmers not to match on a program's structure like a call to a method

`foo()`, but on more semantic properties. We generally agree with this point of view, and 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 errors related to concurrency. 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 forgiving to programmers in an object's initialization phase. RACER, on the other hand, detects and also reports races that comprise the initialization of an object. This revealed 12 data races in program code of the NASA K9 Rover Executive, 11 of which went previously undetected, although extensive studies of this code had already been performed at a time when nine of these undetected races were already present.

Acknowledgments

The authors thank Clark Verbrugge for helping them validate some of the data races they found in the K9 rover executive. Also, they are grateful to him, Richard Halpert, and Chris Pickett for making their thread-local-objects analysis and their benchmarks available to them. They thank Stefan Savage for providing clarifications on ERASER. Bill Pugh pointed out an important shortcoming of their original RACER implementation. They also wish to thank the anonymous reviewers of *TSE* for their helpful comments on this paper. 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 US National Aeronautics and Space Administration. Eric Bodden wishes to thank his supervisor, Laurie Hendren, for all her support during his graduate studies at McGill. The RACER implementation is available online at http://www.bodden.de/tools/raceraj/. The AspectBench Compiler can be download at http://www.aspectbench.org/. Since version 1.3.0, it contains the authors' implementation of the pointcuts `lock()`, `unlock()`, and `maybeShared()` in the extension `abc.eaj`.

References

[1] J. Harrow, "Runtime Checking of Multithreaded Applications with Visual Threads," *SPIN Model Checking and Software Verification,* K. Havelund, J. Penix, and W. Visser, eds., pp. 331-342, Springer, 2000.

[2] R. O'Callahan and J.-D. Choi, "Hybrid Dynamic Data Race Detection," *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming,* pp. 167-178, 2003.

[3] C. von Praun and T.R. Gross, "Object Race Detection," *Proc. Ann. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications,* pp. 70-82, 2001.

[4] K. Havelund, "Using Runtime Analysis to Guide Model Checking of Java Programs," *SPIN Model Checking and Software Verification,* pp. 245-264, Springer, 2000.

[5] K. Havelund and G. Roşu, "An Overview of the Runtime Verification Tool Java PathExplorer," *Formal Methods in System Design,* vol. 24, no. 2, pp. 189-215, 2004.

[6] C. Artho, K. Havelund, and A. Biere, "High-Level Data Races," *Software Testing, Verification and Reliability,* vol. 13, no. 4, pp. 207-227, 2003.

[7] C. Artho, K. Havelund, and A. Biere, "Using Block-Local Atomicity to Detect Stale-Value Concurrency Errors," *Automated Technology for Verification and Analysis,* F. Wang, ed., pp. 150-164, Springer, 2004.

[8] C. Flanagan and S.N. Freund, "Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs," *Proc. 31st ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages,* pp. 256-267, 2004.

[9] L. Wang and S.D. Stoller, "Run-Time Analysis for Atomicity," *Electronic Notes in Theoretical Computer Science,* vol. 89, no. 2, 2003.

[10] F. Chen, T.F. Serbanuta, and G. Roşu, "jPredictor: A Predictive Runtime Analysis Tool for Java," *Proc. 30th Int'l Conf. Software Eng.,* pp. 221-230, 2008.

[11] M. Dahm, "BCEL," http://jakarta.apache.org/bcel, 2010.

[12] E. Bodden, "J-LO—A Tool for Runtime-Checking Temporal Assertions," master's thesis, RWTH Aachen Univ., http://www.bodden.de/publications/, Nov. 2005.

[13] M. d'Amorim and K. Havelund, "Event-Based Runtime Verification of Java Programs," *Proc. Third Int'l Workshop Dynamic Analysis,* pp. 1-7, 2005.

[14] V. Stolz and E. Bodden, "Temporal Assertions Using AspectJ," *Electronic Notes in Theoretical Computer Science,* vol. 144, no. 4, pp. 109-124, 2006.

[15] F. Chen and G. Roşu, "MOP: An Efficient and Generic Runtime Verification Framework," *Proc. Ann. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications,* R.P. Gabriel, D.F. Bacon, C.V. Lopes, J. Guy, and L. Steele, eds., pp. 569-588, 2007.

[16] 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," *Proc. Ann. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications,* R. Johnson and R.P. Gabriel, eds., pp. 345-364, 2005.

[17] R.L. Halpert, C.J.F. Pickett, and C. Verbrugge, "Component-Based Lock Allocation," *Proc. 16th Int'l Conf. Parallel Architectures and Compilation Techniques,* pp. 353-364, 2007.

[18] 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," *Proc. Int'l Conf. Aspect-Oriented Software Development,* pp. 87-98, 2005.

[19] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A Dynamic Data Race Detector for Multithreaded Programs," *ACM Trans. Computer Systems,* vol. 15, no. 4, pp. 391-411, 1997.

[20] E. Bodden and K. Havelund, "Racer: Effective Race Detection Using AspectJ," *Proc. Int'l Symp. Software Testing and Analysis,* pp. 155-165, July 2008.

[21] S. Bensalem and K. Havelund, "Dynamic Deadlock Analysis of Multi-Threaded Programs," *Proc. Haifa Verification Conf.,* S. Ur, E. Bin, and Y. Wolfsthal, eds., pp. 208-223, 2005.

[22] "The AspectJ Home Page," http://eclipse.org/aspectj/, 2010.

[23] "The AspectJ Programming Guide," http://www.eclipse.org/aspectj, 2010.

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

[25] O. Lhoták and L. Hendren, "Scaling Java Points-to Analysis Using Spark," *Proc. 12th Int'l Conf. Compiler Construction,* G. Hedin, ed., pp. 153-169, Apr. 2003.

[26] J. Manson, W. Pugh, and S.V. Adve, "The Java Memory Model," *Proc. 32nd ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages,* pp. 378-391, 2005.

[27] E. Bodden, F. Forster, and F. Steimann, "Avoiding Infinite Recursion with Stratified Aspects," *Proc. Int'l Conf. Grid Service Eng. and Management,* R. Hirschfeld, A. Polze, and R. Kowalczyk, eds., pp. 49-64, 2006.
[28] B. Goetz, *Java Concurrency in Practice.* Addison Wesley, 2006.
[29] T. Aotani and H. Masuhara, "SCoPE: An AspectJ Compiler for Supporting User-Defined Analysis-Based Pointcuts," *Proc. Int'l Conf. Aspect-Oriented Software Development,* pp. 161-172, 2007.
[30] 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,* vol. 25, nos. 2/3, pp. 167-198, 2004.
[31] W. Visser, K. Havelund, G.P. Brat, S. Park, and F. Lerda, "Model Checking Programs," *Proc. 15th IEEE Int'l Conf. Automated Software Eng.,* pp. 203-232, 2003.
[32] D. Lea, *Concurrent Programming in Java: Design Principles and Patterns.* Addison-Wesley Longman Publishing Co., Inc., 1996.
[33] J. Gray, "Why Do Computers Stop and What Can Be Done about It?" *Proc. Fifth Symp. Reliability in Distributed Software and Database Systems,* pp. 3-12, 1986.
[34] A. Eustace and A. Srivastava, "ATOM: A Flexible Interface for Building High Performance Program Analysis Tools," *Proc. USENIX Winter '95 Technical Conf.,* p. 25, 1995.
[35] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM,* vol. 21, no. 7, pp. 558-565, 1978.
[36] T. Elmas, S. Qadeer, and S. Tasiran, "Goldilocks: A Race and Transaction-Aware Java Runtime," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 245-255, 2007.
[37] S. Cohen, "Jtrek," Compaq, no longer maintained.
[38] "Soot Website," http://www.sable.mcgill.ca/soot/, 2010.
[39] R. Agarwal, L. Wang, and S.D. Stoller, "Detecting Potential Deadlocks with Static Analysis and Run-Time Monitoring," *Proc. Haifa Verification Conf.,* S. Ur, E. Bin, and Y. Wolfsthal, eds., pp. 191-207, 2005.
[40] S. Bensalem, J.-C. Fernandez, K. Havelund, and L. Mounier, "Confirmation of Deadlock Potentials Detected by Runtime Analysis," *Proc. 2006 Workshop Parallel and Distributed Systems: Testing and Debugging,* pp. 41-50, 2006.
[41] 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,* vol. 19, no. 3, pp. 267-279, 2007.
[42] H. Barringer, A. Goldberg, K. Havelund, and K. Sen, "Rule-Based Runtime Verification," *Proc. Int'l Conf. Verification, Model Checking, and Abstract Interpretation,* B. Steffen and G. Levi, eds., pp. 44-57, 2004.
[43] E. Bodden, L.J. Hendren, and O. Lhoták, "A Staged Static Program Analysis to Improve the Performance of Runtime Monitoring," *Proc. European Conf. Object-Oriented Programming,* E. Ernst, ed., pp. 525-549, 2007.
[44] 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/, 2007.
[45] E. Bruneton, R. Lenglet, and T. Coupaye, "ASM: A Code Manipulation Tool to Implement Adaptable Systems," *Adaptable and Extensible Component Systems,* http://asm.objectweb.org/, Nov. 2002.
[46] "Valgrind," http://valgrind.org/, 2010.
[47] G.C. Necula, S. McPeak, S.P. Rahul, and W. Weimer, "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs," *Proc. Int'l Conf. Compiler Construction,* R.N. Horspool, ed., pp. 213-228, 2002.
[48] A. Goldberg and K. Havelund, "Instrumentation of Java Bytecode for Runtime Analysis," *Proc. Fifth ECOOP Workshop Formal Techniques for Java-Like Programs,* July 2003.
[49] A. Kinneer, M.B. Dwyer, and G. Rothermel, "Sofya: Supporting Rapid Development of Dynamic Program Analyses for Java," *Companion to the Proc. 29th Int'l Conf. Software Eng.,* pp. 51-52, 2007.
[50] M. Eichberg, M. Mezini, and K. Ostermann, "Pointcuts as Functional Queries," *Proc. Second ASIAN Symp. Programming Languages and Systems,* W.-N. Chin, ed., pp. 366-381, 2004.
[51] T. Aotani and H. Masuhara, "Compiling Conditional Pointcuts for User-Level Semantic Pointcuts," *Proc. Software-Eng. Properties of Languages and Aspect Technologies Workshop,* Mar. 2005.
[52] *Hardware and Software Verification and Testing,* S. Ur, E. Bin, and Y. Wolfsthal, eds. Springer, 2006.

Eric Bodden received the diploma from RWTH Aachen University, Germany, in 2005. Having completed his doctoral dissertation in the Sable Research Group at McGill University, Montréal, Canada, he now continues his research as a postdoctoral fellow at the Technical University Darmstadt, Germany. His research interests include static and dynamic analyses that allow programmers to reason about large-scale object-oriented programs. For his dissertation, he developed CLARA, a framework for evaluating finite-state runtime monitors at compile time. He seeks applied solutions, combining compilation and sound static program analysis techniques with unsound and incomplete techniques from Software Engineering. Early on, he recognized the potential of aspect-oriented programming as a convenient abstraction for program analysis and verification.

Klaus Havelund received the PhD degree in computer science from the University of Copenhagen, Denmark, in 1994 (executed at the Ecole Normale Supérieure, Paris, France). He is a senior research scientist at the Laboratory for Reliable Software, Jet Propulsion Laboratory, California Institute of Technology. His research interests include verification of software, in particular using runtime verification techniques such as specification-based monitoring and dynamic concurrency analysis. He developed the first prototype of the Java PathFinder model checker, capable of model checking Java programs, and has subsequently developed dynamic analysis tools for detection of data races and deadlocks in Java programs.