Finding Synchronization Defects in Java Programs: Extended Static Analyses and Code Patterns

Frank Otto
University of Karlsruhe
76131 Karlsruhe, Germany
otto@ipd.uka.de

Thomas Moschny
University of Karlsruhe
76131 Karlsruhe, Germany
moschny@ipd.uka.de

ABSTRACT

Concurrent programming is getting more and more important. Managing concurrency requires the usage of synchronization mechanisms, which is error-prone. Well-known examples for synchronization defects are deadlocks and race conditions. Detecting such errors is known to be difficult. There are several approaches to identify potential errors, but they either produce a high number of false positives or suffer from high computational overhead, catching only a small number of defects. Our approach uses static analysis techniques combined with points-to and may-happen-in-parallel (MHP) information to reduce the number of false positives. Additionally, we present code patterns indicating possible synchronization problems. We have implemented our approach using the Java framework Soot. Our tool was tested with small code examples, an open source web server, and commercial software. First results show that the number of false positives is reduced significantly.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*; D.2.4 [**Software Engineering**]: Software/Program Verification—*reliability*

General Terms

Reliability, verification.

Keywords

Java, synchronization defects, lockset analysis, anti-pattern detection.

IWMSE'08, May 11, 2008, Leipzig, Germany.

1. INTRODUCTION

Software systems often perform several tasks in parallel, operating on shared data. It is well-known that correct synchronization of parallel applications is difficult. This paper presents a new method for detecting code patterns that may result in deadlocks or race conditions or indicate inefficient synchronization. We focus on shared memory architectures such as multicore systems.

A deadlock happens if threads wait for each other such that none of them is able to make any progress [10]. A race condition occurs when at least two threads access the same resource simultaneously, without sufficient protection [1]. Detecting such errors is difficult because concurrent programs are non-deterministic: the outcome of a parallel application may vary from run to run even if the input is the same.

1.1 Concurrency in Java

Java [9] has become a widespread platform for developing concurrent applications. It explicitly supports concurrency by the `Thread` class and the `Runnable` interface. A key concept for synchronizing threads in Java is provided in form of *monitors* [10]. A monitor is declared by the keyword `synchronized` and protects so-called *critical sections*. Critical sections are guarded by *locks* that guarantee that only a single thread can execute the statements within the section at any time. Before a thread can execute that code, it has to acquire (or already hold) the corresponding lock. In Java, every object and class can be declared a monitor.

1.2 Contributions

To the best of our knowledge, our approach is the first one to combine static analysis with points-to and MHP information to effectively detect synchronization defects in Java programs. The main contribution of our work is an extended static analysis, which – compared to other approaches – produces a tiny number of false positives. Based on the extended analysis, we present detection strategies for code patterns that might lead to synchronization defects.

1.3 Outline

The rest of the paper is organized as follows. Related work is discussed in Section 2. Section 3 presents a brief outline of our approach. Section 4 describes the lockset analysis. Section 5 gives an overview about points-to and MHP analysis techniques, which we use to increase the accuracy of our analysis. Section 6 describes code patterns indicating potential synchronization defects and demonstrates how to

find them in real code based on the previous analysis. Experiences are discussed in Section 7, while Section 8 presents conclusions and future work.

2. RELATED WORK

Detecting synchronization defects is difficult because the execution of concurrent programs is non-deterministic. The thread schedule generally cannot be influenced by the application and may vary from run to run even for the same program input. Three core approaches for detecting such errors are dynamic testing, static analysis, and model checking.

Dynamic testing examines a program's behavior at runtime, i.e. the program is executed with sample input in order to find errors. The main problem is that only a small number of execution paths can be examined. If no problems occur during testing, this does not necessarily mean that the program is actually correct. One of the first tools for detecting race conditions is Eraser [16]. It records locksets and emits warnings if there are unprotected accesses to shard objects. ConTest [3] is a tool which automatically reruns tests to detect synchronization faults.

Static analysis focuses on the structure of the program and does not require its execution. All possible (but also infeasible) execution paths can be taken into consideration. In general, static analyses run faster than dynamic tests but are likely to produce a high number of false positives due to infeasible paths and imprecise local information about the program. JLint [1] is a well-known tool for detecting synchronization faults and other runtime errors in Java programs. It runs extremely fast, but is not precise enough and reports numerous false positives. RacerX [4] is a tool for detecting deadlocks and data races in C programs. It uses a lockset algorithm, heuristics and belief analysis to identify possible faults. ESC/Java [7] is a tool examining annotated code. It checks inconsistencies between the design decisions expressed by the annotations and the actual code and warns of potential runtime errors including synchronization problems. Findbugs [6] is a freely available tool looking for bugs in Java code. There are bug categories such as "multi-threaded correctness", which include pattern-like descriptions of bugs. However, these patterns are simple, since they only refer to single code fragments instead to the whole program, which is less effective.

Model Checking is a verification technique that checks if certain properties hold for a model of a given program. One challenge is to construct a "good" model, which should be as simple as possible but also provide sufficient information about the program. Related to this aspect is the state explosion problem, i.e. the fact that the actual size of the model is usually exponentially larger than the size of its implicit description. A well-known tool is Java PathFinder [19], which can model check Java programs of a size up to 10 KLOC.

3. OVERVIEW

In this section, we give an overview of how our approach works. It was built on top of the Soot framework [17] and encompasses the following steps:

1. Our Java-adapted lockset algorithm computes the sets of locks held at each statement of the program. This algorithm operates on Jimple, an intermediate representation used by Soot.

2. The results of the lockset algorithm are used (i) to retrieve information about the order in which locks are acquired (so-called constraints) and (ii) to group the program into blocks, i.e. sequences of statements protected by the same lock.

3. We extract precise information about constraints and blocks by running points-to and MHP analyses. By this, we can quickly answer questions like "Which objects might this lock name refer to?", "Which blocks may share objects?", or "May these two blocks be executed in parallel?".

4. On that basis, we use detection algorithms to search for code patterns that might lead to synchronization defects. In this paper, we present five code patterns including detection strategies.

5. The findings are presented in form of a problem report, which can be filtered by the user.

4. LOCKSET ANALYSIS

A fundamental part of our approach is a lockset algorithm in the style of RacerX [4]. It operates on Jimple, an intermediate representation used by Soot. For each single statement, this algorithm computes the set of locks which are held at this point. This set is called *lockset*.

Our algorithm traverses the whole program, starting with the first statement of the `main` method. If there is a method call, the callee will be analyzed with respect to the context of the call, which is given by the so-called *entry lockset* consisting of those locks held at the point where the call happens. Since each method can be called within different contexts (i.e. with different entry locksets), separate analyses have to be done for each of those.

Figure 1 shows an example analysis for a simple Java program. Whenever a monitor is entered, the variable name of this monitor is added to the lockset. On exit of the monitor, the corresponding name is removed from the lockset.

The lockset analysis is a classical data flow problem. The purpose of a data flow analysis is to retrieve facts for each point of a program [15]. This usually happens based on the program's control flow graph by computing so-called $GEN(s)$, $KILL(s)$, $IN(s)$ and $OUT(s)$ sets for all statements s. $GEN(s)$ and $KILL(s)$ contain those facts which have to be added or removed for s. $IN(s)$ and $OUT(s)$ contain those facts which hold before or after s.

The goal of our lockset analysis is to compute the locksets for each statement of a program. For a statement s, we define $GEN(s) = \{x\}$, if lock x is acquired by s, and $KILL(s) = \{x\}$, if x is released by s. The flow through a statement s is described by $OUT(s) = (IN(s) \setminus KILL(s)) \cup GEN(s)$.

Determining locksets serves two purposes: to derive so-called *constraints* and to collect information about the program on the basis of *blocks*. These are defined in the following two subsections.

4.1 Constraints

A *constraint* is an expression $a \rightarrow b$, where a and b are locks. $a \rightarrow b$ means that at some point in the program lock b is acquired while a is already held. In principle, constraints

```
                                          GEN   KILL   lockset
public static void main(String[] args) {  {}    {}     {}
  foo();              // context c1       {}    {}     {}
  synchronized(o) {                      {o}   {}     {o}
    foo();            // context c2       {}    {}     {o}
  }                                      {}    {o}    {}
}                                        {}    {}     {}

                                                      c1     c2
static void foo() {                      {}    {}     {}     {o}
  synchronized(p) {                      {p}   {}     {p}    {o, p}
    // ...                               {}    {}     {p}    {o, p}
  }                                      {}    {p}    {}     {o}
}                                        {}    {}     {}     {o}
```

Figure 1: Lockset analysis for a simple Java program

describe the order in which locks can be acquired. They are essential for computing potential locking cycles, which may lead to deadlocks.

The constraints can be derived from the locksets computed for the program: if, for some part of code of the program, there is a lockset $\{a\}$ followed by another lockset $\{a, b\}$, the constraint $a \rightarrow b$ is emitted. For the program fragment in Figure 1, the only constraint is `o` $\rightarrow$ `p`, which holds inside the method `foo()` if it is called from context `c2`.

4.2 Blocks

A *block* is a sequence of statements that are part of a method and either protected by the same lock or not protected. If a block is protected by a lock, we call it a *synchronized block*, and *unsynchronized block* otherwise. Blocks are a fundamental entity with respect to synchronization. They can be easily derived from the locksets: a sequence of statements with the same lockset belongs to one block.

Grouping the program into blocks allows dedicated views on it: for example, to determine which blocks are protected by the same lock, whether two given blocks may be executed in parallel, or which variables are referenced in a given block. Answering questions like these requires more precise information, which we gain from points-to and MHP analysis. These techniques are described in the next section.

5. POINTS-TO AND MHP ANALYSIS

To increase the accuracy of our analysis, we use points-to and may-happen-in-parallel (MHP) analysis. The goal of points-to analysis is to determine the set of objects pointed to by a variable or field carrying a reference. Given a variable v, we define $PS(v)$ to be the points-to set of v. Our approach uses SPARK [11], a flexible points-to analysis framework for Soot [17].

For our work, points-to information is needed to determine whether two variables v_1, v_2 may point to the same object at runtime. This is the case if the corresponding points-to sets have a non-empty intersection, i.e. if $PS(v_1) \cap PS(v_2) \neq \emptyset$. For example, it can be determined whether (1) the same object might be accessed from two different blocks or (2) two "different" locks might actually be the same since their names refer to the same object.

MHP analysis [14] determines whether two pieces of code might be executed in parallel. The Soot-based approach by Lin Li [12] for two methods m_1, m_2 returns $true$ if m_1 and m_2 may be executed in parallel (which does not necessarily mean that this actually happens) and $false$ if they are never executed in parallel. We modified this approach to operate on blocks (as defined in Section 4.2) instead of methods, i.e. we provide a boolean function $mhp(b_1, b_2)$, where b_1 and b_2 are blocks. Since this function also incorporates information about monitors, it is more precise than the original approach.

6. CODE PATTERNS

So far, we have shown how to use the results of the lockset analysis combined with points-to and MHP information to retrieve precise information about the constraints and blocks of a program. On that basis, we use detection algorithms to search for code patterns that might result in synchronization defects. This section gives an overview of some patterns that we defined based on the work of Lea [10] and Farchi et al. [5]. For each pattern, we give example code, briefly describe the problem, and present detection strategies.

6.1 Cyclic Lock Dependencies

The following two code fragments, which are both part of a parallel application, acquire locks in different orders:

```
void foo() {
  synchronized(o1) {
    synchronized(p1) {
      // ...
    }
  }
}
```

```
void bar() {
  synchronized(p2) {
    synchronized(o2) {
      // ...
    }
  }
}
```

Problem. A cyclic lock dependency is the classical precondition of a deadlock [20]. The existence of cyclic dependencies can be determined by considering the constraints of the program. The above example consists of two constraints `o1` $\rightarrow$ `p1`, `p2` $\rightarrow$ `o2`, such that there are cyclic dependencies provided that `o1` in `foo()` and `o2` in `bar()` as well as `p1` in `foo()` and `p2` in `bar()` may point to the same objects, respectively. In this case, a deadlock may occur when `foo()` and `bar()` are executed in parallel.

Detection.
(1) Consider the set C of all constraints of the program.
(2) Build a directed graph $G = (V, E)$. We define vertexes $V := C$ and edges $E := \{(c_1 \rightarrow c_2, c_1' \rightarrow c_2') \in C \times C : PS(c_2) \cap PS(c_1') \neq \emptyset\}$. That is, each constraint is represented by a vertex, and an edge between two constraints says that the second lock of the first constraint may point to the same object as the first lock of the second constraint.
(3) Find all cycles in G, e.g. by using a depth-first algorithm.
(4) For each cycle, determine the set B of involved blocks.
(5) If $mhp(b_1, b_2) = true$ for all $b_1, b_2 \in B$, report the cycle.

6.2 Superfluous Lock

Consider the body of a synchronized block, although it never accesses shared objects concurrently:

```
synchronized(o) {
  // only non-critical operations
  // ...
}
```

Problem. A superfluous lock is a lock protecting some code that only performs "non-critical" operations. We define an operation to be non-critical if it never accesses any shared object concurrently. A superfluous lock may result in inefficient synchronization by making other threads unnecessarily block.

Detection. For each synchronized block b of the program:
(1) Consider the set B of blocks that may be executed in parallel to b, i.e. $B := \{b' : mhp(b, b') = true\}$.
(2) For each block $b' \in B$, check if b and b' might access shared objects such that there could be read-write- or write-write-conflicts.
(3) If this is not the case, then b is probably unnecessarily protected. Report b.

6.3 Missing or Wrong Lock

The following two code fragments access a shared object `x`, but they are not protected by the same lock:

```
void foo() {
  synchronized(o) {
    // ...
    x++;
  }
}
```

```
void bar() {
  x++;
  // ...
}
```

Problem. Let F_1 and F_2 be code fragments. Let F_1 be protected by a lock l_1 and F_2 be protected by either a different lock l_2 or no lock at all. In the first case, we have a wrong lock for F_2, in the second case, we have a missing lock. If F_1 and F_2 are executed in parallel, a race condition may occur since there might be non-exclusive accesses to the shared object. Thus, F_1 and F_2 should be protected by the same lock.

Detection. For each block b of the program:
(1) Consider the set B of blocks that may be executed in parallel to b, i.e. $B := \{b' : mhp(b, b') = true\}$.
(2) For each block $b' \in B$, check if b and b' might access shared objects such that there could be read-write- or write-write-conflicts.
(3) If b and b' are protected by different locks or not protected at all, report the pair (b, b').

6.4 Multiple Stage Access

Consider the following method, which consists of two separately protected blocks:

```
void foo() {
  synchronized(o) { x = x * 2; }
  // ...
  synchronized(o) { x++; }
}
```

Problem. A method may consist of a sequence of operations which are protected separately. Sometimes the programmer wrongly assumes that this is safe, although these operations should be protected altogether. Otherwise there might be unprotected intermediate results, which can lead to a race condition.

Detection. For each method m of the program:
(1) Consider all synchronized blocks in m.
(2) If these blocks might share objects (i.e. if there are non-empty intersections of their variables' points-to sets), report m.

6.5 Synchronization Too General

The following class implements at least two `synchronized` methods that do not share any objects:

```
class SomeClass {
  //...
  synchronized foo() { x++; }
  synchronized bar() { y++; }
}
```

Problem. If two or more methods are protected by the same lock although they do not share any objects, this can make other threads unnecessarily block. To prevent inefficient synchronization, these methods should be protected by different locks.

Detection. For each class c of the program:
(1) Consider the set S of synchronized methods in c.
(2) Build a graph $G = (V, E)$. We define vertexes $V := S$ and edges $E \subset S \times S$, such that $(m_1, m_2) \in E$ if m_1 and m_2 might share objects (i.e. if the points-to sets of the variables accessed within m_1 and m_2 have non-empty intersections).
(3) If the graph is not connected (i.e. if there are at least two components), report c, S, and the found components.

7. EXPERIENCES

To implement and evaluate our approach, we developed a tool called *SyncChecker*, which is based on the Soot framework [17]. Soot reads Java source or class files and generates intermediate representations allowing program analysis and code optimizations. Our tool uses Jimple, a typed 3-address intermediate representation, to perform data flow analyses. Points-to information is computed by using SPARK [11], which is part of the Soot framework.

We applied SyncChecker to small test examples, an open source web server, and commercial software. We ran all analyses on a 1.66 GHz processor machine with 1 GB RAM. Our evaluation focuses on the number of false positives. Experiences will be discussed in this section.

Test example	Deadlocks	SyncChecker	JLint
TE-1	0	0	10
TE-2	1	1	10
TE-3	2	2	10
TE-4	1	1	10
TE-5	1	1	10
TE-6	0	0	10
TE-7	1	1	2
TE-8	0	1	2

Table 1: Comparison of SyncChecker and JLint

Classes	13
Thread classes	3
Methods	119
Fields	136
Synchronized blocks	5
Unsynchronized blocks	82
TLOC	1 435

Table 2: Properties of Tornado 0.2.0

7.1 Test Examples

We designed eight small test examples TE-1 to TE-8. All of them contain locking cycles, but not all of them result in deadlocks, because monitor names reference different objects or some methods are never executed in parallel. We evaluate the accuracy of our analysis, since we know the actual number and location of synchronization defects. In addition, we compared the results of our tool with JLint. All test examples have about 50 to 100 lines of code and took 1 to 3 seconds to be analyzed.

Table 1 shows the number of deadlock-related warnings issued by SyncChecker and JLint. Although JLint generally runs much faster than SyncChecker, it is less precise, producing numerous false positives, while our tool produces only one. This difference can be traced back to the impact of points-to and MHP information.

7.2 Tornado

Tornado [18] is an open source web server providing a full implementation of HTTP 1.1. It consists of listener and server threads and a thread manager. Table 2 shows some properties of Tornado 0.2.0.

Analyzing Tornado took about 20 seconds. SyncChecker reported three warnings: two "superfluous locks" and one "synchronization too general", which were easily confirmed when looking at the source code.

An interesting warning with respect to the impact of the MHP analysis refers to the method `removeThread()` in the `ServerPool` class. The warning says that this method (which is synchronized) probably does not need to be protected because it does not share any objects with another block possibly executed in parallel. When looking at the source code, the programmer wrote the following comment: *"This is synchronized so that we never try to kill the same thread twice – although that should never occur currently"*. This assumption was verified by SyncChecker.

7.3 TerminalX

Finally, we applied SyncChecker to a commercial software package for embedded systems, which was developed by Siemens. The package, which we call "TerminalX" in the following, is based on the Mobile Information Device Profile (MIDP) [13] of the Java 2 Micro Edition [8] and heavily uses multi-threading.

Classes	101
Runnable classes	12
Methods	1 104
Fields	819
Synchronized blocks	15
Unsynchronized blocks	409
TLOC	32 014

Table 3: Properties of TerminalX

Pattern	found	confirmed
Deadlock	0	0
Superfluous Lock	6	2
Missing or Wrong Lock	4	4
Multiple stage access	0	0
Synchronization Too General	2	2

Table 4: Warnings for TerminalX

TerminalX consists of a coprocessor component for managing the communication between a user and external applications (e.g. for reading sensor data). The communication is implemented as a dedicated connector component, which is serially connected to the coprocessor. For this software, some bugs were already known by previous code inspections, which allowed for examining the precision of our analysis. Some properties of TerminalX are shown in Table 3.

Analyzing TerminalX took about 2 minutes. The numbers of reported and confirmed warnings for this software package are shown in Table 4. Our tool emitted twelve warnings, of which six could be confirmed by the code reviewers. Two warnings had not been noticed but could be confirmed as well. Only four were false positives.

7.4 Summary and Discussion

We applied SyncChecker to small test examples, an open source web server, and a commercial software package. Our tool reported possible defects in all of them. Most of these warnings were correct.

Our evaluation focuses on the number of false positives and does not discuss escapes, i.e. defects that go undetected. For our approach, we consider the problem of false positives more important than escapes. Our tool does not report defects itself but code patterns which could produce undesirable runtime behavior. That is, if there exist code patterns corresponding to the definitions in Section 6, our tool will report all of them. Thus, there might be undetected defects, but no undetected patterns. To detect additional defects, it will be necessary to investigate and develop corresponding detection strategies for new code patterns as suggested in the next section.

8. CONCLUSIONS AND FUTURE WORK

We presented a new method for detecting code patterns that may result in deadlocks or race conditions or indicate inefficient synchronization in Java programs. Our approach is based on the Soot framework and uses static analysis techniques combined with points-to and MHP analysis. The

combined analysis extracts precise information about the program in form of constraints and blocks. This information allows for searching for code patterns that might result in runtime errors such as deadlocks or data races. While other approaches suffer from a high number of false positives, the hit rate of our tool is excellent. We applied it to small test examples, an open source web server, and finally to commercial software and found synchronization defects in all of them. However, there are some open questions and aspects for future work as described in the following.

Undetected defects and new code patterns. Since our tool does not report defects itself but code patterns which might result in those, there still might be undetected defects. To detect additional types of defects, new code patterns have to be developed as well as corresponding detection strategies. An example are deadlocks caused by the improper use of `wait()` or `notify()` statements. More patterns could be developed and examined based on Lea's design principles for concurrent programming [10] or experiences from software inspection, for example.

Ranking errors. Our tool only supports a rather simple ranking of the detected errors. The development and evaluation of more complex metrics could be of special interest. Especially for large systems, it is necessary to bring the most severe or most likely problems to the programmer's attention first.

Performance studies. Generally, the runtime of an analysis depends on its accuracy. Our analysis is rather precise since we include points-to and MHP information. It would be interesting to examine the runtime behavior for analyzing larger systems. Since our tool uses the flexible points-to analysis framework SPARK, studies could try to find a balance between accuracy and runtime performance by means of modifying the parameters of the points-to analysis.

Visualization. So far, we provide a basic, yet functional, implementation with textured output. Our tool could be integrated into a development environments such as Eclipse [2]. Problematic code could be highlighted accordingly.

9. ACKNOWLEDGMENTS

This work was funded by Siemens Corporate Technology, Munich, Germany. We are especially grateful for the comments and help of Dr. Marcus Oestreicher. We also thank the anonymous reviewers of this paper as well as the Code Quality Management Group of Siemens for providing their code.

10. REFERENCES

[1] C. Artho and A. Biere. Applying static analysis to large-scale, multi-threaded Java programs. In *ASWEC '01: Proceedings of the 13th Australian Conference on Software Engineering*, pages 68–75, Washington, DC, USA, 2001. IEEE Computer Society.

[2] Eclipse. `http://www.eclipse.org/`.

[3] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for testing multi-threaded Java programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003.

[4] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. *SIGOPS Oper. Syst. Rev.*, 37(5):237–252, 2003.

[5] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 286.2, Washington, DC, USA, 2003. IEEE Computer Society.

[6] Findbugs. `http://findbugs.sourceforge.net/`.

[7] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM.

[8] The Java ME Platform. `http://java.sun.com/javame/`.

[9] Java Technology. `http://java.sun.com/`.

[10] D. Lea. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns.* Addison-Wesley, Boston, MA, USA, 2000.

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

[12] L. Li and C. Verbrugge. A practical MHP information analysis for concurrent Java programs. In *LCPC*, pages 194–208, 2004.

[13] Mobile Information Device Profile (MIDP). `http://java.sun.com/products/midp/`.

[14] G. Naumovich, G. S. Avrunin, and L. A. Clarke. An efficient algorithm for computing MHP information for concurrent Java programs. In *ESEC / SIGSOFT FSE*, pages 338–354, 1999.

[15] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[16] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multi-threaded programs. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 27–37, New York, NY, USA, 1997. ACM.

[17] Soot: a Java Optimization Framework. `http://www.sable.mcgill.ca/soot/`.

[18] Tornado. `http://tornado.sourceforge.net/`.

[19] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *ASE '00: Proceedings of the 15th IEEE international conference on Automated software engineering*, page 3, Washington, DC, USA, 2000. IEEE Computer Society.

[20] C. von Praun. *Detecting Synchronization Defects in Multi-Threaded Object-Oriented Programs.* PhD thesis, Swiss Federal Institute of Technology, Zurich, Switzerland, 2004.