Finding faults in multi-threaded programs

Cyrille Artho

03/15/2001

Abstract

Multi-threaded programming creates the fundamental problem that the execution of a program is no longer deterministic, because the thread schedule is not controlled by the application. This causes traditional testing methods to be rather ineffective. Trilogy, producing many multi-threaded server programs, also has to deal with the limitations of regression testing. New approaches to this problem – static and extended dynamic checking – promise to ameliorate the situation. Many tools are in development that try to find faults in multi-threaded programs in new ways.

The first part of this report describes a detailed evaluation of a wide variety of dynamic and static checkers. That comparison always had the applicability to industrial software in mind. While none of the checking tools was a clear winner, certain tools are more useful in practice than others.

Because simple cases are the most common ones in practice, the decision was made to extend Jlint, a simple, fast static Java program checker. The new Jlint can now also check for deadlocks in `synchronized` blocks in Java, which results in improved fault-finding capabilities. The extensions and their usefulness in an industrial environment are described in the second part of the report. Jlint has been applied to many core packages of Trilogy, and also a few other software packages, and shown various degrees of success.

Acknowledgements

I would like to thank Prof. Armin Biere from the Swiss Federal Institute of Technology for supervising my thesis and taking the risk of working across 5000 miles and seven hours time difference, and for taking the time to proofread the report meticulously and having the patience for revising the definitions several times until they were right.

I would also like to thank Trilogy for taking the challenge of hosting their first thesis, and in particular Bernhard Seybold and Runako Godfrey, who supervised my work. Especially Bernhard Seybold gave me much valuable advice and took the time to run different versions of Jlint on his program, which gave me feedback early on. He also proofread most of the report, and his suggestions helped to improve it.

Many more people helped me at Trilogy, so I cannot list them all here. Special thanks go to Razvan Surdulescu, Dave Griffith and Ruwei Hu for verifying my annotated Jlint warnings.

More thanks go to Konstantin Knizhnik, the author of Jlint, who helped me to understand his code and the ideas behind it, K. Rustan M. Leino from Compaq for answering some ESC/Java related questions, Derek L. Bruening for supplying me the Rivet executables, Moonjoo Kim from the University of Pennsylvania for answering my MaC inquiries, Klaus Havelund from the NASA Ames for his suggestions and his efforts to make JPF(2) available (although it did not work out in the end), and Christoph von Praun from the Swiss Federal Institute of Technology for his feedback about my analysis of his data warehousing package.

Contents

1 Introduction **1**
- 1.1 Multi-threading problems . . . 1
- 1.2 Existing checkers . . . 3
- 1.3 Multi-threaded programming in Java . . . 5
- 1.4 Comparison of Java program checkers . . . 6
- 1.5 Extension of a Java program checker . . . 6
- 1.6 Structure of this report . . . 7

2 Existing work **8**
- 2.1 Dynamic checking . . . 8
- 2.2 Static checking . . . 9
- 2.3 Interface specification . . . 10
- 2.4 Summary . . . 10

3 Evaluation stage **13**
- 3.1 Evaluation criteria . . . 13
- 3.2 Selection of examples . . . 14
- 3.3 Evaluation process . . . 16
- 3.4 Tool evaluation . . . 20
- 3.5 Statistical analysis . . . 28
- 3.6 Comparison of the results . . . 32
- 3.7 Summary . . . 34

4 Jlint extensions **35**
- 4.1 How Jlint works . . . 35
- 4.2 Goals . . . 35
- 4.3 Implementation of extensions . . . 36
- 4.4 Code changes . . . 40
- 4.5 Problems encountered . . . 42
- 4.6 Application of the new Jlint . . . 47
- 4.7 Summary . . . 49

5 Discussion **51**
- 5.1 State of the art . . . 51
- 5.2 Capabilities of Jlint . . . 53
- 5.3 Usage of static analyzers in software development . . . 53
- 5.4 Summary . . . 54

6 Future work **55**
6.1 Future Jlint extensions 55
6.2 Design of a compiler-based analyzer 56
6.3 Future directions for formal analysis 59
6.4 Summary 61

7 Conclusions **62**

A Source code analysis **63**
A.1 Analysis tools 63
A.2 Trilogy's source code 64
A.3 Built-in Java packages 68
A.4 Other packages 69
A.5 Summary 71

B Existing tools **74**
B.1 Dynamic checkers 74
B.2 The Spin model checker 77
B.3 Static checkers 77
B.4 Other tools 84

C Multi-threading in Java **85**
C.1 Threads 85
C.2 Thread synchronization 85
C.3 Summary 88

D Example listings **89**
D.1 Selected programs 89

E Test results **101**
E.1 Benchmark 101
E.2 Program checker results 102

F Results of new Jlint **120**
F.1 Extra Jlint examples 120
F.2 Trilogy's source code 124
F.3 Concurrency package 128
F.4 ETHZ data warehousing tool 129

Bibliography **130**

Index **134**

List of Figures

1.1 Illustrating the scheduling problem. 1
1.2 A simple deadlock example. 2
1.3 Separating model checkers and theorem provers. 4

3.1 Test results for the 15 given examples. 16
3.2 Overall usage of `synchronized` statements in Trilogy's code. . 30
3.3 Total usage of `synchronized` statements. 32

4.1 Call graph extension for `synchronized` blocks. 39
4.2 Call graph extension for Listing F.2. 40
4.3 Constant pool entries for a field. 44
4.4 Call graph extension for `synchronized` blocks. 46
4.5 Test results for the 15 given examples, including the new Jlint. . 47

6.1 The problem of propagating the context. 59

A.1 Breakdown of the usage of `synchronized` statements. 67
A.2 Statistics of the usage of `synchronized` statements. 67
A.3 Types of variables used in `synchronized` blocks. 67
A.4 Overall usage of `synchronized` statements in Trilogy's code. . 68
A.5 Statistics of the usage of `synchronized` statements. 69
A.6 Types of variables used in `synchronized` blocks. 70
A.7 Overall usage of `synchronized` statements in the Java packages. 70
A.8 Statistics of the usage of `synchronized` statements. 71
A.9 Overall usage of `synchronized` statements in `javax`. 71
A.10 Statistics of the usage of `synchronized` statements. 72
A.11 Statistics of the usage of `synchronized` statements. 72
A.12 Overall usage of `synchronized` statements in the Concurrency package. 73
A.13 Overall usage of `synchronized` statements in the ETHZ package. 73
A.14 Total usage of `synchronized` statements. 73

C.1 `Synchronized(this)` vs `synchronized` methods. 86
C.2 The deadlock in example D.1. 87

E.1 Screenshot of warning for Deadlock example. 103
E.2 Graph for DeadlockWait2 produced by VisualThreads. 107
E.3 Graph for BufferNotify produced by VisualThreads 115
E.4 Alternating thread states in VisualThreads. 117

List of Tables

2.1 Overview of existing tools. 11

3.1 Overview about the tested tools. 18
3.2 Annotations required for code examples. 25
3.3 Overall usage of `synchronized` statements in Trilogy's code. . 30

4.1 Growth of Jlint code 41

A.1 Overview about each package. 63
A.2 Overview of Trilogy's source code. 65
A.3 Per module usage of `synchronized(non-this)` blocks . . . 68
A.4 Overview of the source code of the built-in Java packages. . . . 69
A.5 Per module usage of `synchronized(non-this)` blocks . . . 70
A.6 Total usage of `synchronized` statements. 72

F.1 Analysis of Jlint warnings for MCC Core. 125
F.2 Analysis of Jlint warnings for Cerium. 125
F.3 Analysis of Jlint warnings for the Java backbone. 126
F.4 Analysis of Jlint warnings for trilogyice. 127
F.5 Summary of Jlint's warnings in Trilogy's code 127
F.6 Analysis of Jlint warnings for the ETH data warehousing tool. . 129

Table of Listings

4.1 Lock variable analysis example. 37
4.2 Code snippet from Jlint: getting the field context. 44
6.1 Extra `monitorexit` operations inserted by the Java compiler. . 58
D.1 Deadlock: `run` method of two competing threads. 89
D.2 Deadlock2: Locking scheme from Deadlock on a method level. . 90
D.3 DeadlockWait: `run` method of two competing threads. 90
D.4 DeadlockWait2: method `foo` of classLock B. 91
D.5 Deadlock3: `run` method of three competing threads. 91
D.6 Race condition: A lock is released in between a calculation. . . . 92
D.7 Jlint example: Loop in lock graph. 93
D.8 ESC/Java example: Pathological case with two locks 94
D.9 Shared bounded buffer (correct version). 95
D.10 Race condition: condition of `wait` is not checked again. 96
D.11 Condition deadlock: `notify` instead of `notifyAll` is used. . . 96
D.12 Buffer implementation using semaphores. 97
D.13 Semaphore implementation. 97
D.14 Naïve implementation of the Dining Philosophers problem. . . 98
D.15 Solution 1 for the Dining Philosophers problem. 99
D.16 Solution 2 for the Dining Philosophers problem. 100
F.1 Two faults regarding a `wait` call. 120
F.2 Deadlock scenario among two methods. 121
F.3 More complicated version of the same deadlock. 122
F.4 Assigning a new value to a lock variable. 123

Chapter 1

Introduction

In the last few years, multi-threaded software has become increasingly widespread. Especially for large servers, multi-threaded programs have advantages over multi-process programs: Threads are computationally less expensive to create than processes, and share their address space among each other. Java makes it easy to write multi-threaded programs; despite this, writing correct multi-threaded software is still very hard.

1.1 Multi-threading problems

Software should be tested thoroughly. Because it is not possible to prove the correctness of a program, one tries to create situations that discover a fault in the software by choosing a representative set of inputs *(test cases)*. A *fault* is an incorrect implementation, due to a human *error*. A fault can eventually lead to a *failure* during program execution [56]. Because finding a fault requires a test case leading to a failure, this task can be very hard. Usually, test cases are written to model *known* or *anticipated* failures, but of course no test cases exist for unknown ones.

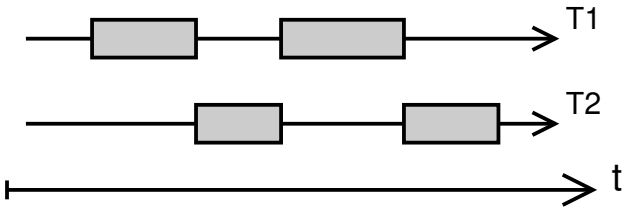

Figure 1.1: Illustrating the scheduling problem.

Multi-threaded programming introduces an entirely new set of difficulties. Unlike in a single-threaded program, the execution of multi-threaded software can be *non-deterministic:* the same input may lead to different outputs. This is because the *scheduling* of the different threads cannot be influenced by the program. If a part of the program depends on several threads executing in a certain order, it is not *thread-safe:* it cannot be guaranteed that the output of the

program is the same, regardless of the scheduling outcome. Figure 1.1 illustrates this problem: Only one thread can run at a time. Neither the order in which threads execute, nor the exact size of the time slots (the gray boxes) they get is known. This is why there is no scale on the time axis. Several typical multi-threading problems exist:

Race condition: several threads access the same resource simultaneously.

Deadlock: threads *starve* each other by holding (and not relinquishing) resources that the other thread needs to continue.

Livelock: in the resource sharing protocol between the threads, an endless cycle without progress occurs.

When investigating multi-threading problems, the checkers investigate the locking behavior of a program. A *lock* controls access to a shared resource: only a thread holding the lock is allowed to access that resource. In many implementations, only one thread is allowed to hold the lock at a time: such a lock is *exclusive.*

For ensuring the absence of a **race condition,** a checker examines the *lock set* L. This is the set of locks held at a certain time, by each thread when accessing a field. A checker has to ensure that a field f is 1) only read when a thread holds at least one lock in L_f and 2) only written when a thread holds *all* locks in L_f [36].

Thread 1

```
synchronized(A) {
  synchronized(B) { }
}
```

Thread 2

```
synchronized(B) {
  synchronized(C) { }
}
```

Thread 3

```
synchronized(C) {
  synchronized(A) { }
}
```

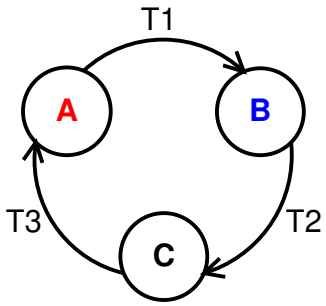

Figure 1.2: A simple deadlock example.

For proving the absence of a **deadlock,** the common approach is to examine the *lock graph,* which shows for each thread the order in which it acquires locks. Figure 1.2 depicts a constellation of three threads competing for three resources (with incomplete Java source code). If all three threads hold one lock each, none of them can continue because the second lock they need is already taken. It can be shown that the *absence* of a loop on the lock graph guarantees the absence of a deadlock.

Livelocks are more difficult to detect. In particular, the entire information about a program state can be very large, and multiplied with the number of

states, prohibitively large to store and compare. Therefore, simplifications have to be made for the program states. As of today, some tools are already capable of assuring a high likelihood for the absence of a livelock in a program.

The goal of this work was to evaluate existing program checkers for multi-threaded programs, and decide which one is best applicable to large scale software, such as Trilogy's.

The next sections briefly describe the two major approaches and what tools are available now for checking Java programs. Based on the outcome of the analysis, one checker was chosen and extended before it was applied to Trilogy's code base.

1.2 Existing checkers

Until a few years ago, the only way to test a multi-threaded program was to run it long enough, hoping that eventually enough scheduling combinations would come into play to uncover most faults. Verifying the properties of a program at run time is also called *dynamic checking*.

Recently, an approach successfully employed in hardware verification has been applied to software: *static checking*. A static checker does not run the program, but it analyzes the structure of the program. This thesis has investigated both possibilities thoroughly. As will be shown, no one is clearly superior to another; instead, the two approaches are complimentary.

1.2.1 Dynamic checkers

Description

Verifying program properties at run time is the traditional approach. *Assertions* are easily monitored at run time; *debuggers* can help automating the tracking of the program state. More advanced dynamic checkers monitor any memory accesses of a program, in order to trap array accesses beyond the bounds of an array and heap accesses outside the reserved range. Such tools are common development tools today. However, no dynamic checker can systematically cover all possible inputs, because the input space is exponential to the length of the input.

The standard tools do not solve multi-threading problems. In particular, they are still vulnerable to the fact that the program execution is no longer deterministic. Some novel approaches try to keep track of the program's *history*, its previous execution stages, and deduct information about other possible outcomes (results of different schedules) from that. In particular, tracking the history of the lock graph has proved to be a reliable guide in finding multi-threading problems.

Advantages

Dynamic checkers are usually easier to use, because the concepts are established and well-known. Usually such checkers do not require any extra modeling information; they only need to know what properties need verification. Moreover, monitoring tools have access to the entire program state at any point

of execution, leaving no sources of doubt when it comes to the values of each variable.

Problems

Certain faults cannot be detected dynamically unless the thread scheduler exactly reproduces the scenario that leads to it. This problem is partially alleviated, but not solved, by keeping track of the history of the state space. Moreover, the classical problem of finding the right test cases is also far from trivial, and limits the abilities of dynamic checkers further. Finally, writing test cases is a time consuming and tedious task, which most developers would gladly avoid.

1.2.2 Static checkers

Description

Static checkers have in common that they build a simplified representation of the program which they check against given properties. The techniques commonly employed are model checking and theorem proving. *Model checkers* operate directly on that model of the program (such as a call graph or a finite state machine). Such a model may represent the *control flow* (flow of execution) or *data flow* (changes in the variables) of a program and is commonly expressed in computational tree logic (CTL) or linear temporal logic (LTL). *Theorem provers,* however, translate the program into logic formulas (in first order or second order logic). These formulas are then processed by a theorem prover. Figure 1.3 shows the distinction between model checking and theorem proving. It should be noted that the two approaches are often combined, so the boundaries are blurring.

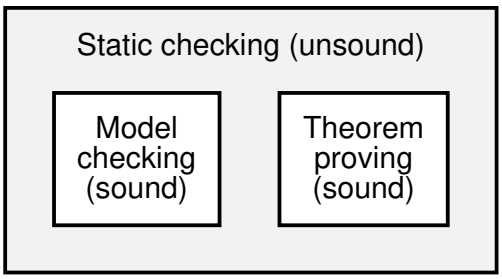

Figure 1.3: Separating model checkers and theorem provers.

Originally, one came from a manually written *formal specification,* where the goal was to prove the *correctness* of that specification. Proofing the correctness of non-trivial programs is impossible in general [57]. Such a proof is almost always *incomplete:* there are always cases where a prover is unable to conclude that an error will never occur. Therefore, a prover is bound to issue *spurious warnings* in such cases [36]. This made that approach very problematic: The specification languages were difficult to learn and to master, creating in themselves a source of errors; and even a successful proof could not guarantee the

correctness. Moreover, errors could be made in the *implementation* of that specification. In the last decade, that approach no longer made major progress.

A *new approach* was to create the model automatically from the program, with little or no human intervention. During this abstraction, information about the program is lost. A *sound* checking (also see figure 1.3), which catches all faults, was no longer feasible. It would constrain static checkers too much [36]. As such, a sound checker could be written, although at the cost of potentially many spurious warnings.

The *applicability* of a checker will, in this report, refer to what multi-threading mechanisms (in the implementation language) and problems the checker can be applied to. This is a subjective measure, because a checker may not cover certain mechanisms very well. In particular, a trivial checker that issues a warning for any statement would be applicable to any kind of problem, without being of any use.

Advantages

Static checkers have the advantage that they work on a more abstract (and thus generic) level than dynamic checkers. In particular, they are independent on both the input and the thread schedule, and therefore can verify program properties for *all* inputs and schedule outcomes.

Static checking also works well on a *unit level*, where only one entity of a larger software package is checked. This allows an application of this method early in the development cycle, even before a working program exists.

Problems

The key problem is that the actual values of variables are usually not fully known at compile-time. In particular, the *aliasing problem* (knowing equalities between two object references with different names) is very hard to solve, in some cases even impossible. The potentially infinite complexity of data structures (such as linked lists) and possibly never terminating loops are the reason for this.

Quite often, static checkers are simply limited by the amount of *context* they can deduce from source or object code, because they have a far more limited capability in deducing information than the human mind. Therefore, such provers are often aided by *annotations* in the source code, which express additional information beyond the given programming language constructs.

Finally, it is hard to assure that a violation of a modeled property corresponds to a fault in the software. In particular, finding a counter-example requires tracing all abstraction steps back to the original software. One possible approach is verify counter-examples dynamically [34].

1.3 Multi-threaded programming in Java

Java was one of the first widespread programming languages that introduced multi-threading as a *language concept*. It has a special class for controlling threads (most importantly, the `Runnable` interface and the `Thread` class) as well as special keywords and methods for communication between objects

(`synchronized`, `wait`, `notify`). Before multi-threading was part of programming languages, it usually could only be used via libraries (e.g. `p_threads` in C or C++).

The key feature in Java, which this thesis is focusing on, are `synchronized` statements. They always cause the thread to obtain a lock (or wait until the lock is available). Therefore, the correct and sufficient usage of these synchronization statements is the key to avoiding deadlocks and race conditions. Also see Appendix C on page 85.

1.4 Comparison of Java program checkers

The previous sections gave an introduction to the problem. The goal of this work was to find the most practical solution for finding faults in Trilogy's software. In a first step, fourteen static and dynamic program checkers were investigated. Some of these checkers do not work on Java programs, or are not yet finished or publicly available. Therefore, the selection was narrowed down to five checkers. In a second phase, each checker was tested on fifteen test examples. These examples represented small, well-known problems and typical errors that can be made when writing multi-threaded programs.

In order to judge the relevance of the fifteen test cases, a statistical analysis of a large body of code provided a solid foundation of the frequency of different problems. In particular, all the Java packages and all core packages of Trilogy were analyzed, in conjunction with a special concurrency package and a data warehousing algorithm [23, 24]. The outcome of the example tests, weighed by the frequency of problems, can be summarized as follows:

MaC [11]: An elegant framework for monitoring programs, but it does not support multi-threading yet; work is in progress in that area.

Rivet [12]: A special virtual machine that tries all possible thread schedules and therefore finds any fault for a given input, although at a prohibitively high overhead.

VisualThreads [14]: By keeping track of the locking history, this program can find deadlocks even if they do not occur in a particular program schedule. The checker is specialized on C programs, not Java bytecode.

ESC/Java [3]: The first available theorem prover for Java programs; very powerful, but still rather limited in the area of multi-threading problems.

Jlint [8]: A simple and very fast model checker that can successfully detect simple faults. Its original version lacked some important features, though.

1.5 Extension of a Java program checker

Because of the limitations of currently available dynamic checkers for Java, and because of Jlint's astounding performance, the decision was made to extend Jlint's applicability to those `synchronized` blocks in Java where no global data flow analysis was needed. Despite existing limitations in Jlint, the desired extensions could all be implemented. However, it was seen that a good static

checker needs to have a clean architecture as much as good algorithms. Based on the insights gained while extending the checker, guidelines for writing a new verifier have been created.

Applying Jlint to Trilogy's code and other packages still resulted in a very high number of warnings. Selectively turning certain warnings off made the output manageable. Many warnings were confirmed to be relevant, and while most of them were false positives, at least 12 of them lead to extra comments or even code changes ("bug fixes").

It was seen that certain checks in Jlint still need refinement, in order to reflect certain common scenarios in multi-threaded programming, such as shared-read variables. Despite that, even in its current state, the simple checker Jlint can already be of great use in software development, as a tool to point out potential trouble spots.

1.6 Structure of this report

Chapter 2 describes existing tools in more detail, and why the five selected tools were chosen for the tests. Chapter 3 gives details of the evaluation of the selected tools, and the results found. The extensions made for Jlint, and their implementation, are described in chapter 4. The next chapter discusses the outcomes of the research and experiments made for this report. Possible directions for future work in both the area of static and dynamic checking are outlined in chapter 6. Chapter 7 concludes this report.

Chapter 2

Existing work

This chapter describes tools that tackle the problem shown in Chapter 1. Some of these programs are still under development; others are either publicly available or proprietary.

Dynamic checkers are listed first, followed by static checkers, in order to facilitate a comparison. For each category, there are tools that check a given set of faults, and those that allow templates of rules or state sequences, which make the tool much more flexible. In the section about static checkers, Spin is presented first. Spin is a model checker that operates on its own input language (which is quite similar to a programming language). It does not directly solve the problem but serves as a back-end for many of the tools described thereafter. At the end, a table summarizes the crucial aspects of these tools, allowing an easy comparison between them. A more detailed description of each tool can be found in Appendix B on page 74.

2.1 Dynamic checking

MaC (Monitoring and Checking) is a framework that combines low-level monitoring with high-level requirement specifications. It is being developed at the University of Pennsylvania. So far, MaC can successfully instrument and verify single-threaded programs, but it has no support for multithreading yet.

Rivet is a special virtual machine for Java, which systematically tries every thread schedule that is relevant for an exhaustive examination of the program behavior. Despite clever optimizations, the run time overhead is still very high, and many practical problems have forced the Software Design Group at the MIT to give up on that project.

Verisoft, by Patrice Godefroid from Lucent Technologies, also systematically explores the state space (including thread interleavings) of a program. By using a new search algorithm, it can explore the program behavior without storing its state space. It supports a check against deadlocks, lifelocks, assertion violations, and other properties. A checker for C programs is available for research; a Java checker is under construction.

VisualThreads part of the development tools of Compaq's Tru64 Unix. It monitors the locking policy of a program and can detect race conditions and deadlocks. Because the monitoring takes action at the POSIX API level, this tool is rather ineffective for Java programs; it works well for C and C++ programs.

2.2 Static checking

This section describes static checkers, both model checkers and theorem provers, in alphabetical order. Spin is presented first because it is often a part of another tool.

Spin is a static model checker and serves as the back-end for other static checkers, such as Bandera, FeaVer or JPF. It takes system specifications in a special process meta language (Promela). Gerard J. Holzmann started the development of this tool in 1980. It is available as Open Source software.

Bandera from the Kansas State University tries to bridge the gap between source code and an abstract representation of a program. Using annotated source code, Bandera tries to simplify the program by *slicing* (omitting properties that are not relevant to the analysis) and *abstraction* (reduction of the state space of variables). The simplified program is then processed by Spin. Spin's output is verified by a *counter-example generator,* which checks Spin's result for validity in the real program.

ESC/Java (Extended Static Checker for Java) from Compaq statically checks a program for common errors, such as `null` references, array bounds errors, or potential race conditions. It is usually used with annotated source code or bytecode. Its compiler generates *background predicates* which are then relayed to a theorem prover. There is no real support yet for counter-examples. The checker is freely available for research purposes.

FeaVer verifies program properties that are extracted from a special *test harness,* a structured test program. Its ultimate goal is to do this fully automatically. Right now, the user still has to provide some extra information in separate files, and the tool is restricted to event-driven programs. Even at that stage, it has proved very useful at Bell Labs, where it is being developed by Gerard J. Holzmann.

Flavers is one of three static checkers developed by the Computer Science department at the University of Massachusetts Amherst. It combines data and control flow analysis and allows checking a software implementation against formalized design requirements. A commercial version (for C++ programs) and a research version (for Ada programs) exist; a Java checker is under development.

Jlint has been developed by Konstantin Knizhnik at the Moscow State University. By performing a global control flow and a local data flow analysis, it can verify a lot of properties in Java bytecode. It is most successful in `null` pointer and a few specialized checks, but also allows checks for deadlocks and race conditions. Jlint is freely available.

JPF (Java PathFinder), developed by NASA, analyzes invariants and deadlocks statically. The original version worked on Java source code, where supporting certain language features, such as arrays or floating point numbers, proved rather difficult. The newer version works on bytecode. JPF uses Spin as its back-end. NASA currently has no plans to release JPF.

LockLint, by Sun Microsystems, detects race conditions and deadlocks in POSIX C programs. It allows interactive or automated queries. Annotations in C sources are not required, but recommended. LockLint is commercially available as part of the Forte development suite.

MC (Meta-level compilation) from Stanford University builds compiler-specific extensions to check and optimize code. A set of simple rules is used to check large packages for violations of certain consistency patterns. MC has been successfully used for checking the Linux and BSD kernels, but it has not been released to the public so far.

SLAM is a large project at Microsoft. Its focus is the automatic abstraction of source code. A new formal model for multi-threaded programs, an extended state machine, has been developed, which is verified by a model checker for Boolean programs. The variables in such programs only have three states: true, false, or unknown. Certain tools should be released to the public in the near future.

2.3 Interface specification

JML/LOOP, by the Iowa State University and the Computer Science Department in Nijmegen (Holland), allows the specification of module properties. These interface specifications can be checked against implementations, which allows a safe "design by contract" in libraries [45]. Concurrency extensions are currently being explored. JML is available under the GPL.

2.4 Summary

This chapter provided an overview about a variety of methods that are currently used for finding faults in multi-threaded programs. Some of these methods are still very experimental; others only work on certain programming languages. Many tools are not available outside the research group or company where it is being developed. Table 2.1, which has been assembled during the analysis of the tools, summarizes this.

For Trilogy, it is of preferable to have a checker that operates on Java programs, because only a small fraction of their source is not in Java. However, in the first evaluation stage, a C/C++ based tool can still provide valuable insights about how other tools could be improved, or in which direction the development of a new tool should go.

In each major category, at least one tool is available. From those tools, JML/LOOP was dropped from the selection, because it does not have any temporal extensions yet, and the main goal of JML is safe "design by contract",

Category	Tool	Detects [violation of]	Static or dynamic?	User-def. model or template?	Req. source?	Java version	Non-Java version	Availability
Static checkers	Bandera	Low-level properties	Static	Yes	Yes	Beta (v0.1)	-	Since March 8, 2001
	ESC/Java	Deadlocks, race cond., other faults	Static	No	No	**Released, stable**	Modula-3	**Binary version for research**
	FeaVer	Test cases	Static	Yes	Yes	-	C: Early prototype	2002?
	Flavers	High-level properties	Static	Yes	Yes	Prototype	Ada/C++ stable	Ada: available upon request?
	Jlint	Deadlocks, race cond. other faults	Static	No	No	**Stable**	-	**Free (GPL)**
	JPF	Assertions	Static	No	Yes	-	Stable	Undecided
	LockLint	Deadlocks, race cond.	Static	No	Yes	-	C: Stable	Part of Sun's Forte for C
	MC	High-level properties	Static	Yes	Yes	-	C/C++ Usable	Not available – possibly later
	SLAM	Assertions	Static	Probably	Yes	-	C: In deve-	Not yet available
Dynamic checkers	**MaC**	High-level properties	Dynamic	Yes	Yes	**Beta (v0.99)**	-	**Binary version for research**
	Rivet	Assertions	Dynamic	No	No	**Discontinued**	-	**available for research**
	Verisoft	Assertions	Dynamic	No	Yes	-	C/C++: Stable	Binary version for research
	Visual Threads	Misc. concurrency errors	Dynamic	No	No	**Stable**	C/C++: Stable	**Part of Alpha Unix development tools**
Interface specification	JML/ LOOP	Incorrect interface implementations	Both	Yes	Yes	-	Partial release.	Free (GPL)

Table 2.1: Overview of existing tools.

which is not an important goal for Trilogy, since all source code of the internal software is available within Trilogy. LockLint was not chosen because Jlint is very similar while being Open Source and Java based.

In table 2.1, the remaining selection of available tools is printed in bold. It should be noted that no static checker that works on high-level templates (such as Flavers, MC, and to some extent FeaVer) was available for evaluation. If none of the given tools had worked satisfactorily, this approach would have been considered as an alternative.

Chapter 3

Evaluation stage

This chapter describes the evaluation of selected program checkers. After considering their availability and applicability to Java programs (as opposed to Ada or C/C++ programs), only five checkers remained:

1. MaC: a dynamic checker verifying high level properties.
2. Rivet: a systematic thread scheduler for exhaustive testing.
3. VisualThreads: a development tool that keeps track of POSIX thread commands.
4. ESC/Java: a theorem-prover based checker by Compaq.
5. Jlint (old version 1.11): a simple, fast checker performing control flow analysis.

In a first phase, each tool was applied to a small set of test examples. The goal was to determine the capabilities of the tools. During the second phase, a statistical analysis of nearly a million lines of code was performed. The aim was to estimate which tools would be the best for application to large scale software packages.

3.1 Evaluation criteria

For the evaluation, the following questions were relevant:

1. How effective is the approach at finding faults?
    - Can a tool give a guarantee for the correctness of a certain property that has to be verified?
    - What kinds of errors are found? Does the checker allow for templates or model specifications to extend its functionality? Does it focus on multi-threading problems only or does its scope go beyond that?
    - How many actual errors are found, and how many spurious warnings (false positives) are reported?

- What is the running time of such a tool? Can it be applied to a large code base, such as Trilogy's?

2. How practical is a tool to use?

   - Does a tool allow a template specification that can be applied to many programs?
   - Does it require the source code, or only compiled versions? Does it require changes (annotations) in the actual source code?
   - What knowledge does a tool require (e.g. formal languages, temporal logic)?
   - How big is the annotation overhead in real-world programs? Does it allow a selective test for certain faults?
   - Is it suitable for being used in conjunction with a compiler, or as a stage prior to regression testing?

Before trying to judge the applicability of each tool to larger programs, it first had to be tested against well-known test examples. These would also show major differences between the tools and give directions for the statistical analysis. Running all tools against the full code base that was finally covered would have required too much time, since some tools require a major amount of work (for the annotations) or time (for dynamic testing).

3.2 Selection of examples

3.2.1 Measuring the complexity of examples

Measuring software complexity is a science of its own. Numerous *software metrics* exist, each one trying to capture a certain aspect of a program's size or complexity (or both). For a comparison of the test examples, the following metrics are suitable:

Metric	Explanation
Non-comment lines of code	Size of program, influences run time of parser.
McCabe's cyclomatic number	Number of independent paths (and decisions).
Number of threads	Heavily influences the size of a model and also the running time of dynamic checkers.
Number of locks	Number of synchronized methods and blocks.

Counting the lines of code is the simplest metric, and depending on the algorithmic complexity of the code and the coding style, it can be highly ambiguous (especially for generated code). Nevertheless, on a large scale, it provides a rough measure of the program size.

McCabe's metric [52] is one of the oldest metrics in existence. It measures the number of decisions in a program (i.e. `if`, `while` and `for` statements). It gives a good measure of the control flow complexity, but only allows comparisons of programs with a similar data structure complexity [55, pp. 320–21].

There are no established multi-threading metrics yet. Counting the number of threads and locks yields a result that is highly correlated with the running

time of program checkers. In particular, Rivet's performance is doubly exponential in the number of states and threads a program can have.

Not all tools examine the behavior of each thread (as opposed to the behavior of *any* thread), so using the number of threads as a metric is problematic in that context. The cyclomatic complexity and the number of locks does not seems to influence a static checker much, if one focuses on multi-threading issues. Moreover, the execution times of the different tools varied so much that a comparative benchmarking, based on these metrics, did not make much sense. See Appendix E.1 for more information. For Jlint, the execution times were always so short that they were not an issue (in general, if the files were already cached and the output was redirected to a file, Jlint requires less than one second even for large packages).

The examples described in this chapter were not chosen based on their values with certain metrics, but to exhibit specific problems in multi-threaded programming. The first few examples all show certain faults; the last ones (shared buffer and Dining Philosophers) show several (correct or flawed) implementations of a more complex algorithm. These examples should provide a much better test of the capabilities of each checker.

3.2.2 Selecting examples

When selecting examples, it was important to keep them small and relatively simple. Besides being easier to understand and more instructive, they are also easier to verify manually. After all, the correctness of the checkers should not be misjudged by flawed implementations that are considered correct.

Moreover, the examples should be simple enough such that all checkers can be applied to them; this would probably not have been the case for examples that require external modules (such as data base wrappers) to work. Nevertheless, the locking schemes and faults displayed by the examples should reflect properties of larger, real life programs. A description of these example programs can be found in Appendix D on page 89. The deadlock examples and the bounded buffer implementations have been taken from the Rivet test suite [29] or are modifications of these examples. Two implementations of the "Dining Philosophers" have been taken from [22], while the "host" variant has been described in [46].

Because it is very hard, even for someone who has been working with large amounts of code, to judge the applicability of such examples objectively, large software packages were analyzed in order to check the relevance of these examples. Most of the analyzed packages were taken from Trilogy's software or the core Java packages which are part of Sun's JRE 1.3. A data warehousing tool and a concurrency framework were analyzed as well ([23, 24]).

3.2.3 Overview of examples

A detailed listing of all 15 examples can be found in Appendix D. Three major categories of examples were used:

1. Six simple **deadlocks** using incorrect locking orders, or exhibiting problems with `wait` and `notify`: The first five examples (D.1 to D.5) belong into this category. The Jlint example (D.7) can also be counted towards

it; it differs slightly from the rest because it exhibits a deadlock between method calls across different classes.

2. A subtle **race condition** due to incomplete locking, as shown in SplitSync (example D.6).

3. Eight **complex locking schemes**, such as the ones in the shared buffer and Dining Philosophers problems. The ESC/Java example can also be counted towards this category. The nesting of the locks is given by a nested data structure, and therefore cannot be fully evaluated at compile-time.
   Five of these programs are correct, while the three others (D.10, D.11 and D.14) exhibit potential race conditions or deadlocks.

3.3 Evaluation process

3.3.1 Overview

A direct comparison between programs that are so different is very hard, even though all programs ultimately try to achieve the same goal. Static checkers cannot detect faults that only occur if certain references change at run time. Conventional dynamic checkers commonly only work within the given schedule for the threads, i.e. other interleavings of threads might lead to failures that go undetected. Moreover, dynamic checkers have the disadvantage that they require a running version of the program and therefore cannot be applied to incomplete programs; because of this, examples D.7 and D.8 had to be omitted from testing for dynamic checkers.

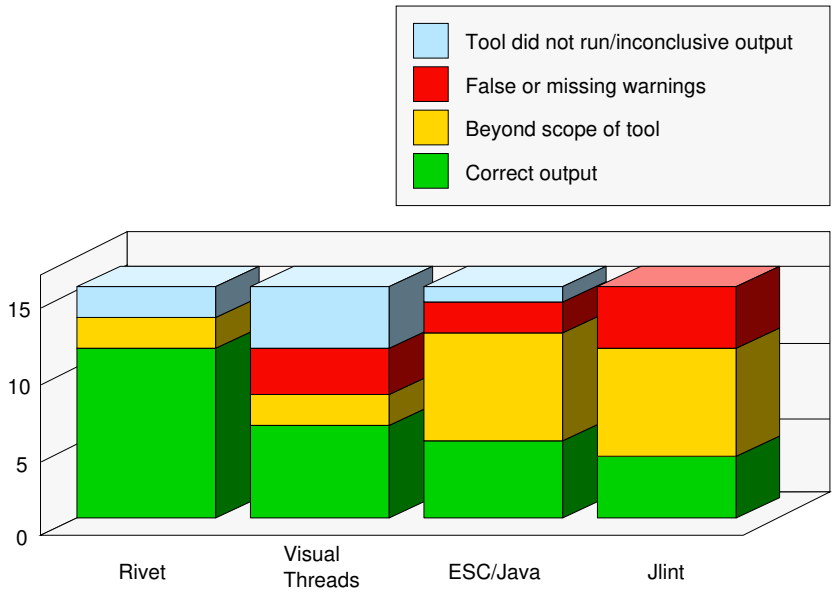

Figure 3.1: Test results for the 15 given examples.

Figure 3.1 shows an overview. The categories have the following meaning:

Tool did not run/inconclusive output: ESC/Java's theorem prover "Simplify" exited "unexpectedly" in example D.3, therefore it could not be evaluated. Rivet does not run anymore under modern Java Run Time Environments; the numbers for it had to be taken from [29], and no new examples could be tested with it. While VisualThreads ran on all examples, its output was sometimes not clear, or several test runs yielded different results.

False or missing warnings: If a tool produced "critical" warnings for a correct program, the output fell under this category. A "critical" warning was one that does not refer to design guidelines or properties of the program that are not related to the multi-threading problems investigated here (e.g. array bound checks).
An output without any such warnings for a faulty program was also counted under this category.

Beyond scope of tool: Dynamic checkers cannot be run on examples D.7 and D.8, since these are not full programs. Therefore, they were counted as "beyond the scope" of those tools.
Today's static checkers cannot yet handle more sophisticated locking structures, such as a (bounded) circular list or buffer, implemented as an array. Such a situation was present in the four version of the "shared buffer" and the three "Dining Philosophers" implementations (examples D.9 – D.16).
Jlint has no way of dealing with such a situation. After some experiments with modeling (`ghost`) variables in ESC/Java, it became apparent that the limitations of the scope of the different annotation statements presented a major difficulty in expressing more elaborate modeling conditions. Even if the annotations could have been carried out successfully (with an effort that would not be realistic under time constraints usually present in industrial projects), it is unclear whether the version of ESC/Java used would have been capable of verifying these algorithms.

Correct output: The checker issued a correct warning for a faulty program, and no warnings for correct implementations. It should be noted that in example D.8, Jlint passed because it was entirely ignoring the critical part of the program.

The simple numbers of correctly detected faults are misleading, even more so because certain problems are over-represented in order to investigate the behavior of the programs more closely. Nevertheless, it was attempted to test each program with as many of the example sources as possible.

MaC currently does not allow checking of typical multi-threading errors at all. Therefore the tests were canceled once the limitations in the current version were obvious. Future extensions may allow MaC to check for deadlocks, race conditions and liveness properties.

Table 3.1 shows what types of faults can be detected by each tool. Again, MaC was not included because the required extensions are not yet written.

	Inter-method deadlocks	Intra-method deadlocks	Race conditions	`wait/notify` deadlocks	Liveness properties
Rivet	Yes	Yes	Yes (using assertions)	Yes	No
Visual Threads	Yes	Yes	Special cases	Yes	Yes
ESC/Java	Yes	Yes	Yes	No	No
Jlint	Yes	No	Special cases	Yes	No

Table 3.1: Overview about the tested tools. Results from Rivet are taken from [29] and could not be verified since Rivet does not run under newer JVMs. MaC could not be applied to the given examples.

Inter-method deadlocks: Potential deadlocks caused by a problematic dependency of `synchronized` methods of different classes.

Intra-method deadlocks Deadlocks caused by an incorrect nesting of `synchronized` blocks.

Race conditions: Concurrent access to a shared resource. Jlint's capabilities are limited to direct field accesses, which is not good coding practice; it cannot detect race conditions via `get` methods. VisualThreads only detects race conditions when they actually occur at run time; incomplete locking schemes as such are not detected.

`wait/notify` deadlocks: If a thread holds several locks when `wait`ing for a lock, it will only relinquish the lock it is `wait`ing on. The inavailability of the other locks can lead to a deadlock.

Liveness properties: A guarantee that a program makes progress in its state space and is able to perform a certain service consistently. VisualThreads cannot guarantee this, but show *livelocks* (the absence of progress) with a high probability.

3.3.2 Program installation

Installation was fairly simple for all programs, with the exception of Rivet:

- The original Jlint comes as one (130 KB large) C++ file and a makefile for compiling it. The new version consists of several files.
- ESC/Java comes as an archive with binaries, examples and a shell script that needs to be customized (after setting some environment variables in the shell).
- MaC comes as an archive of Java `.class` files and needs to be added to the `CLASSPATH`.
- VisualThreads, being a commercial product, comes as an Alpha Unix package, where installation is automatic.

Rivet, on the other hand, is tightly tied to the virtual machine it uses. The main reason, according to Derek Bruening, is that "Rivet does all kinds of things that a later version of Java's security checker might complain about. It makes shadow versions of every class, classes with the exact same name but through a different class loader, and I'm not sure if the more recent versions of Java have closed that name space loophole." Also, a lot of other problems regarding the extension of native methods, minor incompatibilities between the bytecode files generated by different compilers and continuous changes in the Java Run Time Environment (JRE) broke Rivet each time a new version came out.

In this work, various combinations of the following Java compilers and JREs were used: Sun's JDK and JRE version 1.3.0, Blackdown JDK/JRE versions 1.3.0, 1.2.2 and 1.1.8; and jikes/kaffe. Sun's older JREs and Blackdown's version 1.1.8 would not run anymore under RedHat GNU/Linux 7.0, which was used as the development environment. Therefore, an older version of Linux had to be set up using VMWare in order to run both environments concurrently on the same computer. Since it became obvious that the newer class loader in version 1.2 would not cooperate with Rivet, version 1.1.8 was used; at that time, Sun had not even ported their JRE to Linux, so only Blackdown's version was available.

However, Rivet did not work under any of these configurations; indeed, the latest version for which it is known to work is 1.1.5, which is older than the currently supported versions at Trilogy. Therefore, Rivet would have to be ported to a newer JDK in order to become useful. Making Rivet work with version 1.2 or newer would require modifications of the Java class loader itself, because overloading built-in classes is no longer allowed there (although this restriction was not fully implemented yet in older versions and could be circumvented by setting the `CLASSPATH` appropriately).

3.3.3 Common traits

None of the tools can guarantee the absence of a certain kind of fault. The static checkers cannot detect whether the program is simple enough to allow a sound checking. Only some specialized checks allow an exhaustive verification; indicating the guaranteed correctness of certain aspects of the program could be a great help. For those checks where this is normally not the case, adding such a feature would not be very useful. Dynamic checkers, by definition, need a certain input to perform their checks on. Even then, VisualThreads was not successful at detecting a deadlock in all cases. Rivet is the only program that has the potential to detect a fault for sure, because it runs all possible thread schedules in sequence. Even then, the test is only representative for one test case.

Both static checkers could not deal with the complexity involved in the shared buffer and Dining Philosophers examples. While they could give some warnings about potential trouble spots, a full check lies outside the scope of a static checker. Possibly a preprocessor that generates one class for each instance of a Philosopher class, with the index of each instance given, could alleviate the problem in that case. However, such work is specific to this problem, and would not help in "real world" examples where the number of threads is either not strictly bounded or not even constant during program execution.

3.3.4 Testing procedure

Only ESC/Java and MaC required annotations or script specifications, respectively. Therefore, the tests for ESC/Java were usually run many times, until a suitable set of annotations was found. For MaC, some first experiments were done with different scripts, until it was found that currently MaC does not allow checking for liveness properties or deadlocks. Testing MaC was canceled at that stage.

3.4 Tool evaluation

Despite MaC's lack of multi-threading capabilities, this evaluation includes MaC. Rivet is also included, although it requires to be ported to the latest Java Run Time Environment before it can be used for today's Java programs.

For each tool, an overview is given first, followed by a brief summary of its fault-detection capabilities. Its strengths and performance are evaluated, together with the perceived difficulty of learning how to master the tool. While the latter is a very subjective measure, it is yet crucial for the success in an industrial application. Finally, after reviewing the limitations of each tool, a summary is given.

3.4.1 MaC

Overview

MaC is a dynamic checker that has two main components: a *run time checker* and an *event recognizer*. The latter communicates with a Java program that has been *instrumented* (extended) with special instructions that are triggered whenever certain operations occur. The run time checker then verifies whether these events violate the requirements of the program.

Even though there is no direct support for multi-threading issues yet, the goal was to express deadlock problems as liveness properties: by specifying that no thread is allowed to hold a lock for a "long time" (e.g. 5 seconds for simple programs), one could catch deadlocks when they occur.

Required knowledge and effort

MaC comes with a short manual giving a good overview about the different components of the tool. A second document introduces the definitions of the two annotations languages:

1. The "Primitive Event Definition Language" (PEDL) defines which Java variables and methods are monitored, and how these variables are connected to conditions that occur in the properties that will be monitored.
2. The "Meta Event Definition Language" (MEDL) describes the relation between events and conditions, how events are connected to each other and what sequences of events are allowed (properties of the program).

Both languages are quite simple and intuitive, yet powerful enough for most purposes. However, the current description lacks a good reference, so it is sometimes not easy to figure out the exact meaning of certain keywords.

Performance

According to Moonjoo Kim, who is currently working on MaC, "in the worst case of monitoring `i` of `for(int i=0; i<max; i++)`, [the] overhead is 100 times without considering property evaluation. Most of the overhead on this case comes from TCP socket communication overhead." However, this issue is currently being addressed, and an API is being written which allows the processes to communicate via pipes (if running all components on the same computer).

Limitations

The run time checker of MaC is synchronous; i.e. it is triggered whenever the event recognizer is called by the Java program. This makes it impossible to check for deadlock, because the event recognizer would wait forever on events in that case! Also, MaC does not yet have a way to obtain the current time, even though an event count can be obtained.

However, MaC would only require a minor extension and the absence of "stalling" (where not a single thread is active anymore and no events occur) in order to detect deadlocks. This could be simulated by having a dummy thread running that generates an event from time to time. Also, MaC would have to be augmented with the notion of (system) time for such checks. Race conditions are not directly supported, but changes in sources would still allow checks with the current version of MaC.

Because MaC is still work in progress, and the source code was not available, testing was not continued at that stage.

Summary

The simplicity of the annotation language and the wide area of applications (any kind of safety property or constraint can be checked) is very appealing. While MaC does currently not have any features that would allow it to tackle problems specific to multi-threading, extensions will be written for it within the year 2001.

3.4.2 Rivet

Overview

At the cost of a high overhead, Rivet performs an exhaustive checking by testing the program with all possible thread schedules. Unfortunately, Rivet requires a very old Java environment to work at all, because it has to circumvent numerous security features in order to work.

Detected faults

Because many examples were chosen from or based on the thesis about Rivet [29], Rivet would have successfully detected most faults if run on these programs. The two examples which were not yet a running program (Listings D.7 and D.8) could not have been tested.

In sheer numbers, Rivet would have been the most successful checker, although also the slowest one. Its exhaustive checking finds any problem that is not restricted to certain test cases. However, Rivet did not run on JDK 1.1.8 or newer and therefore could not be tested.

Performance

As documented in [29], the run time overhead would be at a factor of roughly 180 – 200. This makes it impossible to run Rivet on larger programs.

Summary

Rivet has quite a potential, but still needs a lot of work on it. It is doubtful whether anyone will port it to a current JDK, which would likely require modifications in the class loader. Even then, there are still many problems that have not been solved yet. However, Rivet incorporates many novel ideas, such as a virtual machine that can backtrack a step, and a systematic thread scheduler; therefore, it would be a pity if that work just died.

3.4.3 VisualThreads test results

Overview

VisualThreads is a dynamic checker that catches deadlocks, race conditions and potentially hazardous locking schemes. When starting this tool, a GUI appears that allows the programmer to enter the program name and all parameters; when running the program, the GUI continously informs the programmer about its status with a graph about the number of threads and events, and dialog boxes about violations (such as deadlocks). VisualThreads operates on the level of POSIX threads, which is probably not the best approach to monitor Java programs. Its focus lies on C and C++ programs.

Detected faults

VisualThreads seemed to be unable to detect circular locking schemes in Java, even though an example program written in C shows that it has this capability. Possibly the addresses of the object locks change at run time in the Java Virtual Machine. Once a deadlock actually occurs, though, it is always detected by VisualThreads.

Quite a few of the examples have been extended with `sleep` calls that stop a thread for a random period of time; without these calls, the faults would not show up at run time and go undetected by VisualThreads. Since it is normally not the case that a programmer inserts random `sleep` calls at critical sections, the actual usefulness of VisualThreads could be quite a bit lower than the numbers suggest. However, because of the large slowdown introduced by VisualThreads, Java's thread scheduling acts quite differently from its normal behavior, so it may still detect a number of faults that would not occur during normal execution.

Strengths

The graphical output allows easy monitoring of the program: It is easy to see when a program is stalled and does not change its state anymore. VisualThreads does not automatically abort the program, though, so it is not suitable for automated testing (especially since it still fully utilizes the CPU when it is monitoring an idle program).

VisualThreads can be used without prior knowledge of problems that can occur in multi-threaded programs; each detected violation is displayed with detailed explanations. It is also very easy to use, due to its graphical user interface.

Its main potential lies in detecting *possible* deadlocks by observing the order in which locks are taken. This feature seems not to work in Java.

Performance

Because VisualThreads was running on a rather old, slow Alpha computer, it is hard to judge its performance. Indeed, a lot of the given overhead may have been caused by the GUI rather than the core program. A rough guess is that it slows down a Java program by at least factor 20.

Limitations

Because VisualThreads only runs together with its graphical user interface, it is not suited to automatic or overnight testing. Also, it needs a fast machine to run on; the fact that it only runs on Alpha Unix makes it harder to get access to such a machine. The generated trace files grow very fast (at a rate of several megabytes per minute), which further slows down the execution.

Summary

Being a commercial product, VisualThreads is the most powerful run time checker available. It requires a well-equipped computer to run on, but can be used on any executable program.

3.4.4 ESC/Java test results

Overview

ESC/Java works mainly on (preferably annotated) source files. If the source code is not available, a specification file can be given (which includes all method declarations and annotations about the behavior of the methods). Alternatively, ESC/Java can also process class files directly, although with much less useful results.

Detected faults

ESC/Java is by no means a *sound* checker (in the sense that it detects all faults), nor is its goal to be *complete* (in the sense that it never gives spurious warnings). [36, Appendix C] explains why:

> "An unsoundness is a circumstance that causes ESC/Java to miss an error that is actually present in the program it is analyzing. Because ESC/Java is an extended static checker rather than a program verifier, some unsoundnesses are incorporated into the checker by design, based on intentional trade-offs of unsoundness with other properties of the checker, such as frequency of false alarms (incompleteness), efficiency, etc. Continuing experience, and new ideas, may lead to re-evaluation of these trade-offs, with some sources of unsoundness possibly being eliminated and others possibly being added in future versions of ESC/Java."

One point that is maybe not quite clear in this quotation is the fact that for certain properties, sound checks may be possible, but would require large extensions of the given checker. These "intentional" trade-offs are usually due to the fact that this (large) project is still far from being finished, and a compromise had been made to produce a working program checker on time.

The focus of ESC/Java is to verify the validity of *assertions* statically. Therefore, it requires annotations in the code in order to be really useful (although certain properties are checked by default). Many annotations are some form of *assumption* where the user supplies additional information to the program, which would otherwise not be available at compile-time. The checks for race conditions and deadlocks are specific extensions of these two primitives and have not originally been the goal of ESC/Java. However, there is work in progress that will make checking for synchronization problems easier.

In the examples, the `DeadlockWait2` example (Listing D.4) was not counted because ESC/Java's theorem prover "Simplify" crashed during execution under Linux. This failure could not be reproduced under Solaris by Compaq's development team, and it can be assumed that it will be fixed for the next Linux version. Also, the Jlint test example gave an output that was hard to interpret; with the improved support for synchronized methods in the next version, it should be clearer.

The Dining Philosophers problem (Listing D.14) could have been made more tractable for ESC/Java by fixing the number of processes, possibly also by preprocessing the code (see Section 3.3.3). However, the amount of annotations necessary in that case (and also for the shared buffer problem) would have been really large, coming close to a formal proof. This is outside the usage that can be expected in industrial application programming, where a tight schedule will not allow for the time needed for constructing model variables that reflect properties of the program which hold during the execution of all threads. Once someone gets that far, the main work of verification is done by a human rather than the computer. When using ESC/Java, it is more beneficial to focus on the faults that occur under simpler circumstances.

Strengths

ESC/Java finds indeed all of the simpler faults and only really fails in two cases: First in the example given in the ESC/Java manual (Listing D.8), where a temporary change in the data structure cannot be reproduced by data flow analysis; second in the SplitSync example (Listing D.6), where it reports a potential deadlock rather than a race condition. In the two more complex cases

(shared buffer and Dining Philosophers), it hints at trouble spots in the code, regardless of whether the given example works correctly or not. Even though this may look like a failure, one has to keep in mind that a user can turn a warning off for a given position in the code, making it easy to eliminate spurious warnings once they have been examined.

Required knowledge and effort

ESC/Java is a very powerful tool, encompassing warnings in 21 categories, 24 annotation pragmas, and 18 specification expressions which are needed for some annotations. Its rich syntax is similar to a small programming language of its own, but on a more abstract level. Therefore fully mastering the annotation language requires a thorough understanding of Java, especially if model variables and lock set annotations are to be used. If one focuses on simpler checks, one can start with fewer, more intuitive pragmas, such as `assert`. Future versions of ESC/Java will hide some of the internal complexity when dealing with `synchronized` blocks, making its usage simpler. The programmer also needs to have a basic understanding of preconditions, postconditions and invariants. Unfortunately, the manual tries to be both an introduction and a reference; it does quite well at achieving the latter, but on the cost of the former.

Program	Size (NCLOC)	Lines of annotations (LOA)	LOA/ NCLOC
Deadlock	50	2	4.00%
Deadlock2	50	8	16.00%
Deadlock3	48	3	6.25%
Deadlock-Wait	43	3	6.98%
Deadlock-Wait2	65	6	9.23%
SplitSync	26	2	7.69%
Jlint	26	9	34.62%
ESC/Java	77	14	18.18%
Buffer	66	8	12.12%
BufferSem	90	18	20.00%
Philosopher	93	11	11.83%
PhilosopherHost	116	16	13.79%
Total	750	100	13.33%

Table 3.2: Annotations required for code examples. Annotations are sometimes incomplete (certain checks were disabled). Annotations for the Semaphore class were counted towards all programs using semaphores. The given numbers reflect the true annotations after a small extension to ESC/Java will be added, which will allow for more concise annotations.

The number of annotations required has to be taken with a grain of salt, because the annotations were geared towards checking for deadlocks and race conditions; certain warnings, such as potentially incorrect array accesses, were ignored. In the current version, the annotation overhead was acceptable, even though one sometimes has to invest some time into finding the right set of annotations to use if complex relationships between objects should be expressed.

For simple cases, the annotations are trivial, usually only for ensuring that references are not `null`.

Performance

ESC/Java is definitely slower than a compiler, since it has to repeat most of the compiler's task and run its theorem prover on top of it. The overhead is not too large, though; in most cases, ESC/Java should be suitable for running before checking in source code, and it is definitely useful as a verification stage prior to testing, given the code is sufficiently annotated. However, the output is not meant to be processed automatically (unlike test cases).

Limitations

A major problem when working with ESC/Java is that it is very likely to generate warnings for any `synchronized` statement. Only with extra annotations, one can remove these warnings. Sometimes, finding the right annotation can be very hard. For an incorrect annotation, ESC/Java will complain about a violated invariant. At that point, it is not clear whether the annotation was incorrect or merely insufficient for ESC/Java's theorem prover, or whether there is a genuine fault in the program. This is precisely the question that the prover should answer, but for complicated programs, it cannot always help.

ESC/Java's warnings are usually very concise – more often than not, a bit too concise. Sometimes, some extra information or a small counter-example would be very helpful. Right now, counter-examples are only given in an internal format, which corresponds to an intermediate language which is used when translating the Java program into a proof. These counter-examples are very hard to read, and it is not possible to understand them fully without thorough knowledge about ESC/Java internals. This is certainly an area that needs improvement. One has to keep in mind that generating examples in the real programming language, based on properties disproved by a formal checker (in a highly abstract representation) is a very hard problem. Bandera and the SLAM tools ([2, 6]) are supposed to solve it.

Summary

ESC/Java is definitely the most powerful static checker currently available. While its key strengths are not in the area of synchronization problems, there are already a couple of features that allow very useful checking. Work is in progress to make checking for deadlocks easier and more powerful.

ESC/Java requires some time to be mastered; it remains to be seen whether the effort is worthwhile, given the current stage of implementation.

3.4.5 Jlint test results (version 1.11)

Overview

Jlint works directly on the compiled classes and therefore does not require the source code. However, it does not allow for annotations and templates either, restricting any checks to the ones which are "hard-wired" into the source code.

On the other hand, this makes it extremely easy to use; very little prior knowledge is needed to run the tool and interpret its output.

Detected faults

Jlint only performs control flow and very limited data flow analysis. Therefore, it can only detect deadlocks that occur on a method level, i.e. where the `synchronized` statement applies to the entire method and obtains a lock on the current instance (`this`). Synchronization problems within methods, where `synchronized(resource)` applies to a block of code, are ignored.

In general, checking these locks is very hard, due to the *aliasing problem*. Because a reference can be copied or changed during run time, the content of a certain variable cannot always be determined statically. Konstantin Knizhnik, the author of Jlint, says:

> "The problem with analyzing `synchronized(lock) { block }` constructions is that "lock" can be arbitrary expression, and it is impossible to detect without execution of [the] program whether two such expressions refer to the same object. Even the simplest case, when lock is just the name of variable, require full data flow analysis to be able to make conclusion which object is locked."

However, if a variable is shared between threads (either as a singleton instance or a static variable) and does not change during the execution, a check can still be done, without (complicated) data flow analysis. For testing this, the two basic deadlock programs (Listings D.1 and D.3) were transformed into programs that exhibit the same locking problem, but between methods rather than within the same method. Jlint successfully detected these deadlocks and gave a very useful and concise description.

Strengths

Jlint's is most useful at checking synchronization problems, where it supports deadlocks on a method level, and certain race condition checks. It also performs a number of other checks using a simple data flow analysis, such as the check for possible `null` references. These faults are easier to find than synchronization problems. Jlint also has some specialized checks which are very reliable, but apply to uncommon types of faults.

Performance

Jlint's biggest strength is its extremely fast performance (since it has been written in C++), which makes it even faster than the current Java compilers (which have to analyze the entire source while a class file is much simpler). This makes it very easy to run Jlint often, once the source files compile.

Required knowledge and effort

As it has been mentioned above, Jlint requires very little prior knowledge. It is recommended to read the manual, which is not too long and still encompasses all of Jlint's functionality, before using the tool. No knowledge other than a good understanding of Java is required, as Jlint does not allow any annotations.

Limitations

Jlint's original scope for synchronizations was very limited, because it only worked on a method level. An extension for the remaining Deadlock examples was shown to be possible with a reasonable effort. In order to verify whether the assumptions needed for doing these checks are valid in real code, an analysis of Trilogy's code followed (see Section 3.5).

Summary

Jlint's simplicity and speed make it worth using on code of any size. However, it does not allow any customization, so repeated usage for gradual refinement is a less likely scenario. Jlint fails to detect many faults, but is generally conservative at issuing warnings, which makes it quite useful.

3.5 Statistical analysis

Knowing now the capabilities of these five tools, it still was unclear which one would be best suited to a real world scenario. How often do synchronized blocks, a feature that Jlint does not support, occur? In synchronized blocks, what resource is usually synchronized on? Are complex locking schemes (such as lists of locks) common?

An analysis of Trilogy's code and other code bases sheds light into this problem. With an increasingly fine level of detail, several aspects of concurrent programming in Java were analyzed. First, the scope of the analysis will be shown, then the results.

3.5.1 Scope

The analysis consisted of three steps:

1. Getting a count of `synchronized` methods and blocks. Their number gives a measure of "parallelism" of a Java package. However, the relative numbers were far more interesting, as they give an indication of what Java checkers should be capable of in order to cover most cases occurring in real software.

2. The different cases of `synchronized` blocks:

   (a) `synchronized([this.]getClass()`: this is a synchronization on the current class. In most cases, the `synchronized` block could be substituted by a call to a `static synchronized` (class) method.[1]

   (b) `synchronized(this)`: a synchronization on the current instance. The `synchronized` block can be substituted with a call to a `synchronized` method. Synchronizing on an instance is far more common than synchronizing on all instances of a class (as in case 2a).

   (c) Other cases. In that case, an arbitrary object is synchronized on (usually a field of the current class).

[1] If instance (non-`static`) fields are used within the `synchronized` block, a direct substitution is not possible.

3. In case 2c, what kind of variable is synchronized on? There are quite a few possible cases:

   (a) Members of the current class:
   - Class (`static`) variables. This is usually the case when a resource is shared between different instances and may only be accessed by one instance at a time. The `static` reference is usually still points to an instance, not a class – therefore the distinction between synchronizations *on a class* (case 2a) and *on a class variable* (this case).
   - Instance variables. Often, this case corresponds a singleton instance [51] which holds a shared resource.
   - Inherited fields. Depending on whether they are static or not, they correspond to one of the two cases above. However, they are more difficult to analyze (and also more difficult to keep track of for a programmer).

   (b) Members of another class: a direct field access to another object usually corresponds to unsafe design, but is sometimes used for a performance advantage.

   (c) Function parameters and local variables: in this case, the reference to the object that is locked on is obtained dynamically, and requires a data flow analysis across methods, often across methods of different classes.

Case 3a usually corresponds to a shared resource that is initialized once during instance (or class) creation and then used throughout the lifetime of an instance. Possibly, that resource may be re-allocated under special circumstances. Because these references generally do not change, their analysis is relatively simple. Therefore, Jlint could be easily extended to encompass such cases. The question was now how common such a case is.

In the other cases, interactions between classes need be analyzed. This requires a *global data flow analysis,* which beyond the scope of currently available checkers.

3.5.2 Analysis method

Cases 1 to 3a could be covered by a shell script, while a closer analysis of `synchronized` blocks was done by a Perl script implementing a small Java parser. See appendix A.1 on page 63 for a description of these programs.

3.5.3 Analysis results

A listing of all the information gathered during the analysis can be found in appendixes A.2 to A.4. This section summarizes these results.

Trilogy's source code

`Synchronized` methods make up about the majority of the occurrences of `synchronized` statements (see table 3.3 and figure 3.2). The synchronizations

on `this` or the current class make up a small amount. 35% are case 2c, where an "arbitrary" variable is synchronized on. This shows that the simpler cases already cover two thirds of all synchronization issues.

487	Total	100.00%
265	`synchronized` methods	54.41%
51	on `this`	10.47%
93	on a class or instance var.	19.10%
30	on a class (on `r.getClass()`)	6.16%
48	Other cases	9.86%

Table 3.3: Overall usage of `synchronized` statements in Trilogy's code.

Out of these harder cases, the majority is a single shared resource that is used among several instances (using a `static` variable) or threads using the same class. The combined "simpler" cases (synchronizations on `this`, class or instance variables, or classes) make up the major part of all synchronization statements (about 85%). Also, this is a substantial increase compared to the `synchronized` methods only. It should be noted that the kind of complexity encountered varies a lot between different packages. For example, the core classes, which administrate many shared resources, contain many synchronizations on such resources. Other packages, especially wrappers, frequently obtain a reference to such a shared resources through interaction with different classes. What do other packages look like? Are they similar to Trilogy's?

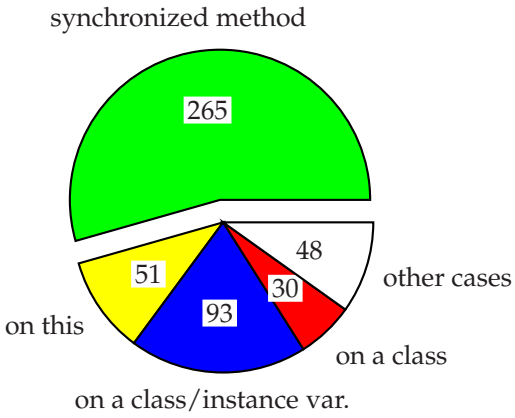

Figure 3.2: Overall usage of `synchronized` statements in Trilogy's code. A distinction is made between `synchronized` methods, synchronizations on object fields (which are references to an object instance), on classes and other cases.

Built-in Java packages

While the big picture is similar, there are huge discrepancies among the different packages. Especially noteworthy is the high number of local variables (due to interaction between objects for managing shared resources) in the `awt` package. The more complex cases are significantly more frequent than in Trilogy's code (17.59% as opposed to 9.86 %). This is not surprising, since these

packages have to implement the core Java functionality, and it is desirable that much of the complexity of multi-threaded programming is taken away by well-designed, thread-safe core classes.

Other packages

Swing

Some `javax.*` packages (`accessibility, naming, swing`) are now included in the Java Run Time Environment, version 1.3 and newer. Except for the Swing GUI toolkit, almost no synchronization statements are used at all. These packages are mostly single-threaded, but thread-safe. Hence most synchronization operations are guards against concurrent usage. The high number (74.77%) of `synchronized` methods and `synchronized this` blocks shows this. Also, the total number of `synchronized` keywords in the code is lower than in any other package analyzed. One can conclude that in packages that deal less often with concurrency issues, the synchronizations on the `this` instance via `synchronized` blocks or methods are by far the most often used ones, since they usually only serve to guarantee an "atomic" operation within a block or method.[2]

OMG (CORBA) packages The `omg.*` packages that come with the JRE 1.3 implement the CORBA functionality for Java. Since the real multi-threading issues are in the underlying natively implemented framework, which has to be able to deal with many requests at a time, the Java packages only include synchronizations on the current instance or class, and no other locking schemes.

Concurrency package Since this package implements higher, concurrent "building blocks", such as shared read locks, it is inherently the most complex one with respect to parallelism. Nevertheless, the number of locks on the current instance makes up 51.41% of all locks. Even in such a complex package, the "atomic block or method" type of synchronization is the most common one. Of the rest, the overwhelming majority (43.13 % of the total) were synchronizations on a class or instance variables (in 195 out of 229 cases, fields that were inherited; in 31 cases, non-`static` fields). Usually, these variables were part of a more complex data structure (e.g. a node in a list or queue), and therefore were sometimes quite hard to analyze (also see Section F.3 on page 128).

ETH Data warehousing tool This tool, having not many lines of code, was the second most complex package analyzed (unless some Java packages such as `java.beans.*` or `java.io.*` are considered separately). However, 83.08% of all cases were `synchronized` methods, and the rest were synchronizations on a class or instance variables. This package should be easier to handle for a static checker because there are no interactions between classes when a lock has to be obtained.

[2] Strictly speaking, the operation is only atomic with respect to the `this` instance.

3.5.4 Common traits

As one can see, there seems to be a trend towards simpler cases of synchronizations in smaller packages and packages that provide a wrapper functionality. All in all, the trend is evident that these simpler cases prevail, even in complex packages. Even most of the complex cases seem to be manageable without inter-object data flow analysis. This is a very encouraging result, as simple checkers can already cover quite a large percentage (about 85 %) of all `synchronized` statements. A "simple" checker in this context is one which can perform inter-object data flow, track references across methods to see if they are not changed, and check all synchronizations using such references or `this`. Of course, it is not certain how likely faults are present in these simple cases, and how likely faults occur in the remaining 15 %, the complex cases.

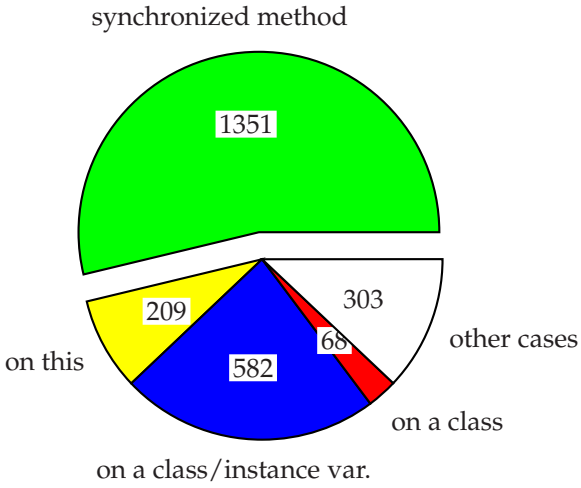

Figure 3.3: Total usage of `synchronized` statements in all analyzed packages. A total of almost a million lines of code (LOC)was analyzed.

3.6 Comparison of the results

3.6.1 Test examples

For analyzing multi-threading problems, MaC cannot be used yet. Therefore it was not considered further.

Rivet

By simply looking at the chart (figure 3.1 on page 16), Rivet seems to be a clear winner. Being an exhaustive checker, it can catch all multi-threading faults by nature. This comes at the price of a high overhead, and an architecture that is no longer compatible with newer virtual machines. Using Rivet in current environments would require a significant porting effort.

VisualThreads

Based on a powerful deadlock/race condition detection algorithm, VisualThreads is very successful at detecting many kinds of faults. Unfortunately, the fact that it monitors the POSIX thread API, rather than the locks in the virtual machine itself, makes it much less capable finding faults in Java programs. A specialized Java virtual machine with the same monitoring algorithms would be much more useful (and probably also faster).

ESC/Java

Due to its more powerful theorem prover, ESC/Java successfully handles all the simpler test examples and only fails in very complex cases. However, it is hard to learn and much more time consuming to use, because the code (either as source code or bytecode) has to be annotated in order to be checked effectively.

Jlint

Jlint's analysis capabilities are limited, since it does not build on a sophisticated model. Despite this simplicity, Jlint was quite effective at finding certain deadlocks. From the developer's point of view, it is definitely the easiest one to use.

3.6.2 Real world problems

Dynamic checkers have been more successful with the given examples. Nevertheless, there were other drawbacks involved in the two specific tools that were tested. As of January 2001, one of the static checkers evaluated here would likely be more useful. All dynamic tools would require major changes or extensions, which would not have been feasible within the given time.

The goal of the statistical analysis was to find whether the static checkers would be severely hampered by their incapability of analyzing complex locking schemes such as present in the shared buffer and Dining Philosophers examples. The analysis showed that they probably are not – the "easy" cases are far more likely than the "hard" ones.

The original version of Jlint (as opposed to ESC/Java) only covered `synchronized` methods; as the statistical analysis showed, an extension to `synchronized` blocks would increase its scope and usefulness substantially. Jlint's small size, its free availability and the fact that it was not being developed further at that time made it easier to extend. Therefore the step was taken to extend Jlint's analysis.

Of course a statistical analysis can only cover the *frequency* of certain locking constellations, not the likelihood that a fault is found in these. Therefore the actual value of a tool that can cover the remaining few cases which are hardest to analyze may be a lot higher in practice. There is no data available yet on that topic; moreover, judging the "severity" of a fault is a very subjective process. As for the state of the art, simpler problems need to be solved first before the very hardest ones can be tackled.

3.7 Summary

This chapter described the questions that are relevant for choosing a checking tool, no matter what its technology (since for the developer, the usefulness is more important than the technology used). Because running all tools against the entire code base is not feasible within the given time, a small set of representative test examples was chosen.

Choosing the test examples based on certain metrics was abandoned in favor of choosing examples that exhibit certain typical problems in multi-threaded programming. The *importance* of each aspect that a tool covers was judged by their number of occurrences found in the source code analysis. This metric may be too simplistic, but no refined metrics are available yet.

It could be seen that even though static analyzers cannot cope with really complex locking schemes, they successfully deal with simpler cases. These are much more common in real source code. This is why the decision was made to extend Jlint's applicability to `synchronized` blocks. This extension would bring Jlint, the simplest and fastest checker, to the same level of more advanced static checkers, such as ESC/Java, without sacrificing Jlint's speed and ease of use.

Chapter 4

Jlint extensions

In its available version (as of December 2000), Jlint was only applicable to `synchronized` methods, the simplest case of synchronization in Java. This was disappointing, because the other cases, `synchronized` blocks, make up about 45% of all synchronization statements in Java, as the evaluation showed. Therefore, the goal was to add support for `synchronized` blocks and refine the deadlock and race condition checks Jlint is performing with this finer granularity.

The first two sections describe the Jlint extensions in more detail. The next two sections deal with algorithmic changes and their implications in Jlint's code. Problems encountered are discussed afterwards. Finally, the results of the application of Jlint to several software packages are shown.

4.1 How Jlint works

Jlint works in two passes. Most checks are done in the first pass. The two passes work as follows:

1. Jlint reads bytecode (`.class`) files and parses them. It then analyzes each method, examining code locally. During theses proofs, the call graph is built.

2. The (global) call graph analysis follows when all class files have been processed. In this pass, Jlint checks the call graph for potential deadlocks.

4.2 Goals

The main goal was to extend Jlint's applicability to the "synchronized block" construct. This would allow the following extra checks:

1. **Wait/notify analysis:** For each `wait`, `notify` or `notifyAll` call, the current thread has to own a lock on the callee [50, p. 414]. Rather than only checking whether *any* lock is owned, the extended Jlint should know which ones are owned.

- If the monitor for the object that is being waited on is *not* held, a race condition or `IllegalMonitorStateException` might occur when the thread resumes execution.
- If the waiting thread holds other locks, these are inaccessible for the time while that thread is waiting. Deadlocks are possible under this scenario.

2. **Synchronized block analysis:** In the same way `synchronized` method calls are checked for potential deadlocks (loop in the call graph), acquiring the monitor in `synchronized` blocks should also be checked for deadlocks. While it was clear (from Listings D.1 and D.2 on pages 89 - 90) that `synchronized` blocks can, under some circumstances, be treated equivalent to calls to a specialized method, it was not clear initially *how* this should be included in Jlint.

3. **Lock change analysis:** During a manual analysis of intermediate files of the statistical analysis, a potential race condition has been discovered in the ETHZ data warehousing package [24] because a lock variable (which was an array, and as such a resource pool) was changed outside any monitors. Because only accesses to individual array entries were guarded by a lock, but not a change to the entire array, a fatal race condition could occur when the array is re-initialized with a different size. It would be desirable if Jlint could find such faults automatically.

Rule 3 is best illustrated by an example. Listing 4.1 shows a scenario that is quite similar to a real case where such a fault has been found. When the entire resource pool is re-allocated, its creation is not guarded by a higher lock. Introducing a lock variable for the resource pool (a simple `Object` will do) and using that one for locking *instead of* the array will fix this bug.

4.3 Implementation of extensions

Now, all these extensions have been implemented. The major algorithmic changes in Jlint for implementing the three major new features are described in this section. The wait/notify analysis was implemented at a method level (locally). The synchronized block analysis is included in Jlint's second (final) pass, the call graph analysis. The lock change analysis is partially implemented locally to each method, with a final analysis at the end of parsing each class.

All extensions have in common that the aliasing problem has to be solved, at least partially. In the Java Virtual Machine, all operations work on a stack of operands. Therefore, the "alias" to each value on the stack has to be known in order to perform any checking.

4.3.1 Tracking reference aliases

How field accesses work in the JVM

A *field* is a variable that is part of each object instance (or class, for `static` variables). Because the operand stack, where all values are "nameless", is central to the Java Virtual Machine, it was necessary to add **reference tracking**, which

Listing 4.1 An example showing a race condition when re-initializing a resource pool. Because such a fault has been found manually in a data warehousing package, a rule to check this was added to Jlint.

```
class ResourcePool {
  int size = 100;
  int count = 0;
  Object[] resources = new Object[size];
  public boolean addEntry(Object entry) {
    if (count < size) {
      synchronized(resources) {
        resources[count++] = entry;
      }
    }
  }
  public setSize(int newSize) {
    size = newSize; // race condition!
    count = 0;      // another race condition!
    oldres = resources;
    resources = new Object[size]; // race condition!
    synchronized(resources) {
      /* restore old values */
    }
  }
}
```

keeps track of the "alias" (the field name) of each stack element. In order to understand the way how field accesses works in Java bytecode, here is a small example ([50, p. 381]):

Java source code	Compiled bytecode
`int getIt() {`	*Method* `int getIt()`
`  return i;`	`  0 aload_0  // this`
`}`	`  1 getfield #4  // Field Example.i I`
	`  4 return  // returns first element on stack`

This example shows a key aspect of how fields are treated in bytecode (and in the virtual machine): fields are never used directly. Instead, they are copied on top of the stack with a `getfield` instruction. From then on, any manipulations of that field are made on the stack. A `putfield` instruction stores the top value of the stack in an instance field.

Tracking values on the stack

The `putfield` and `getfield` instructions have to be treated specially, as they are the only indicators of the "true" identity of a value on the stack. The same goes for `putstatic` and `getstatic`, which are the equivalent instructions for `static` fields. In the extended Jlint, a reference to the original value (the field of a class) is kept for each stack element. In order to keep track of the values used in the stack, all stack instructions had to be augmented with code that copies that reference.

Moreover, the `new` instruction (which allocates memory for an object and returns its reference) had to be treated specially as well.[1] For each new call, a

[1] The `new` instruction does not directly correspond to a constructor. It only allocates the memory needed. A subsequent call to a special `<init>` method then runs the constructor and any other

new field descriptor with the name <new#n> is created, where #n stands for the number of the object that is created (the nth occurrence of the new operation in that .class file).

Finally, the this pointer is always the 0th element on the stack (i.e. the 0th argument of a method) for each non-static method. Therefore, at the beginning of the analysis of each method, the 0th element is set equal to a special <this> field, which is created prior to the analysis of each class. This name (like the <new#n> fields) does not exist in the constant pool of the class, and therefore has to be allocated specially.

4.3.2 Wait/notify analysis

While checking each method, Jlint has to keep track of the locks that the thread executing that method is holding. A lock on this is held while a synchronized method is executed. In synchronized blocks, a lock on any object is acquired and released with monitorenter and moniterexit operations.

Lock set class

Whenever a monitorenter or monitorexit operation is encountered, the name of the field that the locking operation works on is needed. Using the reference tracking described in the previous section, this problem is solved. Furthermore, four major operations have to be performed when checking the correctness of locking:

1. Adding a lock to the current set of locks.
2. Removing a lock from the lock set.
3. Checking whether a lock is currently being held.
4. Getting the most recently acquired lock.

Since properties 3 and 4 cannot be implemented efficiently with a single data structure, a special Locks class encapsulates these (and a few other) operations. It uses both a vector and a hash table, which is the easiest way of performing both operations efficiently. Of course, since the lock set is always small, the performance was of minor importance.

Wait/notify analysis

With the given functionality of the lock set, it is now trivial to check whether the most recently acquired lock was the right one (the callee of the wait or notify method), or whether any other locks are held.

4.3.3 Synchronized block analysis

This extension allows, additionally to synchronized methods, also an examination of deadlocks in synchronized blocks. First of all, a synchronized block has to be recognized as such! This sounds trivial, but it is made much

initializations (such as the ones in variable declarations such as int i = 1).

harder by the fact that `synchronized` methods are supported within the virtual machine: In the bytecode, only a special flag is set for each `synchronized` method, but there are no special instructions. However, `synchronized` blocks are implemented via two special instructions: `monitorenter` and `monitorexit`. Section 4.3.1 has described how the original values of the argument to these two operations can be kept track of.

Originally, Jlint's call graph only included method calls. Even though it was shown that `synchronized` blocks could be converted to method calls to a special method of a special class, that extension was not quite straightforward to implement. In the end, a method call to a special `<synch>` method of the "instance" `OWNER.NAME` was chosen to model the dependency between `synchronized` blocks and method calls. `OWNER` is the class that "owns" the lock (the one where the lock is an instance or static field). The `NAME` entry corresponds to the name of that field.

```
public class Example {
  Object lock = new Object();
  Object innerLock = new Object();

  public void foo() {
    synchronized (lock) {
      synchronized (innerLock) { }
    }
    synchronized (lock) { }
  }
}
```

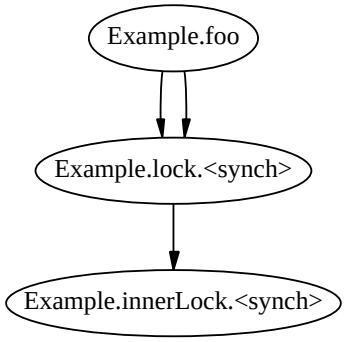

Figure 4.1: Call graph extension for `synchronized` blocks.

A `synchronized` block results in an edge from the current method to a pseudo method representing that block. If a method `Example.foo` acquires a lock on the variable `lock`, the edge `Example.foo` → `Example.lock.<synch>` is added to the call graph. If the *same* method acquires another lock within the first synchronized block, the edge `Example.lock.<synch>` → `Example.innerLock.<synch>` is added. If the same lock is released and then re-acquired, the same pseudo methods are used. This is because only the *nesting* of method calls and `synchronized` blocks is important, not the *order* in which they occur. Figure 4.1 illustrates this.

Figure 4.2 shows a slightly longer example with its extended call graph. What is not shown in this example is that method calls *within* a `synchronized` block require another edge from that pseudo method to the called method.

As one can see, this seamlessly combines the analysis of `synchronized` blocks with `synchronized` methods. In this case, the potential deadlock over the two fields `a` and `b` forms a loop in the extended call graph. The standard loop detection algorithm detects this loop without any changes. The only change that was needed in the remaining code was for reformatting the output. Despite this, it is not clear how extensible such an algorithm is to further alias analysis, especially when a lock variable is obtained via method calls.

In order to avoid duplicate locks, the new Jlint uses the Locks class (see Section 4.3.2) to keep track of the lock usage. By knowing which locks are held at the entry of a method, one can avoid adding another edge to the call

```
public class Deadlock {
  Object a = new Object();
  Object b = new Object();
  public void foo() {
    synchronized (a) {
      synchronized (b) { }
    }
  }
  public void bar() {
    synchronized (b) {
      synchronized (a) { }
    }
  }
}
```

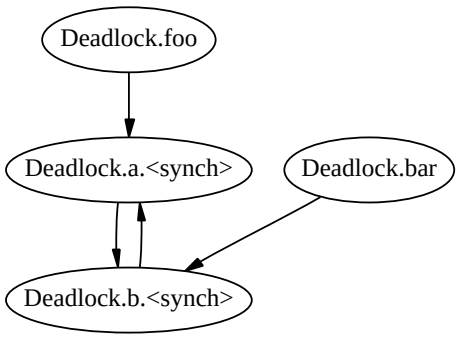

Figure 4.2: Call graph extension for Listing F.2.

graph. This originally was a major problem. There is no API for building the call graph, nor is there any way of checking for the existence of an edge in it. The call graph is assembled in the method descriptor class (in the main loop of the local analysis), and only the algorithm itself guarantees that no edge is added twice. This works fairly well until one extends the call graph to `synchronized` blocks (see Figure 4.1).

The chance is fairly high that the same method is called in several places with the same lock being held. This would result in the same edge being added several times to the call graph. Not only would the same warning be issued multiple times, but the call graph analysis would also be slowed down greatly. That algorithm might even be stuck in infinite loops. Such a behavior had to be avoided at any cost.

By keeping track of the locks that have already been acquired when a method is called, no redundant edges in the call graph are created.

4.3.4 Lock change analysis

The last extension was an analysis of changes made to locking variables, i.e. `putfield` commands on them. For each class, another lock set is used for this purpose only. Each variable in that set is checked for assignments outside the two special initialization methods `<init>` and `<clinit>` and outside any monitors.

In fact, the check is done for any reference during the local (method) analysis, since their usage as locks is not known beforehand. At the end, warnings are printed for each variable that has been used for locking purposes.

4.4 Code changes

This section describes the changes in Jlint's code at a more detailed, lower level. Table 4.1 shows the changes to the Jlint code in simple numbers. Version 1.12 has been formatted using XEmacs' "autoindent" feature. Thus the large code "growth" from version 1.11 to 1.12, which are identical except for the fact that the monolithic header and code files have been split up into several files (one for each class, or a few classes that belong together). Splitting up the files also introduced some extra whitespace, along with many extra include directives

and include guards; this, together with the reformatting, makes up the large difference in lines. In table 4.1, the size of the different versions in lines of code (LOC) and non-comment lines of code (NCLOC) is listed.

Version	LOC	NCLOC
1.11 (last release)	5270	4533
1.12 (separate files for most classes)	6254	5110
2.0 (enhanced Jlint)	6994	5615
Difference between versions 1.12 and 2.0	740	505

Table 4.1: Growth of Jlint code

Since none of the old code has been rewritten (as it should eventually be), the code reuse was quite high, at the cost of much time spent for understanding the code. Only very few changes to existing code were made; the major part of the implementation of the new features were made by additions to existing methods, or by creating new classes. Partially, the fact that the existing code was hard to change was responsible for that, but also the fact that `synchronized` blocks were not at all supported by Jlint before.

String pool

For being able to create symbolic constants that were not part of the "constant pool" in a `.class` file, a special string pool class was created. The reason was that some of these constants could not be stored locally, because they would have to be retrieved much later for printing error messages.

New messages

- Lock `a` is requested while holding lock `b`, with other thread holding `a` and requesting lock `b`. This message gives a more detailed output about (potential) deadlocks occurring with `synchronized` blocks. The analysis in the extended call graph is the same as for methods.

- Value of lock `a` is changed outside synchronization or constructor. This fault has been found in the ETHZ data warehousing package [24]. In that package, an array object represented a resource pool. Each write access to an entry was guarded with a synchronization over the entire array. Changing the size of the resource pool required allocating a new array. Such a situation can lead to a race condition, if another thread is using the object while it is re-allocated. Therefore this rule was created; also see Section 4.2

- Value of lock `a` is changed while (potentially) owning it. A similar situation as above. This fault has never been found in practice; it would likely result in an `IllegalMonitorStateException` at run time.

- Method `instance.wait/notify/notifyAll` is called without synchronizing on `instance`. This is the improved analysis of the lock set for `wait` or `notify` calls.

- Holding n lock(s): `{lock set}`. This message improves the diagnosis of wait/notify problems. It is printed in addition to the old message.

New fields

Some existing classes in Jlint have been extended by new fields. The most important ones are listed here:

Classes `field_desc` and `var_desc`, `struct vbm_operand`

- `const field_desc* equals`: this read-only reference points to the "original" value of a stack element (or any local variable), as outlined in Section 4.3.1.
- `const_name_and_type* name_and_type` (only in class `field_desc`): in order not to lose the type of the values tracked via `equals`, this extra pointer was needed.

Class `method_desc`

- `Locks locksAtEntry`: keeps track of existing locks in order to cut down the redundancy in the call graph.
- `const field_desc* is_this` (argument to `parse_code`). A special `equals` reference that is unique for each analyzed class.

Class `class_desc`

- `Locks usedLocks`: After the method analysis, this set of variables is checked for assignments outside monitors or constructors.

4.5 Problems encountered

Extending Jlint was much more difficult than anticipated. The key reasons are given below. This should not be taken as a criticism of the original author's knowledge of program design and C++. Jlint was written in a very short time (one month), and shows many signs of incremental, "historical" growth without a redesign.

4.5.1 Splitting up the monolithic file

Jlint originally consisted only of four files: three header files (one for the Java bytecode mnemonics, one for the warnings, and the "real" header file) and one source file. The latter was almost 5000 lines big, with type declarations, global variables, global functions and all classes put together. This was clearly no longer maintainable, so the first decision made was to partition all classes into separate files.

Because inlined functions were used heavily throughout the code, some classes could not be separated.[2] However, each file now either contains a single

[2]The high usage of inlined functions is also the reason why many header files have no corresponding `.cc` file; in such cases, the class was entirely defined as a set of inlined methods.

class or several classes with a common purpose (e.g. for the call graph, the node and edge classes are in one file). The goal of having many small files which are easier to edit has not been quite met: most of the code is located in the method descriptor class, even in only one function. Without a redesign, this problem remains.

As many dependencies as possible were factored out into separate include files. The main problem was the fact that many classes access each other's members directly, usually in inlined functions. In order to compile this, the compiler has to know the exact size of that class. This requires parsing the entire header file of the referenced class. This is why almost every file includes every other one indirectly. For this to work, the include directives have to be in the correct order - otherwise the nested dependencies cannot be resolved by the preprocessor. The heavy inter-dependencies between all the classes become quite apparent here.

4.5.2 Architectural shortcomings

The two worst problems are outlined here. For more suggestions about how to design a new checker, see Section 6.2.

Heavily interdependent classes

The main source of problems was the fact that all classes are heavily dependent on each other. This is partially given due to the problem structure: a method has to have knowledge about its class, its local variables, and many other properties. Unfortunately, this has usually been implemented by direct pointers to other objects. From there, the necessary information is accessed via `public` variables. This leads to long chains of direct member accesses, which disallow any major changes in the design.

Many dependencies were introduced which were not needed. For instance, each method has a pointer to its vertex (node) in the call graph and also manipulates the call graph directly. Same for accessors and callees. Because there are no container classes for these data structures, this leads to heavily convoluted code which is almost impossible to maintain.

Data structures not separated from analysis

The lack of container classes is similar to this problem: The algorithms are not separated from the components they work on. Each descriptor class (mainly method and class descriptor) performs the analysis itself, instead of merely giving access to the "innards" to an algorithm. This makes each class very long. Above all is the method descriptor class, which contains more than half of the entire Jlint code, most of which in one, huge function (`parse_code`).

Data structures too closely modeled after class files

The Java class (`.class`) files are designed for having a minimal size. Much information is stored in a *constant pool*, which holds, among other information, all variables and method names. This information is rarely needed at run time

(only for reflection classes and exception stack traces); therefore it is not directly accessible within a method.

Because of this, every access to such meta-data is rather cumbersome: the `getfield` operator, for instance, has only the index of the `name_and_type` entry in the constant pool as its argument. Therefore, a field of type `Object` and name `a` is stored in the constant pool as shown in figure 4.3.

```
3) CONSTANT_Fieldref[9](class_index = 5, name_and_type_index = 17)
 5) CONSTANT_Class[7](name_index = 19)
 6) CONSTANT_Utf8[1]("a")
 7) CONSTANT_Utf8[1]("Ljava/lang/Object;")
17) CONSTANT_NameAndType[12](name_index = 6, signature_index = 7)
19) CONSTANT_Utf8[1]("Equals")
```

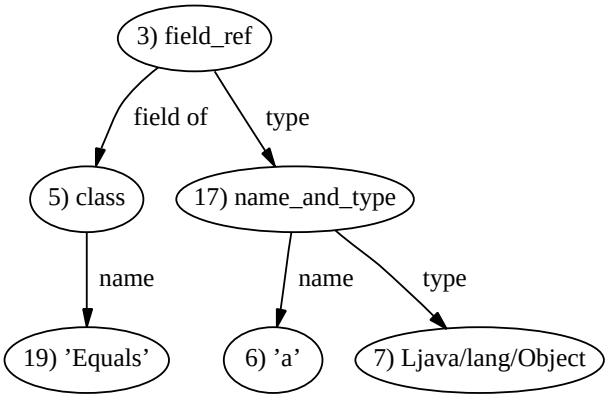

Figure 4.3: Constant pool entries for a field.

The entries in the constant pool look rather convoluted; however, once their dependencies are visualized, once can see that this is an elegant and efficient way of storing that information. In Jlint, all the indirections involved in obtaining the full "identity" of a field have to be taken for each `putfield` or `getfield` command which accesses a field. This makes the code rather clumsy, as shown in Listing 4.2.

Listing 4.2 Code snippet from Jlint: getting the field context.

```
const_ref* field_ref = (const_ref*)constant_pool[unpack2(pc)];
const_name_and_type* nt =
  (const_name_and_type*)constant_pool[field_ref->name_and_type];
const_utf8* field_name = (const_utf8*)constant_pool[nt->name];
const_utf8* desc = (const_utf8*)constant_pool[nt->desc];
const_class* cls_info = (const_class*)constant_pool[field_ref->cls];
const_utf8* cls_name = (const_utf8*)constant_pool[cls_info->name];
class_desc* obj_cls = class_desc::get(*cls_name);
field_desc* field = obj_cls->get_field(*field_name);
```

All this code is needed because Jlint's data structures only map the class files into memory, rather than building a higher level data structure. This code is very difficult to understand (all those direct member accesses yield integers, which are the index in the constant pool) and error prone. It makes further

analysis of field accesses more difficult than needed.

4.5.3 Implementation problems

The design problems described above lead to many difficulties during the implementation of the extensions. There were also some implementation details that made the work harder than necessary.

Lack of comments

The entire Jlint code was very sparsely commented. Usually the only comments were about the purpose of each class and the meaning of some constants. The functionality of methods, for instance, was usually undocumented. Therefore, finding out the correct usage of each function (including the exact meaning of each parameter, the context in which it should be called) was usually quite a bit of guesswork and trial and error.

Sometimes, one variable was duplicated, because it was not always apparent that it existed elsewhere (e.g. in a superclass or in one of the many fields of a member of a class). Of course such redundancy was removed whenever it was found. This could probably have been avoided by a short introductory document to Jlint's classes. Such a document is too often omitted when writing software due to time constraints.

Huge main loop

The `method_desc::parse_code` method contains practically the entire analysis. It does not only parse the code, but it also performs the entire local analysis and builds up all the intermediate data structures. The sheer size (almost 2500 LOC) and complexity of that single function makes it very hard to get into the code. While much of the function consists of simple cases in a large `switch` statement, some analyses go far beyond that. The Jlint extensions aggravated this problem even more.

Call graph extension

There were some difficulties about how to insert an edge into the call graph correctly. Because each method keeps one pointer to a node in the call graph and inserts any edges directly into the global call graph data structure, the procedure is very error-prone. Also, it was not quite clear how to ensure that such edges were marked as "potentially dangerous" with respect to multithreading. As Konstantin recommended, one auxiliary function to build up the call graph, which is normally only used after the methods have been processed, was called within the main (parsing) loop to achieve that effect.

This made the block analysis work – at first. However, at that time, `TYPE.NAME` was used for the pseudo class name rather than `OWNER.NAME` (see Section 4.3.3). Many locking variables are of the same type (often type `Object`), so the former naming scheme did not suffice to distinguish between all the locks. Because no two variables, even of different type, may have the same name, and because their type does not matter for locking purposes, this change did not cause any new problems.

Memory management

The extended analysis required the maintenance of a much larger context information until the end of Jlint's execution (for the final call graph analysis and printing warnings). This required "recycling" some local information, so it could be stored for later. A string pool class was created as a "limbo" where such strings could continue to exist until the entire analysis was over. That string pool class does not implement a full memory management; for instance, for the purpose of Jlint, no reference counting or garbage collection was required.

Lack of `const` usage

This is a minor issue: the code could have been made safer (and also faster) by using `const` to denote read-only references. In some cases, it would also have been easier to understand the code. Using this identifier often is a more recent coding convention which has not yet been adopted by all C++ programmers.

Context (lock sets)

When analyzing a method, the set of locks at the entry of a method was originally "inherited" by the current lock set of a method. However, this sometimes leads to false warnings, and needlessly clutters the call graph.

```
public class Example {
  Object a = new Object();
  public void foo() {
    synchronized (a) {
      bar();
    }
  }
  public void bar() {
    baz();
  }
  public void baz() { }
}
```

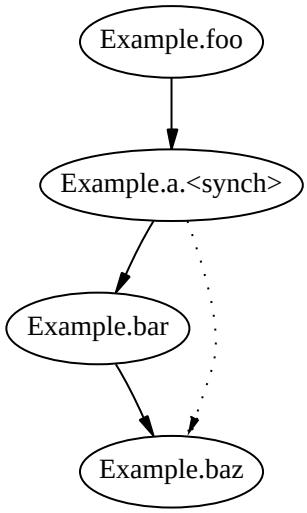

Figure 4.4: Call graph extension for `synchronized` blocks.

Figure 4.4 shows an example of this case. The edges to and from `Example.a.<synch>` are created during the local analysis, when the `synchronized` block and the method call to `bar` are encountered. If another edge from `Example.a.<synch>` to `baz` were be added (the dotted line), the transitive closure would not change, but the call graph would be denser and take more time to analyze. This would have been the effect of including the set of locks held at the beginning of each method (`locksAtEntry`) into the current lock set, whenever a method is analyzed.

Instead, `locksAtEntry` is used to ensure that such redundant edges are not added to the call graph (even if another `synchronized(a)` block were found in the method `bar`). The edge `bar` → `baz` is added during the global data flow analysis, by existing Jlint code, which ensures having the full transitive closure.

Although it would have been possible as such to insert edges for any method call within a `synchronized` block, calls to methods of other classes than the current one are now ignored. This means that the locking context across classes is not investigated. In the current Jlint version, the call graph filled up too quickly, and the analysis suddenly took minutes rather than just seconds. Whether this is due to the NP-hard nature of the call graph analysis or a flaw in the call graph extension scheme, could not be determined. In any case, this part of Jlint needs to be rewritten.

4.6 Application of the new Jlint

After the extensions were implemented, Jlint was run on the examples described in section D.1 on page 89. A detailed breakdown of the output and a comparison to the other tested checkers, in particular the old version of Jlint, can be found in Appendix E.2. Jlint now successfully checks three more programs, while it produces a new false positive for the ESC/Java example (Listing D.8). This makes Jlint as effective as ESC/Java (one case where ESC/Java failed was due to a bug in its theorem prover; the algorithm as such would have been capable of finding that fault).

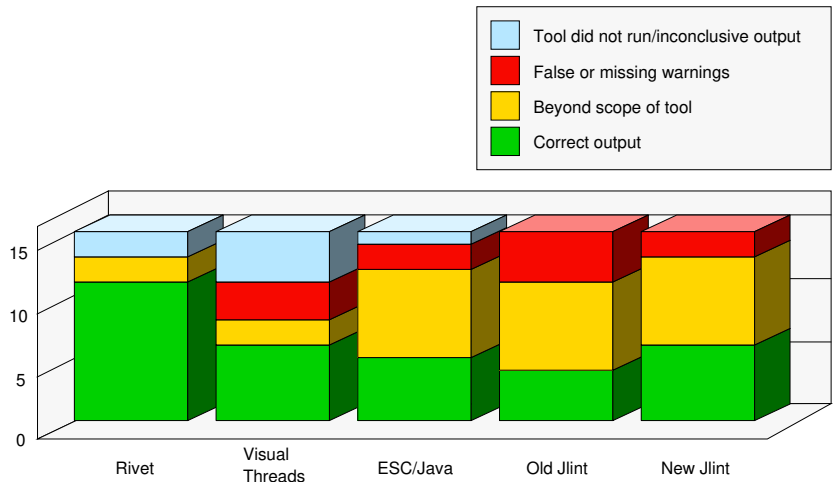

Figure 4.5: Test results for the 15 given examples, including the new Jlint.

4.6.1 Test examples

Old examples

As expected, Jlint now successfully detects the deadlocks in all deadlock examples, specifically in D.1, D.3 and D.5. This is a direct consequence of sup-

porting `synchronized` blocks. Jlint still fails to detect the unusual race condition found in example D.6, and now issues another spurious warning in the ESC/Java example (D.8). In a previous version, Jlint had simply ignored the difficult part in the code (which has deliberately been designed to fool static checkers).

Besides finding three more faults, Jlint also no longer prints a spurious warning for example D.16, where `wait()` was called inside a `synchronized (this)` block. Therefore, the test examples already indicate quite an improvement: 7 rather than 4 cases pass successfully, while a fifth case now fails (because it only passed by luck previously). In simple numbers, this is an improvement by 75%! The seven complex examples – shared buffer and Dining Philosophers – are still outside the scope of Jlint.

Additional test examples

In order to ensure the correctness of the extensions, a couple of new examples were made to test Jlint. These examples all passed successfully in the new Jlint, while the old version failed to detect the faults therein or reported a warning for the wrong reason.

4.6.2 Trilogy's source code

The detailed results of the analysis can be found in section F.2 on page 124. Because the number of warnings was very high (usually in the hundreds), several categories of warnings had to be filtered out. This is because Jlint still lacks some crucial functionality, such as tolerating shared read locks.

Table F.5 summarizes the result. All the warnings about possible lock variable changes – one very specific extension of Jlint – were false positives, since these changes were only made during initialization. This is easily verified manually, and usually beyond the context of static checking. Jlint also reported 12 missing `super.finalize()` calls; these cases all require code changes, at least for safety purposes. In one case, a genuine fault was found by this simple check; in the other cases, one relies on the fact that the superclasses do implement a finalizer.

None of the deadlock warnings, neither those previously present nor those from the extensions, were actual faults. Because the analyzed packages were already fairly mature, it was not to be expected to find such a fault. However, Jlint detected a genuine race condition in a class that was used for debug output.

Interestingly, Jlint was most successful with `null` pointer checks. It reported over 20 cases of unsafe use of parameters and six cases where `null` pointers would likely be dereferenced under some circumstances (e.g. if an input string did not have the right format, some pointers to substrings would then point to `null`). Two of these warnings has been confirmed as being related to an error, and some other cases will now lead to extra comments in the code (which is also an achievement).

Out of the "other" category, the "integer overflow" bug is noteworthy. In one module, a hash value comprised 64 bits. It consisted of two 32 bit outputs of a hash function. After the first hash function, the value was to be shifted left by 32 and then ORed with the result of the second hash function. However, the

programmer probably assumed a wrong operator precedence; the extension of the value to 64 bits occurred *after* the left shift by 32. This fault was easily fixed, and (together with numerous `null` pointer warnings) pointed out yet another weakness in a deprecated module. As a consequence of these warnings, that module will likely be removed from the code altogether.

4.6.3 Other source code

Concurrency package

This package is still too complex for Jlint. It pointed out that the lack of a working context switch (as it is used for any statement that transfer control, e.g. `if`, `for`, `return`) can produce a lot of (new) spurious warnings if it is used inside a `synchronized` block. Despite that, there was a slight reduction of total warnings from 197 to 170, due to the better treatment of `wait` and `notify[All]` calls.

Quite a few warnings (over 20) were because of some bugs in Jlint that still remain. Those warnings only occur under specific (complicated) circumstances, which were apparently not too uncommon in that package. The numerous (26) deadlock warnings are hard to verify; but it can be assumed that they are false positives, given more context (which is not usually not available at compile-time). The remaining warnings are all cases were manual inspection could disprove the warnings, and also show that a static checker cannot successfully cover such cases (such as – potentially infinite – lists of locks).

ETHZ Data warehousing tool

Jlint was very successful in this case. Again, the warnings had to be filtered before they were useful. After filtering, only important warnings remained.

Jlint's specific extension for locking variables detected a fault which has meanwhile been fixed. Other than that, it reported a couple of other warnings, only one of which being definitely a false positive. Some other warnings refer to unsafe code (assumptions that super classes do not change or that a certain reference is never `null`) in the best case and potential faults in the worst one. The ratio of `null` pointer warnings that are not just about unchecked parameters is very high here. Possibly algorithmic properties can guarantee the correctness of these cases.

4.7 Summary

Despite many difficulties, all the desired extensions for Jlint could be implemented. Nonetheless, the extensions clearly showed the limitations of the current Jlint architecture, and while it may still be possible to add some small features to the current code base, this is not recommended. Instead, Jlint should be rewritten from scratch, with the insights gained during writing the extensions.

Applying Jlint to various packages revealed at least 12 (confirmed) faults, two in the area of multi-threading. No deadlock warnings could be verified as a fault as yet. Some warnings had to be ignored altogether because the checks are not refined enough. With only some warnings selectively turned off, Jlint

can already provide valuable information about potential trouble spots in a program. This shows that even a simple program checker can provide a great benefit, as long as the implemented checks are reliable and do not generate too many spurious warnings.

Chapter 5

Discussion

This chapter discusses the results of the two major phases of the project:

1. Evaluation of existing tools.

2. Extension and application of Jlint.

It also describes possible uses of current static checkers in software development.

5.1 State of the art

There are two major categories of program checkers: dynamic and static checkers. The former monitor program execution at run time; the latter work "at compile-time" and try to find faults without actually running the program.

5.1.1 Dynamic checkers

Dynamic checkers can build on years of experience and practical usage that go back as far as using `assert` macros and debuggers. Nowadays, more enhanced tools check function usage (profilers) or memory allocation problems. Therefore, the problem of how to monitor a running program is well understood. Still, new approaches are being taken to make this approach scale up beyond monitoring simple variables; MaC is one such project [11]. For checking multi-threading problems, the problem of the non-determinism of the thread schedule has to be solved, which has not been done in conventional dynamic checkers (also see Section 1.2.1).

Once this problem is overcome, dynamic checkers can take advantage of their biggest strength: full knowledge of the program state. There are two ways of eliminating the total dependency on a particular schedule:

1. Exhaustive scheduling.

2. Keeping the (locking) history of a program and reconstructing possible alternative outcomes.

The first approach is taken by Rivet [12]. It has been shown to work well and find all faults in small programs. Still, the overhead involved is usually too high to allow an application of Rivet to large scale software. Because the Rivet project was discontinued two years ago, no further research has gone into the area of potential performance gains. The major problems with keeping Rivet in line with the restrictions of new Java Virtual Machines has proved too much work for a single research group.

The second approach is taken by the other tools. Only VisualThreads [14] supports this fully automatically; for the other checkers, the programmer still has to supply many assertions and code a lot of the checks himself. It is possible that MaC [11], once it supports synchronization statements, can make this easier by supplying a generic template that will work on any Java source code.

While a working commercial tool exists with VisualThreads, it still has the big drawback that it only works on Alpha Unix, on the low level of POSIX API calls. Having such a tool for Java (possibly working non-interactively in the background rather than with a GUI that slows down the checking) would be very useful. Likely this would require patching the virtual machine itself. Despite that, VisualThreads has proved the validity and effectiveness of this approach, because it works very well on a given C example (Dining Philosophers), where static checkers fail because they do not recognize the circular structure of the protocol (the modulo operator used for the array indices).

5.1.2 Static checkers

The merits of static checking are described in section 1.2.2 on page 4. Static checkers can be divided into extended compilers and (more complex) translators of programs into a mathematical model (theorem provers). The latter approach is theoretically more strongly grounded, while the former has a tradition in continuous improvement of compilers.

Although theorem provers have more sophisticated checking capabilities than simpler model checkers, they still fail to capture many algorithmic properties of programs, which has to do with their potentially infinite nature: loops could never terminate, and lists are not bounded in length. Therefore, the low-level, compiler-based approach stands up very well against more sophisticated methods. Indeed, while quite a few tools in that area are available today, many theorem provers are still in their early development stage.

Not very much work has gone yet into checking multi-threading properties of programs. This is surprising, because this area is a traditional weakness of dynamic checkers, and some checks (e.g. for potential deadlocks and race condition) could be very effective if they were refined enough. As for today, no tool fully supports the necessary features for easy, comprehensive checking of race conditions (which would include shared read locking). In the area of deadlock checking, the "identity" (equality) of two object instances is very hard to verify statically. This still leads to many spurious warnings, because a static checker has to assume the worst case to guarantee a certain soundness.

Nevertheless, the two available checkers (ESC/Java and Jlint) are already quite powerful with respect to common synchronization scenarios, if only certain categories of warnings are used. Both tools still produce many spurious warnings when it comes to array bound checks and potential deadlocks. In the latter area, it is still quite hard to supply the necessary source code annotations

to ESC/Java, but work is in progress to improve this. Both Jlint and ESC/Java show that static checkers have a great potential, and are on their way to becoming a standard development tool like `lint` or the `-Wall` switch for the GNU compiler (but going much further than those).

5.2 Capabilities of Jlint

The static checker Jlint was the most promising tool, given the absence of working dynamic checkers with full Java support, and the fact that the other static checker, ESC/Java, still requires many annotations in order to work effectively. Jlint's amazing speed and ease of use make it well applicable to large scale software.

Jlint found two faults in the area of multi-threading: a race condition in a debug class, and a race condition when re-initializing a shared resource. The latter was thanks to a specialized check that was built in after such a fault had been found through manual inspection. It is hard to say whether the given algorithm for deadlock checking is too simplistic or whether the checked software was already mature enough not to have simple deadlocks.

Notwithstanding, it was clear that the race condition detection suffered severely by the lack of read-only notions (and other cases of "slightly unsafe" locking that still produces the correct result). Such checks also require additional information, which is not available in source or object code. Rather than annotating a program, a template-based solution (such as in MaC) is far easier to maintain, and scales much better to large software packages with hundreds of source files.

Another area where Jlint is still lacking is its *alias analysis.* Tracking equivalent values of references across method calls is very hard, and impossible to implement with the current Jlint architecture. Right now, Jlint cannot even solve this problem *within* the same method or class (for instance variables). Because aliases for locks within the same class are commonly not used, the assumption was made that no aliases exist. This is usually, but not always, the case, and may cause Jlint to miss some deadlocks because it cannot resolve the aliased variable.

5.3 Usage of static analyzers in software development

Two directions are possible for using static checkers in a modern software development process: 1) Usage as an (automated) stage prior to (time consuming) integration and reliability testing, or 2) as a code review/debugging tool that points out potential problems, which have to be verified manually.

As an automated tool, an analyzer should (ideally) issue no warning for correct code, or only in very rare circumstances (so this warning can be automatically turned off in future runs once it has been verified). The currently available checkers are not very useful for this purpose. They still produce far too many warnings.

On the other hand, Jlint or ESC/Java can already be very useful for preparing a code review. With only the most reliable warnings activated, they can

give a list of issues that need to be checked during a review. Furthermore, ESC/Java could also be used with full annotations to formalize comments and verify certain algorithmic properties in small, complex packages.

It is more likely that future work will proceed in the area of tools that are used manually, either as a tool that improves the capabilities of debuggers (or just gives additional information as an independent tool) or helps to spot potentially faulty code. In the latter category, quite a few tools already exist today that find *coding convention violations*, which are not actual faults but may still cause problems with software maintenance. Some of these checks are also included in Jlint (almost all inheritance checks fall into this category).

5.4 Summary

It has been shown that simple static checkers can already cover a large number of potential faults. While complex static checkers are more powerful, they also fail in their analysis of complex data structures, which reflect a potentially unbounded complexity of the algorithm.

Extended dynamic checkers with specializations for locks have a great potential; but so far, no working, publicly available implementation exists. Because of this, the decision has been made to use Jlint, a fast data flow analyzer. In its given state, some of its checks still need refinements to be truly useful. Other checks are already quite mature and have successfully been applied to large scale software packages.

Because static checkers still generate a large number of spurious warnings, an automated usage is still far from realistic. However, as a verification step prior to testing or a code review, static checkers can already enhance the software development process today.

Chapter 6

Future work

In this chapter, an outlook is made on possible extensions in the area of dynamic and static checkers. Immediate Jlint extensions, which could improve its usefulness a lot, are described first. Because Jlint should eventually be rewritten, design guidelines are given to create a new static checker, which will be more extensible and maintainable than Jlint. This chapter concludes with some ideas about how dynamic and static checkers could be improved or even combined in the future.

6.1 Future Jlint extensions

The list below shows the most important steps that need to be taken for improving Jlint's capabilities for analyzing multi-threading problems. Of course, this list is incomplete and would likely be extended once some of the listed features were implemented. The priority of each item (1 highest, 3 lowest) is given in brackets. Some extensions are far more difficult to implement than others; in particular, those that require an exact context across methods or classes require a global data flow analysis. General usability issues such as reducing the total number of spurious warnings are not shown here: Such issues are commonly known, but it is not yet known how to implement a solution.

- (1) Restructuring/rewriting of the data structures: Some classes are poorly designed, and in general the given framework is inadequate for a more powerful analysis (such as the analysis of lock sets when calls from one class to another class are made).
- (2) Context merging (`return`, exceptions, `if`, `while`, `for`) to eliminate some spurious warnings. See section 6.2.3.
- (2) Suppressing unnecessary warnings for variables and methods that are read-only. A resource that is shared for reading is a very common scenario in real software. Therefore, Jlint should suppress a warning about concurrent variable or method accesses if the following conditions hold:

  Shared-read variables: The variable is not accessed outside the constructor or `synchronized(this)` blocks.

Shared-read methods: The method does not access fields that are not shared for reading (according to the definition above).

- (2) Adding a check for "`const` methods" (as in C++): In C++, a method which is declared `const` may not change the `this` instance or any `const` (`final`) fields. As a consequence of that, calls to non-`const` methods and the usage of `const` fields as non-`const` arguments (in method calls) are forbidden. This is one of the few areas where C++ is more type safe than Java. Adding such a check to Jlint (`const` annotations could be stored in a template) should not be very hard, and would be a minor extension of "shared-read methods" (see above).
- (2) Full analysis of lock sets across method/class boundaries for `wait` and `notify` calls.
- (2) Checking the visibility of object instances in other threads: A non-`synchronized` method where the callee instance is not visible outside the current thread (e.g. as a local variable) can be called without any danger of race conditions. In such cases, a warning should be omitted. Verifying this requires evaluating accessor information.
- (2) Fixing the fault in the warning selection switches: right now, switching warnings on and off based on their message code produces an empty output.
- (2) Improving verbosity/information content of some (existing) warnings.
- (2) Adding an improved selectability for warnings, e.g. by a severity level or likelihood.
- (3) Support for locks on classes (`getClass`) in synchronizations.
- (3) Eliminating spurious warnings where implicit synchronization exists (e.g. streams).
- (3) Texinfo documentation: fixing old spelling and grammar mistakes from version 1.11, better Texinfo tagging of text (e.g. `@var`), full `@node` tree for document.

6.2 Design of a compiler-based analyzer

During the implementation of the Jlint extensions, many limitations were found in the current architecture of Jlint. Some of them are design flaws that should be avoided for any software, others were limitations that apply specifically to program checkers or even Java bytecode verifiers.

6.2.1 Easy access to meta data

For all the crucial information about fields (or methods and classes), their descriptors should offer direct accessor methods for these properties. These include:

- name and type
- for methods: parameters, signature, lock sets
- for fields: set of possible values or range, ...

This enumeration is of course incomplete. It is obvious that the different descriptors should encapsulate that information in an API rather than only having index entries to a global data structure. Such an encapsulation will make the data flow analysis much more extensible (e.g. a possible set of values could be stored along with a possible range; the set of possible values would be set to "any numbers" or `*` once it grows beyond a certain bound).

This would be a first step towards an improved aliasing analysis. Of course, having the design that allows keeping track of equivalent references does not solve the problem yet; but it would at least make it very easy to cover (and prove) the simple cases, where no alias exists (private instance or class variables).

6.2.2 Resource management

A problem that was not apparent before is the fact that much more information has to be retained until the end of the first pass (reading the class files). Many analyses require the entire call graph, as well as more detailed information about the calls. Jlint was built with strictly partitioned local and global analyses, and only certain strings (source file names, line numbers) were kept in memory for printing the warnings.

The extended analysis of `synchronized` blocks required the availability of field information until the end of the first pass, and the field names until the end of the second one. This requires its own, small memory management. The string pool was a first, simple step towards this, fulfilling the current needs. It is probably not adequate for Jlint extensions, which may require a more sophisticated memory management (e.g. reference counts).

This problem suggests Java as an implementation language for a checker, because its built-in garbage collection solves this problem. In C++, a similar functionality would have to be implemented first.

6.2.3 Full control flow analysis

Jlint includes an implementation for a full control flow analysis (context splits and merges for control flow statements such as `if`, `for`, `return`). Unfortunately, this implementation is very obscure (because it is not commented and involves `memcpy` calls with certain offsets rather than explicit copies) and does not seem to work fully for the local analysis.

A simplified control flow analysis with only forward jumps, such as found in `if/else` blocks, would suffice (rather than fully implementing backward jumps, which occur in loops). Full support for backward jumps would include counting the number of jumps (for avoiding infinite loops); unrolling a loop for a given number of times is much simpler, and commonly used in other analyzers.

This feature would avoid a lot of spurious warnings about freed locks, which only occur because a `monitorexit` statement can occur several times

for only one corresponding `monitorenter` operation. It is automatically "embedded" by the compiler prior to a `return` or `break` statement, in order to release the lock in any possible case. The following code snippet demonstrates such a situation.

Listing 6.1 Extra `monitorexit` operations inserted by the Java compiler.

```
synchronized(lock) {
  if (resource == null) {
    // compiler will insert "monitorexit"
    return;
  }
  /* do something */
} // "monitorexit" for closing bracket
```

6.2.4 Combining the lock and call graphs

The call graph extension described in section 4.3.3 is one way to include `synchronized` blocks in the call graph. Possibly this is the best way, since it allows for nested synchronizations combined with method calls. It fails, however, to include a more advanced data flow analysis, where the interaction with another object results in a reference which is used as the lock variable.

For this reason, the call graph also has to be built in several passes (also see section 6.2.7 below). This report shows one way to include `synchronized` blocks, but it is not necessarily the most elegant way. For example, ESC/Java does the reverse: it treats `synchronized` methods as normal methods with a `synchronized(this)` block spanning the entire method. This helps to separate the call graph from the lock graph.

6.2.5 Separating analysis from data

Not separating the descriptor classes from the graph and code analyses was a violation of a basic design principle. The graph generation, analysis and output is scattered over several classes, including some where it should definitely not be. For instance, the graph data structure is directly manipulated in the method descriptor class. This was probably a result of the amazingly short time in which Jlint was developed. Regardless, this drastic design shortcut leads to many errors during development, and makes it hard to maintain the program.

6.2.6 Maintain rich context for output

Jlint has been written with a fixed set of checks in mind. As many resources as possible are only stored locally to one method or one class, and they are explicitly replicated if they may be needed for warnings later on. For now, such copies include strings (such as file names) and line numbers (in the callee descriptor class). In such cases, a richer context should be stored for later analysis. This would not only a potentially improve a latter analysis of the found violations (and maybe eliminate some spurious warnings), but also allow a more detailed output.

For instance, a call graph including the lock sets or a possible execution trace would greatly improve the usability of a static checker. Of course, implementing these feature is far from trivial.

An indication of the likelihood of a warning would also be very helpful. This could either be done numerically or (probably better, since numbers will not be accurate) verbally, e.g. "certain fault", "probably fault", "unlikely fault". The more assumptions had to be made during checking, the less likely a warning is correct. Having the full context for a warning would help to determine the "confidence level".

6.2.7 Multi-pass call graph analysis

Jlint now only performs two passes. Most checks are done in the first pass, and a call graph analysis follows for finding potential deadlocks. By performing other types of analysis at a later stage (after the call graph is built), one could take advantage of that extra context. This requires storing the context at the beginning of each method.

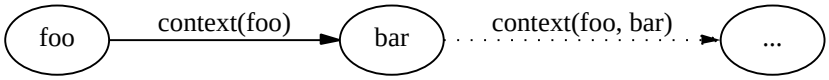

Figure 6.1: The problem of propagating the context: The context of each method can only be propagated by one recursion level in one pass.

If one separates the state of a method from the call graph, a difficult problem remains: For building up the context at the beginning of a method (e.g. the lock set or the possible states of instance variables), one can only cover one level of recursion at a time. For example, a method `foo` that acquires a lock `a` will cause the checker to store that information while parsing the method. If another method `bar` is called while holding the lock, this context cannot be included in `bar` before the entire call graph is built up (because the method `bar` might already have been parsed, with only its local context, when `foo` is read). Further recursions (method calls from `bar` in this case) will require further passes on the extended call graph. This is a major disadvantage of building up the call graph with extra state information rather than using the extended call graph with pseudo method calls.

6.3 Future directions for formal analysis

This section outlines some possible future directions for static and dynamic analysis. It is of course hard to predict future development in a field where progress is made so fast.

6.3.1 Dynamic checking

Scope

Dynamic checking for multi-threaded programs works as such – VisualThreads demonstrates this, although it is not yet very effective for Java programs. Still,

performance is a huge problem for any dynamic checker. A possible direction is the implementation of a virtual machine whose goal is not fast code execution, but a small state and easy verification of aliasing. This should facilitate and speed up dynamic checking. The Rivet project has gone into that direction, but it has not arrived at its goal.

The MaC approach is a very effective way to monitor a large body of code. Its template-like approach allows a convenient customization of the checks and an easy inclusion of many fault categories. Once its performance bottlenecks are taken care of, support for synchronizations and a lock detection algorithm can be added to it. Work in this area is already in progress.

Usage

The usage of dynamic tools is likely to remain as it is now: once the software is fully functional, it is tested with a dynamic checker. Today such tests are restricted to memory allocation and access problems. In the future, they will likely be extended to multi-threading problems. It is unlikely that this prevents *static* checkers from catching on, as they can already be applied before the software is finished.

6.3.2 Static checking

Scope

The current static checkers are still rather weak when it comes to multi-threading problems. This should be changed as soon as possible. In fact, this area could become a key strength of static analysis, because certain multi-threading problems are almost impossible to reproduce. Software development may even reach a stage where writing multi-threaded software would be impossible without the help of verification tools.

For now, the modeling still has to be improved a lot. Some extensions are even rather straightforward (e.g. shared read scenarios), so it is rather surprising that they have not been implemented yet.

One crucial question for a static checker is whether it should work on (possibly instrumented) object code or source code. Byte code has a simpler structure and fewer possible commands, which makes it easier to translate into a model [42]; source code, on the other hand, allows for an easy inclusion of annotations.

Usage

A conservative static analyzer could be an excellent code review tool. Right now, eliminating false positives still requires a lot of human intervention. Refining the analyzers will certainly cut down on false positives. Once analyzers have reached a level where usually not more than one warning per class file is displayed (and that warning is easy enough to verify, maybe with the aid of graphical tools), it can even be included in the standard development tools that are used – maybe as a stage after compilation?

An automated use is still problematic. As non-trivial checks will likely always produce a few spurious warnings, a mechanism has to be in place to mark

those as "safe". Jlint already has on option (`-history`) to eliminate warnings it has printed before. But what makes it certain that the warning is not justified after the code has changed? Possibly, storing the full context of the warning can help to avoid a scenario where a warning is suppressed even though a fault was introduced by a code change.

It is also possible that powerful static checkers that require more information (via annotations or templates) will be applied to algorithmically complex subsets of software packages, while simple and fast checkers perform more coarse checks on entire packages. For a widespread usage, both kinds of tools still have to become easier to use. Especially the verification of a warning has to be made simpler, maybe with improved code inspection tools.

6.3.3 Combination of both approaches

Both static and dynamic analyzers usually require extra information, e.g. source code annotations (assertions) or templates that define properties across several files. It is desirable that a common denominator is found across tools of both domains, so *one* annotation could be verified both statically and dynamically, by two different tools.

When automatic model generation becomes more sophisticated, it may even be possible that a static analyzer or a modeling tool *instruments* the code for a dynamic checker. For example, a static analyzer could suppress assertions and other checks where the correctness of a property can be proved; where this is not possible (because the dynamic state cannot be predicted), it could insert extra code to verify that property.

6.4 Summary

Dynamic checking for multi-threading problems can be successful, but no implementation that is suitable for Java is available yet. Moreover, dynamic checking still suffers from a high run time overhead.

Currently available static checkers could be much more useful by including more rules in the model that is verified. Centrally to such improved checks are common optimizations in software, such as shared reading without locking.

Jlint has shown that the usefulness of a static checker also depends on its architecture, not only on its algorithms. Some design guidelines have been given about how a more flexible static checker could be written. These guidelines are a result of the insights gained when extending Jlint.

Finally, some ideas about possible directions in both areas were outlined. Both static and extended dynamic checkers have a largely untapped potential for checking multi-threading problems. It can be expected that much work will be done in these areas in the near future.

Chapter 7

Conclusions

Multi-threaded programming poses a new challenge: the behavior of multi-threaded programs is non-deterministic, because the thread scheduler cannot be controlled by the application. This puts severe limitations on classical regression testing: even for the same test case, the result cannot always be reproduced. Two promising new approaches are *extended dynamic checking* and *static checking.* The former technique employs new ways of overcoming multi-threading problems; the latter analyzes a program at compile-time.

Extended dynamic checkers can either explore the state space of all possible threads or keep track of the program history in order to analyze the full capability of a multi-threaded program. The latter approach proved more practical, although there is no tool available yet that efficiently performs such checks for Java programs.

Static checkers can be divided into (usually simpler) model checkers and more complex theorem provers. The former, despite their limitations, have proved to be almost as effective as the latter, especially in the area of multi-threading problems. As the technologies are merging, the distinction becomes increasingly difficult. Any static checker which is available today still needs further work.

Because Jlint is fast, easy to use, and well applicable to large scale projects, the decision was made to extend its applicability to include `synchronized` blocks. As statistical analyses showed, this was a significant improvement and allowed Jlint to check a larger program base effectively. During the extensions, it was also seen that a clean architecture is as important for a good static checker as good algorithms are.

The extended Jlint was applied to Trilogy's source code and a few other software packages. It discovered two multi-threading faults and ten other faults, mainly potential `null` pointer references. Many multi-threading checks could be more useful with more refined static analyzers. Even at its current stage, Jlint can already be a very useful tool for finding faults early or for preparing a code review.

In the last chapter, some ideas for future extensions of Jlint and other static checkers were presented. Both static and dynamic checkers still have a largely untapped potential, especially in the area of multi-threading problems.

Appendix A

Source code analysis

This chapter lists the results of all source code analyses. First, a short description of the tools which were developed for this specific task is given; then, the results of applying these tools to various source code bases are shown.

The sources have been chosen from various domains, in order to ensure a good coverage of different types of programs. In Trilogy's packages, the complexities lie in the algorithms and I/O; in the CORBA and concurrency packages, I/O and synchronization issues are dominant.

	LOC	synch. statements	synch./KLOC
Trilogy	284875	487	1.710
Java	290428	1156	3.980
Javax	245700	217	0.883
CORBA	21180	56	2.644
Concurrency	23782	531	22.328
ETHZ DW	24649	147	5.964
Total	890614	2594	2.913

Table A.1: Overview about each package.

Table A.1 shows an overview about each analyzed package. As a first, simple metric for the "parallelism" present in the packages, the number of `synchronized` statements per thousand lines of code (KLOC) is shown. As comments and code layout (e.g. number of empty lines) may slightly distort the statistics, the numbers have to be taken as an approximate measure. Nevertheless, due to the large code size, they are certainly comparable, and there is no doubt that the concurrency package is, due to its very nature, the most complex one when it comes to parallelism.

More about the different cases of synchronization that were investigated can be found in Section 3.5 on page 28.

A.1 Analysis tools

Since no existing tools analyze parallelism in the way that was needed for this project, a customized suite of tools, consisting of five shell scripts and one Perl

script, were written.

The first script, `source-stats.sh`, is a stand-alone script that gives an overview of the package. It analyzes all `.java` files in the local directory tree. Because the "textutils" proved, due to their context-insensitive nature, to be slightly inadequate for removing all C style comments from the Java sources, the C preprocessor `cpp` was used for that task. While this adds some overhead to the script, it ensures a correct removal of any comments. The rest of the script is mainly a construct of nested `grep` and `sed` commands. All in all, several hundred files and processes are created and destroyed during execution of this script. However, due to the efficient file caching and process creation mechanisms in Linux, the script still runs within a few seconds on a fast computer, eliminating the need for any optimizations.

For a closer analysis of synchronized blocks, `variable-type.pl` was created. It parses any Java source file, removing comments and analyzing the block structure. Whenever it encounters, for the current file, a given variable within a `synchronized` block, it prints out what type of variable that variable is. For a distinction of the different cases and their relevance, see section 3.5. If the variable is not declared at that time or further down in the file, it recursively parses all superclass files, searching the directories in the `CLASSPATH` environment variable. For class and instance variables, the program also counts the number of assignments to that variable in the current file (though not recursively in the superclasses).

Initially, the Perl script was much simpler and had several shortcomings (e.g. it could not parse variable declarations occurring after the `synchronized` blocks and did not work recursively on superclasses). The elegance of the script has definitely suffered under all the extensions of the ten revisions made so far, but since the script will not be needed for later work, it was not rewritten.

Because `variable-type.pl` has to be called once for each shared resource and each source file, two shell scripts, `analyze-file.sh` and `analyze-package.sh`, perform the necessary preprocessing on a file or directory level. The latter script writes its output to `<directory name>.details`, a file that, while being human readable, still contains too much information to be of direct use. Two more scripts, `total.sh` and `total-of-a-kind.sh`, parse that output file and give a total count of all variable types of each category.

Additionally to these analysis tools, the following shell script was sometimes used to generate a call graph such as in figure 4.1 on page 39 (Jlint has to be compiled with `-DDUMP_EDGES`):

```
echo "digraph lockgraph {"; jlint *.class | grep ^Call | \
sed 's/Call graph edge //' | sed 's/([^)]*)//g' | \
sed 's/\(.*\) -> \(.*\)/"\1" -> "\2"/'; echo "}"
```

The output of the script is the input for `graphviz` [25, 54].

A.2 Trilogy's source code

A.2.1 Introduction

Because Trilogy's sources which were analyzed did not consist of one, monolithic package, it warrants a breakdown into its modules. Table A.4 shows a

list of all the seven packages (denoted by the names that are used in the source repository). Despite the variations between the different packages, only the diagrams for the totals are shown. Indeed, only two of the smallest packages significantly deviate from the others in complexity.

	LOC	synch. statements	synch./KLOC
catalogsvc	5661	5	0.883
ffcaf	46994	161	3.426
hec	23947	12	0.501
sbjni	19031	32	1.681
scbbjava	129327	165	1.276
tce2	31242	36	1.152
trilogyice	28673	76	2.651
Total	284875	487	1.710

Table A.2: Overview of Trilogy's source code.

A.2.2 Description of analyzed packages

This is a very short description of the analyzed packages, and is by no means complete or comprehensive. It may help the reader to see in which areas the problems lie when it comes to concurrency.

MCC catalog (catalogsvc)

This is a wrapper for the "Trilogy Classification Engine" (tce):

> "The Classification Service provides remote access to the TCE in the Multi Channel Commerce framework. It provides lightweight versions of its classifiers which understand how to talk to the service. Using these classifiers, TCE may be used remotely in a manner indistinguishable from local access. Also contains Modules, Controls, etc."

MCC Core (ffcaf)

The MCC Core handles the following functions:

- wrappers, load balancing and failover of remote services (CORBA/RMI)
- XML/JNDI configuration and tools
- module manager and module event handling
- cache semaphores for the registry service (pause/unpause services, cache flushing)
- web site finite state machine code, custom TWC dispatcher, custom TWC beans
- ACL abstraction layer for UserACL service(s)

Cerium (hec)

This is a re-implementation of SalesBuilder in Java, which was originally written in C++. Its core is an engine that allows configuration modeling to be implemented in Java. As such, the program does not use a lot of multi-threading.

SalesBuilder (sbjni)

This package is a Java wrapper around the SalesBuilder, Checker, and Pricer engines. These engines, when used together, can help a company to track sales, check and define price structures, and update them based on sales. Out of this package, the LogPlayer was analyzed. It can read a log file and recreate the scenario that lead to such log entries. For this purpose, it starts and controls other applications as needed.

Java backbone (scbbjava)

This is a Java-based object-oriented abstraction layer for relational DBs. As such, it also manages sessions to the DBs.

Trilogy classification engine (tce2)

This is a generalized classification engine for storage and retrieval of information.

> "The engine ... understands many different kinds of "classifiers" used to organize information. The most common classifiers describe various characteristics of data items, e.g. "Color", "Category", "Price", "Name". The engine facilitates maintaining, searching, and browsing these classifiers. The new engine is schema independent. A classification 'schema' describes how the data is organized in the database, i.e. where the classifiers are. Schema independence greatly increases the flexibility of the engine."

Trilogy Insurance Calculation Engine (trilogyice)

> "Trilogy's Insurance Calculation Engine allows insurers to create rating and automated underwriting models, and compile them to stand-alone Java classes. These Java classes can be used to calculate quotes and underwriting answers."

A.2.3 Overview

Figures A.1 and A.2 show the same result in two different views. From the 487 `synchronized` statements in Trilogy's code, the majority occurs as `synchronized` methods. The number of `synchronized` blocks is almost as big. When breaking down the latter number, a small, but still significant part is taken up by `this` and `this.getClass()`, locks on the current instance and the current class, respectively. However, most cases are more complex than that, involving a lock on a variable other than `this`. For these cases, the ratio of locks on entire classes (rather than instances) is about the same.

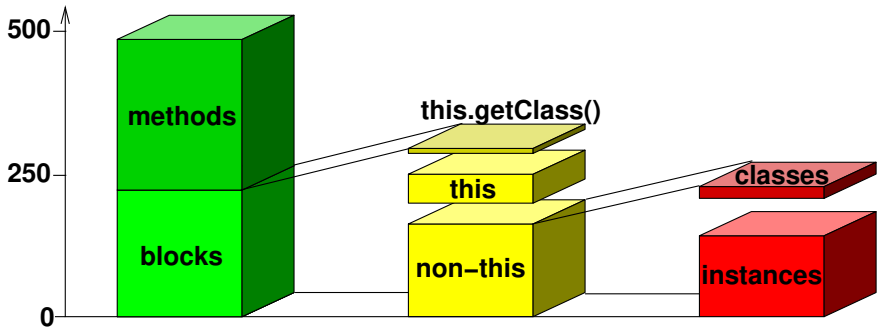

Figure A.1: Breakdown of the usage of `synchronized` statements in Trilogy's code.

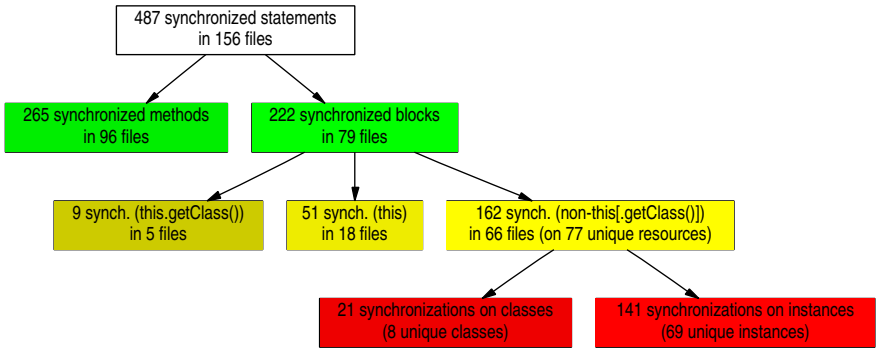

Figure A.2: Statistics of the usage of `synchronized` statements in Trilogy's code.

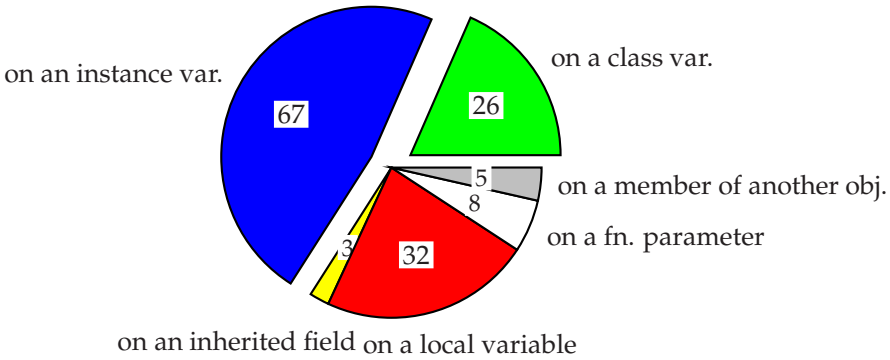

Figure A.3: Types of variables used in `synchronized` blocks in Trilogy's code.

Figure A.3 shows an overview of the *types* of variable that holds a lock in a `synchronized(lock)` statement. As one can see, the vast majority are class or instance variables (which are only initialized once as a shared resource and then used throughout the program; this is not shown in the diagrams).

	class var.	instance variables	local var.	fn. param.	other cases	total
catalogsvc	-	1	4	-	-	5
ffcaf	9	38	5	6	1	59
hec	-	3	-	-	-	3
sbjni	-	11	2	1	-	14
scbbjava	17	17	9	1	4	48
tce2	-	-	12	-	-	12
trilogyice	-	-	-	-	-	0
Total	26	70	32	8	5	141
100.00%	18.44%	49.65%	22.70%	5.65%	3.55%	

Table A.3: Per module usage of `synchronized(non-this)` blocks in Trilogy's code.

Table A.3 gives the numbers per module, where inherited variables were counted towards instance variables. As one can see, the usage of synchronized blocks varies quite a lot between different modules.

The results from the previous tables and figures are condensed in figure A.4. The synchronizations on class or instance variables, and the "other cases" represent the synchronized blocks shown in table A.3.

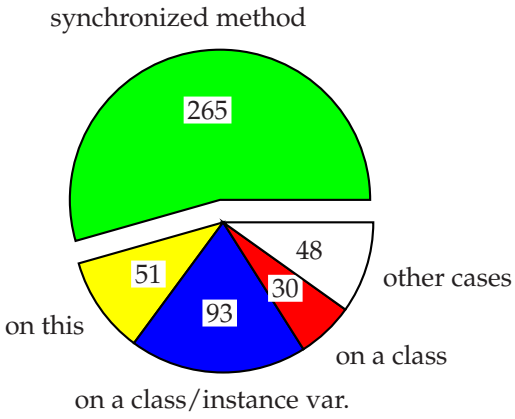

Figure A.4: Overall usage of `synchronized` statements in Trilogy's code.

A.3 Built-in Java packages

As in the previous section, table A.4 gives an overview about the source code of the built-in Java classes. The variations between the packages is surprisingly big; however, the numbers have to be taken with caution, since only the actual `.java` source files, but no native implementations, have been analyzed.

	LOC	synch. statements	synch./KLOC
applet	761	0	0.000
awt	114431	441	3.854
beans	9461	106	11.204
io	23489	145	6.173
lang	29307	100	3.412
math	5325	1	0.188
net	11292	68	6.022
rmi	7667	26	3.391
security	17892	31	1.733
sql	10940	12	1.097
text	23602	17	0.720
util	36261	209	5.764
Total	290428	1156	3.980

Table A.4: Overview of the source code of the built-in Java packages.

The following figures are equivalent to the ones from the previous section. A surprising difference is the amount of inherited fields used: it is almost as high as the number of "ordinary" instance variables. This greatly increases the dependency between a class and its superclass. In these cases, both classes are developed by the same team, so this should not be a problem.

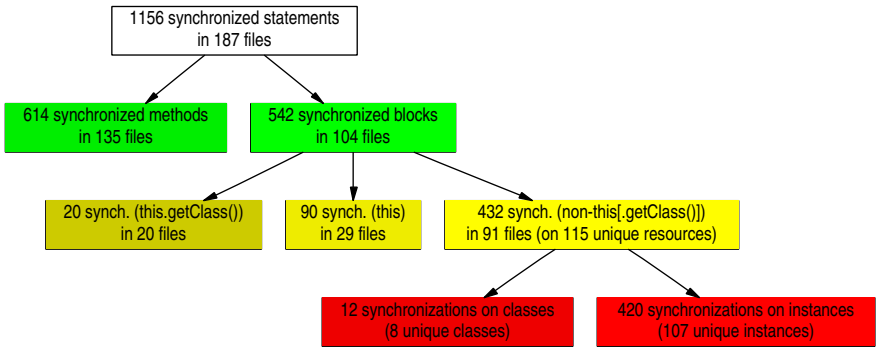

Figure A.5: Statistics of the usage of `synchronized` statements in the built-in Java packages.

In table A.5, inherited fields are listed separately, unlike in the previous section. Also, the "other cases" encompasses fields of other objects and (in the `rmi` package) native fields.

A.4 Other packages

The other packages are monolithic or small enough, so they are not broken down into their components.

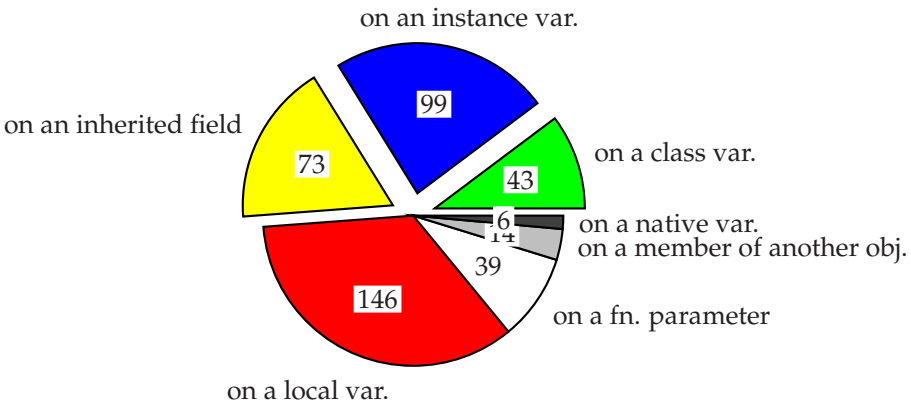

Figure A.6: Types of variables used in `synchronized` blocks in the Java packages.

	class var.	instance variables	inherited fields	local var.	fn. param.	other cases	total
applet	-	-	-	-	-	-	0
awt	3	8	-	134	1	-	146
beans	-	31	6	3	2	11	53
io	2	3	67	-	3	-	75
lang	19	6	-	6	10	1	42
math	-	-	-	-	-	-	0
net	7	1	-	1	3	-	12
rmi	2	-	-	1	-	6	9
security	-	3	-	-	2	-	5
sql	-	-	-	-	-	-	0
text	-	2	-	-	-	-	2
util	10	45	-	1	18	2	76
Total	43	99	73	146	39	20	420
%	10.24%	23.57%	17.38%	34.76%	9.29%	4.76%	

Table A.5: Per module usage of `synchronized(non-this)` blocks in the Java packages.

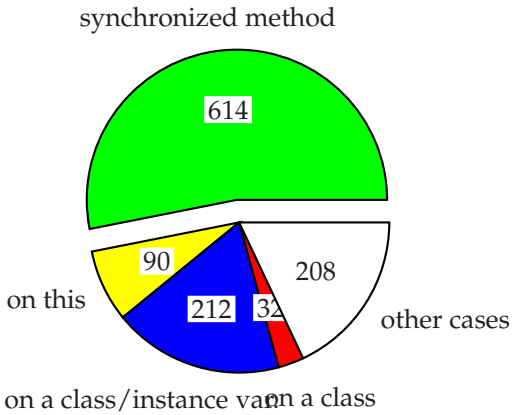

Figure A.7: Overall usage of `synchronized` statements in the built-in Java packages. The blue and grey pie slices represent the synchronized blocks shown in table A.5.

The `javax` packages include `accessibility`, `naming`, and `swing`. `accessibility` has no `synchronized` statements, while `naming` only has four; therefore they were not treated separately. `javax` encompasses 617 files with 245700 lines of code.

The CORBA and ETHZ data warehousing package were both comparably small in size and (parallel) complexity; however, the Concurrency package [23] was surprisingly complex.

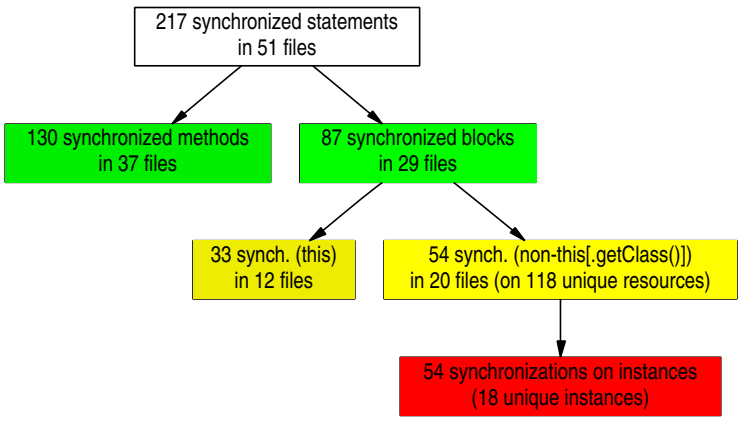

Figure A.8: Statistics of the usage of `synchronized` statements in the Javax (Swing) packages.

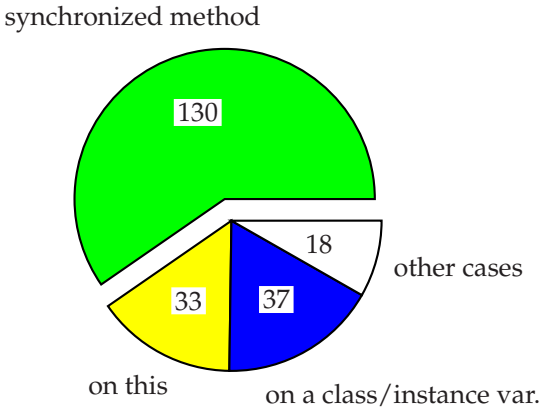

Figure A.9: Overall usage of `synchronized` statements in the `javax` packages. There were 24 class and 14 instance variables. The "other cases" include 16 local variables and two native fields.

A.5 Summary

Even though some variations between different packages were present – especially in the number of `synchronized` statements per LOC – the overall picture is quite homogeneous.

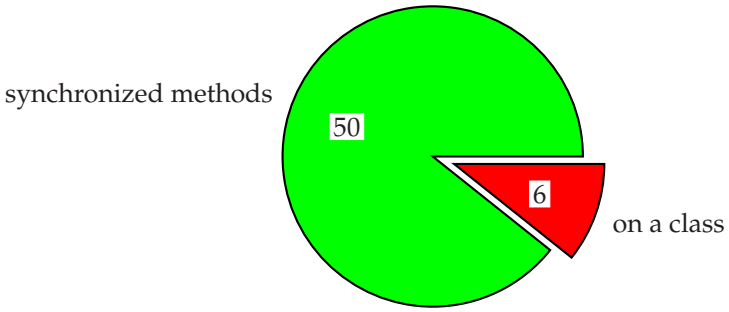

Figure A.10: Statistics of the usage of `synchronized` statements in the OMG (CORBA) packages. These Java CORBA classes (`org.omg.CORBA.*` and `org.omg.CosNaming.*`) were a simpler case; almost all synchronizations are done with synchronized methods. The total amount of code were 21180 lines in 247 files.

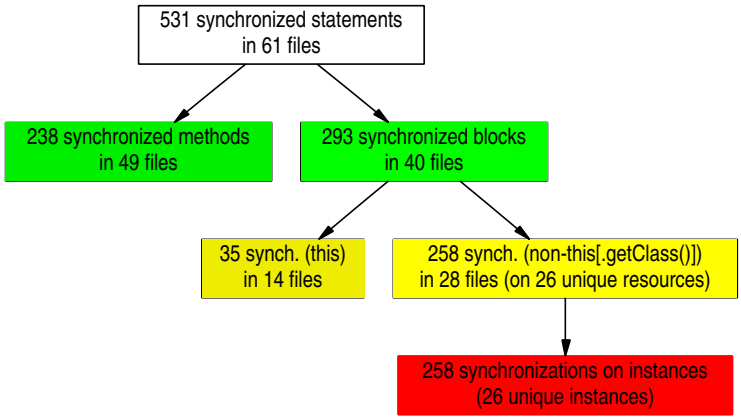

Figure A.11: Statistics of the usage of `synchronized` statements in the Concurrency package [23]. The total amount of code were 23782 lines in 104 files.

Category	#	%
`synchronized` methods	1351	53.76 %
`synchronized(this)`	209	8.32 %
synchronizations on current class	68	2.71 %
synchronizations on other classes/instances	582	23.16 %
other cases	303	12.06 %
Total	2513	100.00 %

Table A.6: Total usage of `synchronized` statements.

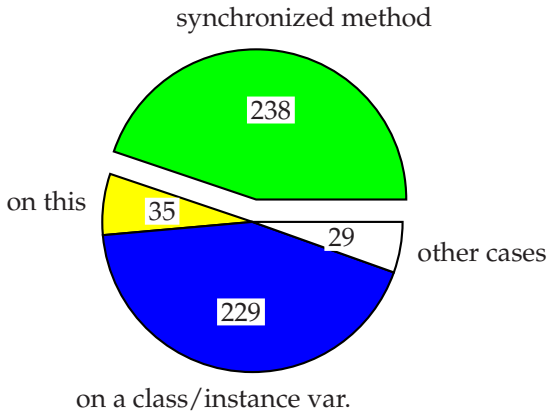

Figure A.12: Overall usage of `synchronized` statements in the Concurrency package [23]. The blue slice includes 3 class and 31 instance variables as well as 195 inherited fields. The other cases are 1 local variable, 8 function parameters and 20 members of other objects.

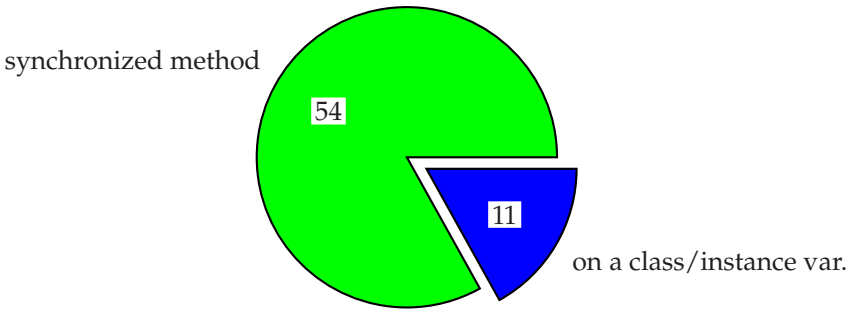

Figure A.13: Overall usage of `synchronized` statements in the ETHZ data warehousing package [24]. The total amount of code were 24649 lines in 147 files. There were 7 class and 4 instance variables.

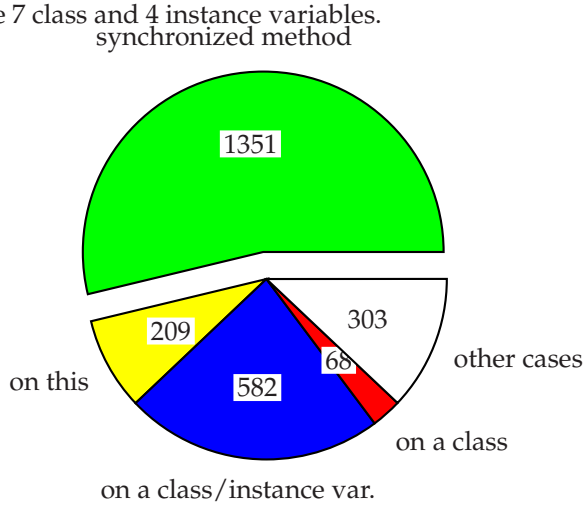

Figure A.14: Total usage of `synchronized` statements in all analyzed packages. A total of about 900000 LOC was analyzed.

Appendix B

Existing tools

B.1 Dynamic checkers

B.1.1 MaC

Purpose: Monitor running systems against a formal specification.

Producer: Real-time systems group (RTG) at the University of Pennsylvania.

Technology: Automatically generated run time checker.

Overview: MaC stands for "Monitoring and Checking". It combines a high-level requirement specification and a low-level monitoring script that verifies the given requirements at source code level. An *Instrumentor* generates a run time checker based on the given data. This checker verifies the given properties after each method call. This process is easier to use but less general (in terms of schedules covered) and efficient than static checking.
The requirements are expressed in an extended form of linear temporal logic; the monitoring script is written in a simple event definition language. Optionally, extra actions for exceptional conditions can be given via a steering script [27, 28].
MaC does not yet have a framework for systematically testing multi-threaded programs, but it could be combined with a special JVM in order to achieve this.

Availability: MaC is available for research purposes, including source code. It is written in Java and platform independent. It requires the JTrek library from Compaq (`http://www.digital.com/java/download/jtrek/`), which is available under a special license.

Usage: MaC has been applied to a couple of small test programs [27].

URL: `http://www.cis.upenn.edu/~rtg/mac/`

B.1.2 Rivet

Purpose: Create advanced debugging and analysis tools.

Producer: Software Design Group at the MIT.

Technologies: Custom Java Virtual Machine (JVM); systematic scheduling algorithm for dynamic testing.

Overview: Rivet is a platform for sophisticated Java debugging and testing tools. Its goal is to expose the internals of the virtual machine in an structured, well-documented way in order to allow the construction of tools based on that information. So far, the initial suite of tools contains a bi-directional debugger (not yet fully implemented) and a tester for multi-threaded programs.
Rivet is a partial JVM running on top of another virtual machine. Therefore, it only implements the key components needed for systematic testing, such as the thread scheduler.
Deterministic replay, which allows the virtual machine to undo a step in its execution, was planned as a future extension. This would not only allow more efficient testing, but also allow the creation of a new, very powerful debugger. A couple of challenges, such as efficient representation of classes and instances in the virtual machine, still had to be overcome. Performance problems were likely a reason why development on Rivet was discontinued.

Availability: The professor who managed the project has left the MIT on spring 1999. Work ceased on both Rivet and the ExitBlock systematic testing algorithm.

Usage: Rivet has been used on a couple of test examples and small programs, but did not scale to large software [29].

URL: `http://sdg.lcs.mit.edu/rivet.html`

B.1.3 Verisoft

Purpose: Systematically test multi-threaded applications written in any programming language.

Producer: Bell Laboratories, Lucent Technologies; main author: Patrice Godefroid.

Technology: Systematic state space exploration (in dynamic checking) using a new search algorithm.

Overview: Verisoft is a dynamic checker that allows the programmer to systematically explore the *state space* (state of all variables and interleavings of threads) of a program. While the checker does not require the program source, having the source available allows the use of assertion statements. The programmer can also use a special non-deterministic operation in the program source to model the environment to be simulated. The thread scheduling represents the other source of non-determinism, which is not controllable by the programmer.
Verisoft checks programs dynamically for deadlocks, divergences (a process stops communicating), lifelocks and assertion violations. It uses a

new search algorithm (a refined incremental depth-first search) to guarantee coverage up to a certain level while using sophisticated state space pruning techniques to keep the search manageable. By using a *state-less* search algorithm (i.e. no caching of previously visited states), a much larger amount of code can be checked. It still has a rather long run time for larger programs [30, 31].

Availability: Verisoft is available in binary format for research purposes (under a special license).

Usage: Verisoft has been successfully applied to a complex small C program (2500 LOC), but no larger projects are documented [31].

URL: `http://www1.bell-labs.com/project/verisoft/`

B.1.4 VisualThreads (Eraser)

Purpose: Detect concurrency errors in multi-threaded programs.

Producer: Compaq Inc.

Technologies: Dynamic monitoring techniques, lock set algorithm for detecting race conditions.

Overview: Eraser checks the correctness of locking schemes in multi-threaded programs. It ensures that access to shared resources is always guarded by certain *locking disciplines.* The goal of such a policy is to ensure that no race conditions can occur and that read-write locks operate in a correct manner.
Eraser works by dynamically monitoring the locking of all shared variables. It constantly refines the lock sets, and warns when the lock set becomes empty. Various special cases, such as initialization (where a resource is not available to other threads), read-shared and read-write locks are considered if the source code is properly annotated. These annotations are mainly used for suppressing false alarms [32].
The Eraser algorithm has been successfully used on complex real-life programs, such as the AltaVista indexing engine, where it found one or two race conditions in four sample programs.

Availability: The Eraser algorithm has been implemented in "VisualThreads". VisualThreads is commercially available for OpenVMS and Tru64 Unix Alpha systems, where it is part of the development tools. Java is supported by monitoring the POSIX calls of the Java Virtual Machine.

Usage: VisualThreads is available as part of the development tools; in which projects it is actually used is unknown. Before it was a commercial tool, it has been used on an experimental OS kernel, the Altavista indexing engine, and a couple of other projects, of about 25000 LOC each [32].

URL: `http://www5.compaq.com/products/software/visualthreads/`

B.2 The Spin model checker

Purpose: Static model checker, back-end for other tools.

Producer: Bell Labs (now Lucent Technologies); main author: Gerard J. Holzmann.

Technologies: Explicit state model checking, partial order reduction.

Overview: Spin is a software verification tool that uses a high-level language to specify systems. The language is called PROMELA (PROcess MEta LAnguage). Its development has started at Bell Labs in 1980. Spin has been used to trace logical design errors in distributed systems design, such as operating systems, data communications protocols, switching systems, concurrent algorithms. It can also serve, in conjunction with other tools, to verify the correctness of an abstract representation of source code.
At the core of Spin is a linear temporal logic (LTL) checker. However, it also supports verification of safety and liveness properties not expressible in LTL. It accepts, besides Promela and LTL, also so-called Büchi automata or *never claims.* Promela supports a large variety of high-level constructs such as processes, shared memory, and (buffered or unbuffered) message queues [33].

Availability: Spin is available as Open Source software. Spin is written in C, and can be compiled on any standard ANSI C platform such as Linux, Unix and Windows 9x/NT.

Usage: Several projects (such as Bandera, FeaVer or JPF) are ongoing which use Spin as their back-end model checker.

URL: `http://netlib.bell-labs.com/netlib/spin/whatispin.html`

B.3 Static checkers

B.3.1 Bandera

Purpose: Build a model suitable for model checkers from Java source code.

Producer: Laboratory for Specification, Analysis, and Transformation of Software in the CIS Department at Kansas State University.

Technologies: Program slicing, program abstraction, static model checking; two-way conversion between abstraction levels.

Overview: Bandera tries to bridge the gap between software source code and an abstract representation of it. A special annotation language allows to express assertions and temporal or quantified properties in the source code. *Predicate definitions* for each method are used in *property specifications* which contain the program properties (invariants or sequences of states through which the program always has to go).
Using program analysis (slicing), the first stage of Bandera generates a

simplified version of the program, containing only the statements of interest for the correctness of the program. This can drastically cut down the complexity of the model that is generated from the program.
The second stage reduces the model size further via data abstraction. It generates an intermediate representation of a finite-state model in an intermediate format.
This format is then translated into the specification language of a model checker of choice; so far, SPIN [1] is supported. Translators for the Symbolic Model Verifier (SMV) [20], developed in the Carnegie Mellon University, and Stanford's forthcoming SAL model checker [21] are under construction.[35]
A newer component is the *counter-example generator* that checks faults found in the abstract model for their validity in the actual program, and reports where in the source code the fault was found[34].

Availability: Originally planned for summer 2000, the first beta version has been released on March 8, 2001, under the GPL.

Usage: Bandera has been applied, in conjunction with JPF, to a couple of small programs, including Doug Lea's concurrency package [34].

URL: `http://www.cis.ksu.edu/santos/bandera/`

B.3.2 ESC/Java

Purpose: Detect common programming errors at compile-time.

Producer: Compaq Systems Research Center

Technologies: Generator of background predicates and verification conditions, Simplify theorem prover.

Overview: The "Extended Static Checker" for Java has been developed by Digital Equipment Inc. (now part of Compaq). The first version has been written for checking Modula-3 programs. ESC/Java statically checks a program for `null` reference errors, array bounds errors, potentially incorrect type casts and race conditions.
ESC/Java requires annotations in the source code in its own annotation language. In an internal study, the annotation overhead in the source code was about 13.6% [38]. However, less scrupulous annotations can be made, ignoring certain types of faults.
The checker first generates type-specific *background predicates* to encode data types and type relations for each class and interface. Then, each routine is translated into a *verification condition.* As an intermediate step, a sequence of commands similar to Dijkstra's guarded commands is produced [37]. The Simplify theorem prover then tries to disprove each one of these verification conditions. If it succeeds, the front end transforms the *counter-example context* into a warning and (optionally) a counter-example [36].

Availability: The checker has recently been released and is freely available for research and educational use. A binary version can be downloaded for

Alpha Unix, Solaris, Linux and Windows 9x/NT. The front end has been written in Java while the theorem prover Simplify is written in Modula-3. – A Modula-3 front end is also available, but for Alpha Unix and Intel Windows 9x/NT only.

Usage: ESC/Modula has successfully found fault in several small projects, being totally 20 K LOC in size [38]. There are no numbers available yet for ESC/Java.

URL: `http://research.compaq.com/SRC/esc/`

B.3.3 FeaVer

Purpose: Verify program properties extracted from a test harness.

Producer: Bell Labs Computing and Mathematical Sciences Research Division; main author: Gerard J. Holzmann

Technologies: Model extractor for special C source files, SPIN model checker, error trace generator.

Overview: FeaVer's ultimate goal is a mechanical extraction of a model from the source of software applications. It uses a structured test program (the *test harness*) and a description of program features in order to find violations of such rules. The tool is very small and (so far) only works on event-driven programs, yet it is already quite powerful at finding faults that are very hard to find through conventional testing.
Besides a *test driver* program, the user has to provide a lookup table or *map*, which defines the relevant portions of the source statements to be checked. The developer can start with a simple, coarse mapping and gradually refine it as faults are removed or false warnings occur (the default mapping ignores the values of expressions when evaluating possible execution paths). Together with a list of *properties* (given as logic formulas, where a variety of default properties is already available), the FeaVer framework pre-processes the program source code and produces a Promela (also see Section B.2) specification [39].

Availability: Development is still in its early stage, and the tool will not be released for at least another year (i.e. not before 2002). FeaVer is written in C and only supports C programs so far.

Usage: Only as a prototype within Bell Labs, for the PathStar call processing and telephone switching software, which is certainly a very large and complex piece of software.

URL: `http://cm.bell-labs.com/cm/cs/who/gerard/abs.html`

B.3.4 Flavers

Purpose: Answer verification queries about program properties during design and debugging.

Producer: Laboratory for Advanced Software Engineering Research (LASER), Computer Science Department at the University of Massachusetts Amherst.

Technologies: Data flow and control analysis, static finite state verification, incremental query evaluation.

Overview: Flavers uses data and control flow analysis to build a model of the program to be checked. It then checks this model against verification queries. This also works on concurrent systems. The tool allows to ensure that the software architecture meets the design requirements and consistency rules, such as sequences of events, safety and liveness properties.
Flavers automatically guarantees the presence or absence of certain properties while not requiring knowledge in formal methods. The user can also specify additional properties once he is familiar with the system. Moreover, the Flavers framework offers specialized algorithms for different development stages (referred to as "exploratory", "fault-finding" and "maintenance" modes).
The INCA project is quite similar and uses Integer Programming in order to verify certain properties; however, that project has not yet come far enough to have an automated translation from source code to linear inequalities, which are required by the theorem prover.
The Static Concurrency Analysis Research Project collects patterns of problems that frequently occur in multi-threaded programs, in order to facilitate the development of future software tools [16].

Availability: An Ada and C++ version are implemented. The C++ version belongs to the company MCCquEST and is not available; the Ada version has been developed by the university and is available upon request. A Java version is under development, but still in its early stage and not yet ready for a release.

Usage: No numbers are available.

URL: `http://laser.cs.umass.edu/tools/flavers.html`

B.3.5 Jlint

Purpose: Semantic verifier detecting certain deadlocks, race conditions and a few other faults.

Producer: Moscow State University, Research Computer Center; main author: Konstantin Knizhnik.

Technology: Control flow/lock dependency analysis, specialized checks for other faults.

Overview: Jlint comes as two programs, a simple syntax verifier (AntiC) and a semantic verifier (Jlint). The former checks for a few common potential syntax errors. The latter is much more interesting, for it extracts information from (non-annotated, normally compiled) Java class files and performs consistency and flow analyses on them. Jlint is capable of dealing

with missing debugging information which some Java compilers cannot (yet) generate. It also allows a hierarchical selection of the checks that should be performed.
The core algorithm checks Java class files for loops in the lock dependency graph. This graph includes both static and dynamic methods. It also makes sure the programs follow certain consistency rules when using the `wait` method in Java. Race conditions are found by building the transitive closure of methods which can be executed concurrently and the methods they call. Then, all fields accessed by such methods which fulfill certain conditions are reported as possible race conditions in data access. Jlint is rather conservative at reporting errors, since it does not allow annotations which could eliminate false positives.

Availability: Freely available; written in C and C++, and should work on any platform.

Usage: No other numbers are available, but Jlint has been applied successfully at Trilogy to large scale software (several projects of several ten thousand LOC each).

URL: `http://www.ispras.ru/~knizhnik/jlint/ReadMe.htm,http://artho.com/jlint/`

B.3.6 JPF

Purpose: Integrate model checking, analysis and testing.

Producer: Automated Software Engineering Group (ASE) at NASA; main author: Klaus Havelund.

Technologies: Slicing, abstraction; 1.0: Java to Promela translator; 2.0: special JVM (MC-JVM) and model checker.

Overview: The "Java PathFinder" has been developed at the Automated Software Engineering (ASE) department at NASA. Currently, JPF can only check invariants and deadlocks. Invariants are given as a Boolean Java method.
After an abstraction and a slicing stage, which both reduce the state space of the program a lot, a depth-first search is performed on the program stages. A special JVM, which allows to move forward and *backward* one state in the bytecode execution, is used for this.
The first version was a translator from Java to Promela [40]. Special assertion and error methods specify the properties to be checked. It has, however, only supported a fairly restricted subset of Java. Because it was too difficult to extend the program to support more Java constructs, a different approach has been taken for the second version, which works directly on bytecode. It can therefore fully support all Java features [42].

Availability: It is currently being checked by NASA's legal department whether the program can be made available to selected third parties or not.

Usage: JPF has been applied to the Remote Agent Spacecraft Controller (RAX), where it found a deadlock, and the DEOS Avionics Operating System. After the slicing stage, the largest package was 1443 LOC in size [41].

URL: `http://ase.arc.nasa.gov/jpf/`

B.3.7 LockLint

Purpose: Detect race conditions and deadlocks in C programs.

Producer: Sun Microsystems Inc.

Technology: Control flow analysis.

Overview: LockLint consists of two parts: A special mode in Sun's C Compiler (invoked via a command line switch) and a program that analyzes the resulting LockLint files. Assertions about a large variety of lock properties can be made in the source code via macros, which are evaluated by the C compiler. These include (possibly intended) side-effects of functions, properties that should hold upon entry of a function or when accessing a variable, and consistent lock usage.
When LockLint is run, it spawns its own shell, which allows the user to enter verification properties or LockLint commands interactively or run them via a shell script. Additional annotations in the C sources are not required, but recommended since the checker cannot make all assumptions correctly.
LockLint tries to guess the set of possible values for function pointers and global variables; again, manual overrides (in the LockLint shell) are possible to correct wrong assumptions.

Availability: Commercially available as a part of the Forte (formerly SunWorkshop) compiler suite for C programs.

Usage: The Forte tools are widely used in the industry, but no numbers are available about the projects that have used LockLint.

URL: `http://www.sun.com/forte/c/`

B.3.8 MC

Purpose: Build specific compiler extensions to check, optimize and transform code.

Producer: Computer Systems Laboratory, Stanford University.

Technology: Static analysis (by an extensible compiler).

Overview: "Meta-level Compilation" (MC) is a project that verifies whether a program violates certain consistency patterns. The user can combine simple rule templates and apply them to specific rules such as "system libraries must check user pointers for validity before using them". By setting up a few such rules, one can effectively check a program against

a large range of errors. The rules are expressed in a high-level state-machine language.
After building a control-flow and data-flow graph, MC checks that model against the specified correctness properties. Even though a basic backtracking algorithm is used for the state space search, an effective caching algorithm avoids an exponential run time behavior in practical cases [43].

Availability: It has not been decided yet whether MC will ever be publicly released.

Usage: MC has been successfully used in searching the Linux and BSD kernels for faults, specifically for incorrect resource management and interrupt disabling/enabling schemes [43].

URL: `http://hands.stanford.edu/`

B.3.9 SLAM

Purpose: Check that software satisfies critical behavioral properties.

Producer: Software Productivity Tools Research group, Microsoft Inc.

Technologies: Boolean programs, parametrized verification of models for multi-threaded software.

Overview: SLAM (Software, Analysis, Languages, Model Checking) is a suite of programs that is being developed at the Microsoft Software Tools research group. One of the main challenges is the automation of the abstraction of source code. The focus of the project is the checking of invariants and temporal properties. For the latter, a new formal model for multi-threaded program is used: the *LGFSM*, an extended finite state machine.
At the core of the tools is a model checker for Boolean Programs (programs that only use Boolean variables). This (strong) abstraction allows to check invariants and termination of programs, a problem which is in general undecidable [44].
So far, SLAM only works on C (partially on C++) programs, verifying the correct behavior of drivers and system libraries. It has been successfully used for internal projects.

Availability: SLAM will be made available for research. The roll out for the SLAM tools is planned as follows:

- bebop (model checker) will be released in early 2001
- c2bp (abstractor) will be released mid-year, 2001
- other tools to follow.

Usage: So far, SLAM has been used internally to model a multi-threaded memory manager [44].

URL: `http://research.microsoft.com/slam/`

B.4 Other tools

B.4.1 JML/LOOP

Purpose: Behavioral interface specification language to specify properties of Java modules.

Producer: Department of Computer Science, Iowa State University; Computer Science Department Nijmegen (Holland)

Technologies: LOOP translator, PVS [19] or Isabelle proof tool [18].

Overview: The "Java Modeling Language" is an interface specification language describing the behavior of classes. The intent is to make it safe to write subclasses to existing classes, given only access to the object code and a specification written in JML.
This project works in cooperation with ESC/Java; both specification languages are merging, but JML's goal is focused on "design by contract" [45] and behavior specification, while ESC/Java works on a lower level. The development of the JML tools is still in an early stage, but an automated translation of JML specifications into verification conditions (using LOOP: Logic of Object-Oriented Programming) is being developed. Also, extensions regarding concurrency are being explored. While the syntax for temporal statements in JML already exists, the work in this area is still experimental. (ESC/Java offers other temporal constructs, which operate on a lower level, such as lock sets.)

Availability: JML is available (including source code) under the GPL. The current version, written in Java, features a type checker and a run time assertion checker for Java programs and JML annotations.

Usage: As the entire tool suite has not been developed yet, no reports about their usage are available.

URL: `http://www.cs.iastate.edu/~leavens/JML.html`

Appendix C

Multi-threading in Java

C.1 Threads

Java includes multi-threading in the language itself, which makes it much easier to use. This appendix only describes those aspects of multi-threading that are relevant to static checking and this thesis. For more information about multi-threading, see [48, pp. 149–152] or [47].

The `java.lang.Thread` class allows the creation and control of several threads. These threads share the address space of the virtual machine.[1] It is possible to run different instances of threads, with their own data; however, all instances are (potentially) accessible to all threads. For practical purposes, the programmer can assume that the virtual machine runs on only *one* CPU, and each thread periodically receives a "time slice" by the scheduler. Note that the official Java specification poses no requirement for a fair scheduling among threads of the same priority. This emphasizes once more that the programmer has to take *any* possible schedule into account.

C.2 Thread synchronization

C.2.1 Introduction

For ensuring the correctness of a multi-threaded program, *thread synchronization* is crucial. The basic technique is to prevent two threads from accessing the same object simultaneously. This is done via a *lock* on that object (or another object which represents the lock on that object – this technique is sometimes employed for primitive types like integers, or for collections). While any one thread holds the lock, another thread requesting it is *blocked* (suspended) until the first thread has released the lock.

There is only one way in Java to acquire a lock: the `synchronized` statement. With `synchronized(resource) { /* block */ }`, the current thread blocks until it is able to acquire a lock on `resource`. The lock is held until the entire block is finished (either when the last statement is executed

[1] This is not quite correct, because the Java Virtual Machine performs some kind of "caching" for the variables accessed by threads. This is, however, not relevant for this discussion. For more information about this, see [50, Chapter 8].

or the block is aborted by other means, e.g. `break` or `return` statements, or exceptions.

A special case of synchronization is `synchronized(this)`. This acquires a lock on the *current instance*, which is often described as "making a block atomic". This is a common misconception. Indeed, the execution of that block is *not* atomic; holding a lock on the current instance does not prevent a preemption of that thread by the scheduler. The execution is, though, "atomic" *on that instance* – no two threads may execute that block for the same object instance at the same time.

```
                                  method() {
synchronized method() {           synchronized(this) {
  ...                        ⇔      ...
}                                   }
                                  }
```

Figure C.1: `Synchronized(this)` vs `synchronized` methods.

If a `synchronized(this)` block spans an entire method, `synchronized` methods are commonly used instead. Such a method automatically acquires a lock on `this` before its body is executed. After method execution, the lock is released. (If a lock is held before, acquiring it again simply increases a counter within the virtual machine, but has no other effect.)

Synchronizations may also be used on classes, where they have the effect of "locking out" all instances belonging to that class. Commonly, this used to synchronize on the class of the current instance, but a synchronization on other classes is possible as well.

Two other important synchronization primitives are `wait` and `notify`. If a thread holds a lock on a resource, and has to wait for a certain condition to become true, it should call `resource.wait()` inside a loop. This causes that thread to "sleep" (block) until another thread calls `resource.notify()`, which "wakes up" any thread waiting on `resource`.

Calling `notify` releases the lock, and causes the original (`wait`ing) thread to re-acquire it before resuming execution. Normally, that thread has to verify again whether the condition it is waiting on now holds; hence `wait` is usually called inside a loop rather than an `if` statement. How the latter can introduce subtle faults is illustrated in example D.10.

If it cannot be guaranteed that any thread that has just been notified can actually resume execution (i.e. the condition it is waiting on has become true), then `notifyAll` needs to be used instead (see example D.11). This will "wake up" all threads waiting on that resource (in random order). At least one of them has to be able to continue execution; otherwise all waiting threads may end up stopped.

C.2.2 Usage

Preventing race conditions

Synchronization is crucial for preventing *race conditions* (incorrect concurrent access of a shared resource). Whenever the access to a resource r has to be exclusive, the programmer has to ensure that each thread always holds a lock

guarding r when it is used. If L is the set of locks held at a certain time, the programmer has to ensure that a r is 1) only read when a thread holds at least one lock in L_r and 2) only written when a thread holds *all* locks in L_r [36]. Often, a synchronization on r itself is used for guaranteeing non-concurrent access; more advanced locking schemes require additional locks.

It has to be noted that the current instance is "invisible" to other threads inside the constructor (i.e. while the constructor is being executed; therefore the instance is also invisible if the constructor calls other methods). This is because the current thread does not hold a reference to that object yet; and it cannot share something it does not possess. An exception to this rule are constructors which pass the `this` pointer to other classes. This is very rare, though, and usually indicates a poor design.

Preventing deadlocks

By obtaining too many locks, the execution of each thread can be slowed down greatly, because they spend a lot of time requesting locks. Even worse, incorrect locking schemes can lead to a *deadlock:*

```
public void run() {
  if (ab) {
    synchronized (a) {
      synchronized (b) {
      }
    }
  } else {
    synchronized (b) {
      synchronized (a) {
      }
    }
  }
}
```

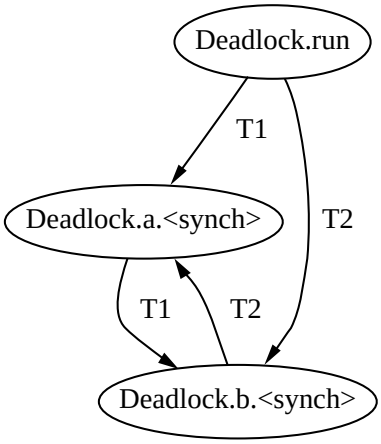

Figure C.2: The deadlock in example D.1.

If thread T_1 is holding a lock a and requests lock b, it will wait until that lock becomes available. Assume another thread T_2 already holds b and requests a; in this case, we have a deadlock because both threads wait for an event that will never happen. Figure C.2 shows the *lock graph* for this example. In this graph, there is a loop between the two nodes `Deadlock.a.<synch>` and `Deadlock.b.<synch>`, which represent the `synchronized(a)` and `synchronized(b)` statements, respectively.

The virtual machine has no built-in deadlock resolution, which would cause one of the two threads to give up its lock. Therefore deadlock analysis is an integral part of writing safe multi-threaded programs in Java.

C.3 Summary

This appendix gave a short introduction to the crucial features of multi-threaded programming in Java. The synchronization is the core of the design of a multi-threaded program. A lack of synchronization can lead to race condition, while an inconsistent locking scheme may result in deadlocks.

Appendix D

Example listings

D.1 Selected programs

D.1.1 Deadlock

These examples contain some simple classes showing various variants of deadlocks. The deadlocks are all intra-method deadlocks, using the `synchronized(resource)` statement. This statement blocks the current thread until it can obtain a lock on the object `resource`.[1] All programs within this section were taken from the Rivet test-suite [29]. The deadlocks exhibited are all easily detected by Rivet, yet may be hard to detect for static checkers.

Deadlock

This is the simplest possible deadlock: two instances compete for resources `a,b` in the opposite order, therefore forming a loop in the lock-acquisition hierarchy. For more details about why a deadlock occurs here, see Appendix C.

Listing D.1 Deadlock: `run` method of two competing threads.

```
public void run() {
  if (ab) {
    synchronized (a) {
      synchronized (b) {
      }
    }
  } else {
    synchronized (b) {
      synchronized (a) {
      }
    }
  }
}
```

[1] This is a simplification; actually, the current thread becomes *runnable* once the lock on resource is available. If it is not running at that time, it might occur that another thread obtains the lock in the meantime. Therefore, thread starvation can still occur when only using the standard `synchronized(resource)` mechanism in Java.

However, Java makes it fairly simple to implement more advanced resource sharing mechanisms such as a queued locking [47, pp. 178 - 180].

Deadlock2

This is a slight variation of the Deadlock example. Here, the nested synchronized blocks in the `run` method were replaced with calls to static synchronized methods of the dummy classes LockA and LockB, which call in turn a static synchronized method of the other class. This models the same intra-method locking scheme (within the `run` method) on a method level.

Listing D.2 Deadlock2: Locking scheme from Deadlock on a method level.

```
public synchronized void run() {
  if (ab) {
   LockA.foo(); // recursively calls LockB.bar, obtaining lock on B
  } else {
   LockB.foo(); // recursively calls LockA.bar, obtaining lock on A
  }
}
```

DeadlockWait

This example shows a deadlock when using `wait` and `notify`. The first thread obtains locks `a` and `b`, then waits on `b`. The second thread then blocks trying to obtain lock `a`.

Listing D.3 DeadlockWait: `run` methof two competing threads. While the first thread is waiting on b, the second thread never reaches the statement `b.notify()`.

```
public void run() {
  if (ab) {
    synchronized (a) {
      synchronized (b) {
        try { // Java throws an exception when execution continues
          b.wait();
        } catch (InterruptedException i) { // continue execution
          System.out.println(name+" was interrupted!");
        }
      }
    }
  } else {
    synchronized (a) {
    }
    synchronized (b) {
      b.notify();
    }
  }
}
```

DeadlockWait2

This is a slight variation of the DeadlockWait example. Here, the nested synchronized blocks in the `run` method were replaced with a call to a synchronized methods of the dummy classes LockA. That class is a singleton class,

having only a single instance.[2] The method which is called by `run` calls a method of singleton class LockB, shown in Listing D.4.

Depending on the value of a boolean variable, it either calls `wait` or `notify` on itself. Since, at that stage, a lock on the only instance of LockA is still active, the `notify` message never reaches its destination. This models the same intra-method locking scheme (within the `run` method) on a method level and creates the same deadlock.

Listing D.4 DeadlockWait2: method `foo()` of class Lock B, which is called from a synchronized method of class LockA.

```
class LockA {
  public synchronized void foo(String name, boolean ab) {
    LockB.getInstance().foo(name, ab);
  }
}
class LockB {
  public synchronized void foo(String name, boolean ab) {
    if (ab) {
      try { this.wait(); }
      catch(InterruptedException e) { }
    } else {
      this.notify();
    }
  }
}
```

Deadlock3

This program demonstrates a simple cyclic deadlock with three threads competing for locks `a`, `b`, and `c`.

Listing D.5 Deadlock3: `run` method of three competing threads.

```
public void run() {
  if (order == 0) {
    synchronized (a) {
      synchronized (b) {
      }
    }
  } else if (order == 1) {
    synchronized (b) {
      synchronized (c) {
      }
    }
  } else {
    synchronized (c) {
      synchronized (a) {
      }
    }
  }
}
```

[2] Static methods could not be used here because the `wait` and `notify` methods are not available to them.

D.1.2 SplitSync

This example is also taken from the Rivet test-suite [29]. It simulates a case where the locking granularity is too low. The lock is released in between a calculation, and the assumption is made that the value is unchanged in between. This programming mistake leads to a race condition on the shared variable.

Listing D.6 Race condition: A lock is released in between a calculation.

```
public void run() {
  int y;
  synchronized (resource) {
    y = resource.x;
  }
  synchronized (resource) {
    if (resource.x != y) {
      System.out.println("****** Assertion violation! *******");
      System.exit(0);
    }
    resource.x = y + 1;
    //      System.out.println(name+" = "+x[0]);
  }
}
```

D.1.3 Jlint test example

Classes A, B with cyclic locks (method calls)

Listing D.7 is taken from the `jlint` documentation. It shows two classes that call each other's synchronized methods in a way that a loop in the lock graph results.

D.1.4 ESC/Java example

The ESC/Java manual contains one slightly bigger example (Listing D.8). It deals with a pathological case where a part of a tree is rotated. Therefore, the static checking assumes that the locking order is inconsistent; however, while it appears so syntactically, the locking scheme is semantically correct, because the two locks have been swapped in the tree structure. For more information, see the ESC/Java user manual [36].

As a result of this unusual behavior, the ESC/Java checker generates a spurious warning about a non-existent deadlock and also misses a possible race condition if a synchronization statement is left out in the second block.

D.1.5 Shared buffer (producer/consumer problem)

The first three versions of this implementation are taken from the Rivet test suite [29]. They show a working version where two common faults are introduced. These faults were successfully detected by Rivet.

A shared buffer also models a data base wrapper pretty well. This is very interesting, because such wrappers (or similar wrappers for a server-based service) are very common in practice. Indeed, one of Trilogy's packages (see Section A.2.2) performs exactly this functionality: among other things, it handles

Listing D.7 Jlint example: Loop in lock graph.

```
class A {
  public synchronized void f1(B b) {
    b.g1();
    f1(b);
    f2(b);
  }
  public void f2(B b) {
    b.g2();
  }
  public static synchronized void f3() {
    B.g3();
  }
}
class B {
  public static volatile A ap;
  public static volatile B bp;
  public synchronized void g1() {
    bp.g1();
  }
  public synchronized void g2() {
    ap.f1(bp);
  }
  public static synchronized void g3() {
    g3();
  }
}
```

a pool of connections to the data base. Such a wrapper, running in several threads, offers connections to the data base. The number of threads is certainly limited. The access to the data base is modeled by the buffer; what is not modeled are dependencies (i.e. locks) between data base transactions, which are in the data base layer.

Correct multi-threaded implementation

Listing D.9 shows the source code of the buffer class. The main method, where two producers and one consumer are created and started, is omitted.

Broken implementation (if instead of while)

Since `notifyAll` wakes up all threads, it is very well possible that several threads are woken up. These could finish the body of the enqueue method one by one, thus overflowing the buffer: After the first thread that fills the buffer has finished execution, it has falsified the condition that made the other threads wait. When the other threads continue, they have to verify that condition again. Listing D.10 shows the faulty `enq` method:

> "The enq function of the buffer has an if to check the buffer-full condition instead of a while. We create one producer, one consumer, and another low-priority producer – low-priority so that a typical scheduler will never encounter the if-vs-while bug, but the systematic tester will find it!" [29]

Listing D.8 ESC/Java example: Pathological case with two locks: hierarchy of locking data structure is reversed during program execution.

```
public class Tree {
  public /*@ monitored */ Tree left, right;
  public /*@ monitored non_null */ Thing contents;

  //@ axiom (\forall Tree t; t.left != null ==> t < t.left);
  //@ axiom (\forall Tree t; t.right != null ==> t < t.right);

  Tree(/*@ non_null */ Thing c) { contents = c; }

  //@ requires \max(\lockset) <= this;
  public synchronized void visit() {
     contents.mungle();
     if (left != null) left.visit();
     if (right != null) right.visit();
  }

  //@ requires \max(\lockset) <= this;
  public synchronized void wiggleWoggle() {
    // Perform a rotation on this.right (but give up and just
    // return if this.right or this.right.left is null):
    //
    //   this              this
    //   /  \              /  \
    // ...   x           ...   v
    //      / \    -->        / \
    //     v   y            u   x
    //    / \                  / \
    //   u   w                w   y
    //
    Tree x = this.right;
    if (x == null) return;
    synchronized (x) {
      Tree v = x.left;
      if (v == null) return;
      synchronized (v) {
        x.left = v.right;
        v.right = x;
        this.right = v;
      }                           // line (a)
    }
    // Undo the rotation:
    Tree v = this.right;
    synchronized (v) {            // line (b)
      Tree x = v.right;
      if (x != null) {            // line (c)
        synchronized (x) {        // line (d)
          v.right = x.left;
          x.left = v;
          this.right = x;
        }
      }                           // line (e)
    }
  }
}
```

Listing D.9 Shared bounded buffer (correct version).

```
public class BufferWorks {
  static final int ITEMS_PRODUCED = 2;

  static class Producer implements Runnable {
    private Buffer buffer;
    public Producer(Buffer b, String n) { buffer = b; }
    public void run() {
      try {
        for (int i=0; i<ITEMS_PRODUCED; i++) {
          buffer.enq(name);
        }
      } catch (InterruptedException i) {
        System.err.println(i);
      }
    }
  }

  static class Consumer implements Runnable {
    private Buffer buffer;
    public Consumer(Buffer b) { buffer = b; }
    public void run() {
      try {
        for (int i=0; i<ITEMS_PRODUCED*2; i++) { // while (true)
          buffer.deq();
        }
      } catch (InterruptedException i) {
        System.err.println(i);
      }
    }
  }

  static class Buffer { // shared bounded buffer
    static final int CAPACITY = 1;
    // Need extra slot to tell full from empty
    static final int BUFSIZE = CAPACITY+1;
    private int first, last;
    private Object[] els;
    public Buffer() { first = 0; last = 0; els = new Object[BUFSIZE]; }

    public synchronized void enq(Object x) throws InterruptedException {
      while ((last+1) % BUFSIZE == first)
        this.wait();
      els[last] = x;
      last = (last+1) % BUFSIZE;
      this.notifyAll();
    }

    public synchronized Object deq() throws InterruptedException {
      while (first == last)
        this.wait();
      Object val = els[first];
      first = (first+1) % BUFSIZE;
      this.notifyAll();
      return val;
    }
  }
}
```

Listing D.10 Race condition: condition of `wait` is not checked again after having received notification.

```
public synchronized void enq(Object x) throws InterruptedException {
       if ((last+1) % BUFSIZE == first)
         this.wait();
       els[last] = x;
       last = (last+1) % BUFSIZE;
       this.notifyAll();
     }
```

Broken implementation (notify instead of notifyAll)

The `notify` method may only be used if all waiting threads are waiting on the same condition. Otherwise, it can happen that thread a_1 asserts condition `b` and calls `notify`, intending to wake up a thread b_n which is supposed to check that condition in a `while` loop after returning from waiting. However, if thread a_2 received the notification instead, it only realizes that the condition it is waiting on is still false. It continues waiting, having consumed the `notify` message. As a result of this, both threads will `wait` forever, never receiving the message from a_1.

In Listing D.11, "buffer not full" and "buffer not empty" are the two conditions checked by `enq` and `deq`, respectively.

Listing D.11 Condition deadlock: `notify`instead of `notifyAll` is used, while threads are waiting on different conditions.

```
public synchronized void enq(Object x) throws InterruptedException {
   while ((last+1) % BUFSIZE == first)
     this.wait();
   els[last] = x;
   last = (last+1) % BUFSIZE;
   this.notify();
}

public synchronized Object deq() throws InterruptedException {
   while (first == last)
         this.wait();
   Object val = els[first];
   first = (first+1) % BUFSIZE;
   this.notify();
   return val;
}
```

Version based on semaphores

This version has been implemented based on an example found at [22], which does not use real Java code for shared access. The `Buffer` class as such is a "naïve" implementation, with no statements for concurrent access. Instead, the producers and consumers share semaphores in order to guard access to the buffer and prevent illegal operations (Listing D.12). Three semaphores are used: `mutex` for locking, and `full` and `empty` for guarding the preconditions.

In this implementation, all the Java-specific statements for synchronization are hidden in the Semaphore implementation (Listing D.13).

Listing D.12 Buffer implementation using semaphores.

```
class Producer implements Runnable {
  private Buffer buffer;
  public Producer(Buffer b) { buffer = b; }
  Object produce() { /* ... */ return new Integer(42); }
  public void run() {
    Object item;
    for (;;) {
      item = produce();
      empty.down();
      mutex.down();
      b.addElement(item);
      mutex.up();
      full.up();
    }
  }
}

class Consumer implements Runnable {
  private Buffer buffer;
  public Consumer(Buffer b) { buffer = b; }
  void consume(Object o) { /* ... */ }
  public void run() {
    Object item;
    for (;;) {
      full.down();
      mutex.down();
      item = b.removeElement();
      mutex.up();
      empty.up();
      consume(item);
    }
  }
}
```

Listing D.13 Semaphore implementation.

```
public class Semaphore {
  private int value;
  public Semaphore(int initialValue) { value = initialValue; }
  public synchronized void up() { value++; notify(); }
  public synchronized void down() {
    while (value == 0) {
      try { wait(); }
      catch (InterruptedException e) { }
    }
    value--;
  }
```

D.1.6 Dining philosophers

These examples show different implementations of the famous "Dining Philosophers problem" [53]. All implementations are based on semaphores. As they

are implemented, all versions can show starvation (lifelocks) without further refinement, depending on the scheduler used in the Java Virtual Machine. The semaphore implementation is the same is in D.13. More implementations of algorithms that solve this problems can be found at [22].

Listing D.14 Naïve implementation of the Dining Philosophers problem.

```
class PhilosopherDeadlock implements Runnable {
  private int i; // which philosopher
  private static int N; // # of philosophers
  private static Semaphore[] fork;
  private static PhilosopherDeadlock[] philosopher;

  public static void main(String[] args) {
    N = Integer.parseInt(args[0]);
    fork = new Semaphore[N];
    philosopher = new PhilosopherDeadlock[N];
    for (int i = 0; i < N; i++) {
      fork[i] = new Semaphore(1);
      philosopher[i] = new PhilosopherDeadlock(i);
    }
    for (int i = 0; i < N; i++)
      new Thread(philosopher[i]).start();
  }

  public PhilosopherDeadlock(int num) { i = num; }

  public void run() {
    for (;;) {
      take_forks();
      eat();
      put_forks();
      think();
    }
  }

  void take_forks() {
    fork[i].down(i);
    fork[(i+1)%N].down(i);
  }

  void put_forks() {
    fork[i].up(i);
    fork[(i+1)%N].up(i);
  }
}
```

Naïve implementation (has deadlock)

In Listing D.14, each philosopher tries the right fork first, then the left fork. A deadlock occurs when each philosopher has taken the right fork and no more forks are available.

Version with different allocation strategy for one philosopher

In this version, instead of trying the *right* fork first in all cases, each philosopher tries to *lower numbered* fork first. This means that the Nth philosopher

tries the first fork, then the Nth fork – which is the same order in which the first philosopher acquires the forks. This small change breaks the loop in the locking scheme and prevents a deadlock.

Listing D.15 Solution 1 for the Dining Philosophers problem involving a different locking scheme for the last philosopher.

```
void take_forks() { // try lower numbered fork first
  if (i == N-1) {
    fork[0].down(i);
    fork[i].down(i);
  } else {
    fork[i].down(i);
    fork[i+1].down(i);
  }
}
```

Version with host

This solution has been inspired by [46]. Here, a central host grants access to the forks. He only allows at most $N - 1$ philosophers to hold forks at any time. This algorithm, shown in Listing , simulates environment with a central resource pool. The addition to the algorithm is a minor one (it uses the same circular fork allocation strategy as the naïve approach). The implementation of the host is quite simple. It uses class (`static`) variables to track the number of philosophers holding forks, while each philosopher has his own instance of a host, allowing an easy implementation of the "blocking" behavior of `host.request()`.

Listing D.16 Solution 2 for the Dining Philosophers problem where a central host controls access to the resources (forks).

```
class Host {
  private volatile static int N = 0; // number of resources (forks)
  private volatile static int count = 0; // number of resource users
  private static Object lock = new Object();

  public Host() {
    N++;
  }

  public void request() {
    synchronized(this) {
      while (count == N-1) { // not if
        try { wait(); }
        catch(InterruptedException e) { }
      }
      synchronized(lock) { // cannot lock on count ...
        count++; // ... because count is an "int", not an Object
      }
    }
  }

  public synchronized void release() {
    count--;
    notify();
  }
}
class Philosopher {
  /* ... */
  public void run() {
    for (;;) {
      think();
      host.request();
      take_forks();
      eat();
      put_forks();
      host.release();
    }
  }
}
```

Appendix E

Test results

This appendix contains excerpts of the output of the tests runs. Especially in the case of ESC/Java, all irrelevant information has been omitted. The counter-examples are not shown either, because they are too long for being included here.

Execution times for Jlint, ESC/Java and MaC were measured on an unloaded Pentium III (650 MHz) running Linux and Sun's JRE 1.3. Extremely short execution times were indicated as such, since the results of the `time` command is not fully reproducible in such cases.

ESC/Java has its own timing facility, which excludes the initialization of the engine. Since the initialization time, while being moderate, still matters in practice, the `time` command was still used to measure the execution time in that case. The effort required for the annotations is usually moderate, but can be quite high for complex programs.

For VisualThreads (running on an old Alpha/233 MHz computer with only 116 MB RAM), the built-in time measuring was used. Because VisualThreads only runs in GUI mode, the Java classes implementing the GUI and the X window protocol were probably also slowing down the program quite a little. Therefore, the numbers given for VisualThreads should be taken with caution. The output of VisualThreads is in a GUI window, where it can unfortunately not be copied to a text file. It includes the full class name of the object instance and also the memory address of the reference to that object. The output given here is a simplified version of the content of the table in the GUI window.

E.1 Benchmark

For testing the speed of the ESC/Java, a small program was taken and gradually extended in order to increase the size of the code to be checked.

For Jlint, all these tests ran clearly below 0.05s and were not very useful. When Jlint is tested with all `java/*` classes that ship with Sun's JDK 1.3.0, the old version can validate all class files within about 0.75 seconds! The extensions did not slow Jlint down a lot - it still runs within 1.0 seconds. VisualThreads running times could not be directly compared, since the programs were run on an old Alpha machine, which was clearly outmatched by the PC running Jlint and ESC/Java.

ESC/Java

Program	LOC	KB	Time [s]
Philosopher with status monitor	126	3	6.2
Status output now in HTML	504	11	10.4
Status with 1000 integer calculations	1512	24	12.8

E.2 Program checker results

E.2.1 Deadlock

Jlint

```
Deadlock.java:31: Method Deadlock.run() implement-
ing 'Runnable' interface is not synchronized
Verification completed: 1 reported messages
```

Execution time: very short (< 0.05 s)

This was the first program where Jlint showed its major weakness: It does not recognize the synchronized statement within a method. The `run` method is not always synchronized (a correct program design eliminates the need for this, since each instance of the class `Thread` is initialized with a reference to a distinct object), so this warning generates some "noise" (but it can easily be turned off).

New Jlint

```
Deadlock.java:31: Method Deadlock.run() implement-
ing 'Runnable' interface is not synchronized
Deadlock.java:48: Lock Deadlock.a is requested while hold-
ing lock Deadlock.b, with other thread holding Dead-
lock.a and requesting lock Deadlock.b.
Deadlock.java:39: Lock Deadlock.b is requested while hold-
ing lock Deadlock.a, with other thread holding Dead-
lock.b and requesting lock Deadlock.a. Verification com-
pleted: 3 reported messages.
```

The extension now fully analyzes this case and prints a very precise and useful warning.

ESC/Java

```
Deadlock: run() ...
------------------------------------------------------------
Deadlock.java:33: Warning: Possible deadlock (Deadlock)
      synchronized (a) {
                    ^
Execution trace information:
    Executed then branch in "Deadlock.java", line 32, col 12.
------------------------------------------------------------
```

Execution_time: 5.2 s

Annotations: 2

The main effort about the annotations was for figuring out which annotations to use where. Also, the `run` method had to be synchronized in order to get a useful counter-example.

The warning issued by ESC/Java is correct, but not very specific. The counter-example is not very helpful either for locating the source of the fault, unless one is familiar with its notation.

VisualThreads

Since this is the first example, a more detailed version of the output is given here:

```
Deadlock occurred with cycle length 4.
```

- Thread Thread-2@0x51c2a0 waits for thread Thread-3@0x51c1f0.

Thread	Event type	Object	Function
Thread-2@0x51c2a0	blocked on	Deadlock$Lock@<addr1>	phread_mutex_lock
Thread-3@0x51c1f0	locked	Deadlock$Lock@<addr2>	phread_mutex_tryrec

- Thread Thread-3@0x51c1f0 waits for thread Thread-2@0x51c2a0

Thread	Event type	Object	Function
Thread-3@0x51c1f0	blocked on	Deadlock$Lock@<addr2>	phread_mutex_lock
Thread-2@0x51c2a0	locked	Deadlock$Lock@<addr1>	phread_mutex_tryrec

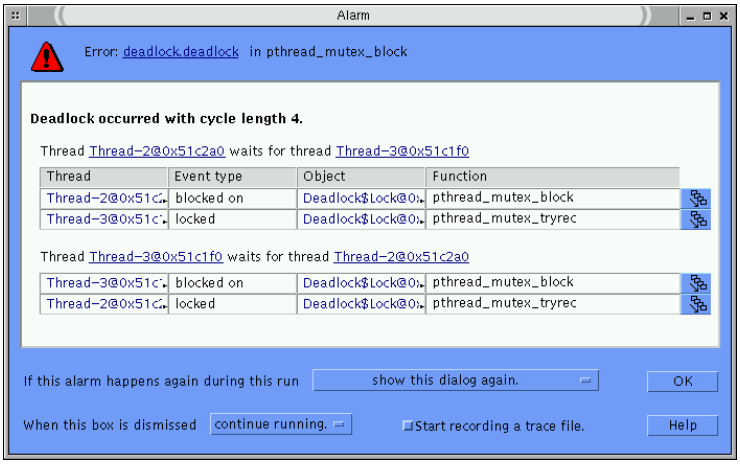

Figure E.1: Screenshot of warning for Deadlock example.

Figure E.1 shows a screenshot of the program. The full name of a thread of lock (including its memory address) can be seen by moving the mouse pointer over it.

Execution time: 22 s (mainly for initializing the Java Virtual Machine)

VisualThreads detects the deadlock and shows a detailed report. Unfortunately, the output of that window cannot be saved to a text file. It is possible, however, to record the behavior of the program in a trace file, which can be replayed afterwards. The output, despite the fact that the main thread of the JVM is counted, too (so the Java threads start with index 2), is easy to interpret.

E.2.2 Deadlock2

Jlint

```
LockB.java:9: Loop 1: invocation of synchro-
nized method LockA.bar() can cause deadlock
LockA.java:12: Loop 1: invocation of synchro-
nized method LockB.bar() can cause deadlock
Verification completed: 2 reported messages
```

Execution time: very short (< 0.05 s)

After remodeling the recursive locking scheme on a higher level (by introducing dummy classes which could be automatically generated), Jlint successfully recognizes the deadlock.

New Jlint

The new version still finds the fault.

ESC/Java

```
Deadlock2: run() ...
------------------------------------------------------------------------
Deadlock2.java:14: Warning: Precondition possibly not established (Pre)
      LockA.foo(name); // recursively calls LockB.bar, obtaining a loc ...
               ^
Associated declaration is "LockA.java", line 6, col 6:
  //@ requires \max(\lockset) <= LockA.class;
      ^
Execution trace information:
    Executed then branch in "Deadlock2.java", line 13, col 12.
------------------------------------------------------------------------
...
LockA: foo(java.lang.String) ...
------------------------------------------------------------------------
LockA.java:13: Warning: Precondition possibly not established (Pre)
    LockB.bar();
             ^
Associated declaration is "LockB.java", line 16, col 6:
  //@ requires \max(\lockset) <= LockB.class;
      ^
Execution trace information:
    Executed then branch in "LockA.java", line 9, col 6.
------------------------------------------------------------------------
...
LockB: foo(java.lang.String) ...
------------------------------------------------------------------------
LockB.java:13: Warning: Precondition possibly not established (Pre)
    LockA.bar();
             ^
Associated declaration is "LockA.java", line 16, col 6:
  //@ requires \max(\lockset) <= LockA.class;
      ^
Execution trace information:
    Executed then branch in "LockB.java", line 9, col 6.
------------------------------------------------------------------------
```

Execution time: 6.0 s

Annotations: 5

ESC/Java recognizes the deadlocks, and after adding some annotations, spurious warnings vanish. However, the output is not easy to interpret. Especially the information about the "then branch" in the two lock classes is confusing, since that statement just prints out some status information. It is the first statement in that method, which is probably the reason why the execution trace appears in this way.

VisualThreads

```
Deadlock occurred with cycle length 4.
```

Execution time: 21 s

The output resembles the one from the previous example in Section B.2. Again, the output is detailed enough for eliminating the fault.

E.2.3 Deadlock-Wait

Jlint

```
DeadlockWait.java:40: Method wait is called from non-
synchronized method
DeadlockWait.java:54: Method notify is called from non-
synchronized method
DeadlockWait.java:31: Method DeadlockWait.run() implement-
ing 'Runnable' interface is not synchronized
Verification completed: 3 reported messages
```

Jlint misses the synchronization statements working on the resources `a` and `b` and therefore misses the actual problem. Reorganizing the source code intro three classes eliminates the warnings (while keeping the problem), showing that Jlint does not check for synchronization problems within methods.

Execution time: very short (< 0.06 s)

New Jlint

```
DeadlockWait.java:40: Method wait() can be invoked with moni-
tor of other object locked. DeadlockWait.java:40: Hold-
ing 2 lock(s): b, a.
DeadlockWait.java:33: Method DeadlockWait.run() implement-
ing 'Runnable' interface is not synchronized. Verifica-
tion completed: 2 reported messages.
```

The first warning, despite consisting of two statements, is counted as one. It precisely states where and why the failure occurs and also gives the current lock set (in the reverse order in which the locks were acquired).

ESC/Java

```
DeadlockWait: run() ...
------------------------------------------------------------------------
DeadlockWait.java:34: Warning: Possible deadlock (Deadlock)
      synchronized (a) {
                    ^
Execution trace information:
    Executed then branch in "DeadlockWait.java", line 33, col 12.
------------------------------------------------------------------------
```

Execution time: 4.9 s

Annotations: 3 (trivial) annotations plus `synchronized` declaration of `run` method

ESC/Java finds the deadlock, but information it gives about it is rather scarce.

VisualThreads

Interestingly, the output was not the same when VisualThreads was ran twice. For better readability, the memory addresses have been replaced with an index, where each index corresponds to a unique memory address.

Run 1:

```
Thread 2: Blocked on DeadlockWait$Lock_1, owned DeadlockWait$Lock_2
Thread 3: Blocked on DeadlockWait$Lock_2
```

Execution time: 22+ s (graph continues with all threads blocked)

Run 2:

```
Thread 2: Blocked on DeadlockWait$Lock_1, owned DeadlockWait$Lock_2
Thread 3: Terminated
```

Execution time: 22+ s (graph continue s with all threads blocked and one thread terminated)

The output is harder to interpret than in the previous examples, because Thread 2 waits on a `notify` call from Thread 3. The POSIX level on which VisualThreads operates is too low to show this Java functionality.

E.2.4 Deadlock-Wait2

Jlint

```
LockB.java:22: Method wait() can be invoked with moni-
tor of other object locked
LockA.java:21: Call se-
quence to method LockB.foo(java.lang.String, boolean) can cause dead-
lock in wait()
Verification completed: 2 reported messages
```

Execution time: very short (< 0.05 s)

After remodeling the problem on a method level, Jlint detects the fatal dependency between the two classes that lock each other's instance.

New Jlint

The new version still finds the fault.

ESC/Java

```
DeadlockWait2: run() ...
-------------------------------------------------------------------------
DeadlockWait2.java:14: Warning: Precondition possibly not estab-
lished (Pre)
     lock.foo(name, ab);
              ^
Associated declaration is "LockA.java", line 16, col 6:
   //@ requires \max(\lockset) <= this;
-------------------------------------------------------------------------
LockA: foo(java.lang.String, boolean) ...
Fatal error: Unexpected exit by Simplify subprocess
```

Execution time: 5.4 s

Annotations: 3

ESC/Java detects the deadlock (i.e. the violation of the assumption that the singleton instance of the LockA class can always obtain an exclusive lock on itself). The reason why Simplify exits abnormally later on is unknown.

VisualThreads

In this case, VisualThreads showed very nicely how the deadlock occurred. Since the Java source code included a `sleep()` call that waited for some time, the deadlock was to happen for sure, and the intermediate phase (with both threads waiting) is shown nicely by VisualThreads. (The delay is higher when run in VisualThreads because the entire virtual machine is slowed down.)

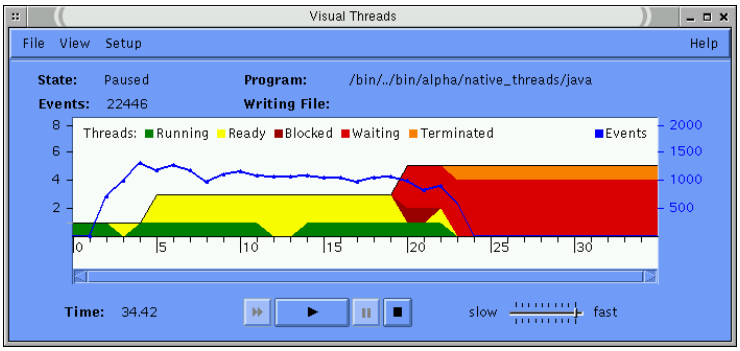

Figure E.2: Graph for DeadlockWait2 produced by VisualThreads.

State 1 (at $T = 21s$):

Thread 2	Blocked on LockA	Owned: DeadlockWait2
Thread 3	Running	Owned: java.lang.Class, LockA

State 2 (at $T \geq 23s$):

Thread 2	Blocked on LockB	Owned: LockA, DeadlockWait2
Thread 3	Terminated	

Execution Time: 23+ s (no state change after that)

While one can assume that the program is not progressing anymore (since no events occur), VisualThreads again gives no hints about `wait` or `notify` methods. However, since all the important classes are directly visible in the output, it should not be too hard to find the fault in a real program.

E.2.5 Deadlock3

Jlint

```
Deadlock3.java:35: Method Deadlock3.run() implement-
ing 'Runnable' interface is not synchronized
Verification completed: 1 reported messages
```

Execution time: very short (< 0.05 s)

Again, Jlint cannot detect deadlocks within a method. After the two preceding examples, it is obvious that Jlint would detect the cycle on a higher level.

New Jlint

```
Deadlock3.java:35: Method Deadlock3.run() implement-
ing 'Runnable' interface is not synchronized.
Deadlock3.java:58: Lock Deadlock3.a is requested while hold-
ing lock Deadlock3.c, with other thread holding Dead-
lock3.a and requesting lock Deadlock3.c.
Deadlock3.java:40: Lock Deadlock3.b is requested while hold-
ing lock Deadlock3.a, with other thread holding Dead-
lock3.b and requesting lock Deadlock3.a.
Deadlock3.java:49: Lock Deadlock3.c is requested while hold-
ing lock Deadlock3.b, with other thread holding Dead-
lock3.c and requesting lock Deadlock3.b.
Verification completed: 4 reported messages.
```

As in the "Deadlock2" example, Jlint detects the fault and gives a precise analysis.

ESC/Java

```
Deadlock3: run() ...
------------------------------------------------------------------------
Deadlock3.java:38: Warning: Possible deadlock (Deadlock)
       synchronized (a) {
                     ^
Execution trace information:
    Executed then branch in "Deadlock3.java", line 37, col 20.
------------------------------------------------------------------------
```

Execution time: 4.9 s

Annotations: 3 annotations and `synchronized` declaration of `run` method in order to get a more useful counter-example.

ESC/Java detects the possible deadlock, but the counter-example information is incomplete and likely not very helpful for a more complex program.

VisualThreads

First run: nothing detected.

Second run:

```
Deadlock occurred with cycle length 6.
Thread 4 waits for Thread 2.
Thread 2 waits for Thread 3.
Thread 3 waits for Thread 4.
```

Execution time: 20 s

Here, a principal weakness of a dynamic checker shows clearly: even with the same input, it is not certain that a deadlock actually occurs. It is still surprising that VisualThreads did not detect the cycle in the lock graph when run for the first time. Is this because the JVM may not report the same memory address for each thread for the static locks? Running the demonstration program written in C suggests that VisualThreads would have detected this fault in a program.

E.2.6 SplitSync

Jlint

```
SplitSync.java:32: Method SplitSync.run() implement-
ing 'Runnable' interface is not synchronized
Verification completed: 1 reported messages
```

Execution time: very short (< 0.05 s)

Jlint gives the usual warning about the `run` method but fails to see the real problem. Even remodeling the problem into two synchronized methods, with two threads using the same instance of that object, does not enable Jlint to recognize this race condition.

New Jlint

The output is unchanged; Jlint was not extended for detecting this fault.

ESC/Java

```
SplitSync: run() ...
------------------------------------------------------------
SplitSync.java:34: Warning: Possible deadlock (Deadlock)
    synchronized (resource) {
                  ^
------------------------------------------------------------
```

Execution time: 4.8 s

Annotations: 3 annotations, `synchronized` declaration of `run` method.

Instead of a possible race condition, ESC/Java detects a non-existent deadlock. Because the `monitored` pragma does not work on static fields, this spurious warning is difficult to eliminate. A try with an altered version of the program, where the static field `resource` was replaced with a singleton instance, did not produce the desired result either.

VisualThreads

VisualThreads was run twice, and no race condition or other problem was detected.

Execution time: 21 s

It was to be expected that such a race condition due to a "gap" in the locking scheme would not be detected by a dynamic checker that does not control the Java thread scheduler.

E.2.7 Jlint test example

Jlint

```
B.java:5: Loop 1: invocation of synchronized method B.g1()
can cause deadlock
B.java:8: Loop 2: invocation of synchronized method A.f1(B)
can cause deadlock
A.java:3: Loop 2: invocation of synchronized method B.g1()
can cause deadlock
B.java:8: Loop 3: invocation of synchronized method A.f1(B)
can cause deadlock
A.java:5: Loop 3/1: invocation of method A.f2(B) forms the
loop in class dependency graph
A.java:8: Loop 3: invocation of synchronized method B.g2()
can cause deadlock
Verification completed: 6 reported messages
```

Execution time: very short (< 0.08 s)

As expected, Jlint finds all possible deadlocks and gives a very helpful report.

New Jlint

The new version still finds the fault.

ESC/Java

```
A.java:4: Warning: Precondition possibly not established (Pre)
    b.g1();
       ^
Associated declaration is "B.java", line 5, col 6:
  //@ requires \max(\lockset) <= this;
      ^
A.java:9: Warning: Precondition possibly not established (Pre)
    b.g2();
       ^
Associated declaration is "B.java", line 10, col 6:
  //@ requires \max(\lockset) <= this;
      ^
B.java:7: Warning: Precondition possibly not established (Pre)
    bp.g1();
       ^
Associated declaration is "B.java", line 5, col 6:
  //@ requires \max(\lockset) <= this;
      ^
```

```
B.java:12: Warning: Precondition possibly not established (Pre)
    ap.f1(bp);
          ^
Associated declaration is "A.java", line 2, col 6:
  //@ requires \max(\lockset) <= this;
      ^
B.java:15: Warning: Possible deadlock (Deadlock)
  public static synchronized void g3() {
                                  ^
5 warnings
```

Execution time: 4.0 s

Annotations: 9 standard annotations for synchronized methods.

This example shows two classes without any context in which they are used. This is why a few `non_null` annotations are required in order to suppress warnings about initialized fields. Even before this is done, ESC/Java warns about possible deadlocks in each method. After adding a few assumptions about the locking order, ESC/Java shows a slightly more detailed warning, but the warning itself does not include enough context to be helpful, and the counter-example is full of low-level information that makes it hard to determine whether the warning refers to a real fault and what the cause might be.

VisualThreads

Since this example was not a full program, it could not be tested with VisualThreads.

E.2.8 ESC/Java example

Jlint

```
Tree.java:16: Loop 1: invocation of synchro-
nized method Tree.visit() can cause deadlock
Tree.java:17: Loop 2: invocation of synchro-
nized method Tree.visit() can cause deadlock
Verification completed: 2 reported messages
```

Execution time: very short (< 0.05 s)

The warnings here are false positives, and only justified if the data structure (a binary tree) is somehow corrupted; otherwise, no circular locks can occur during the recursive visiting.

Jlint correctly gives no warnings for the `wiggleWoggle` method, but it does not seem to analyze the method body at all.

New Jlint

```
Tree.java:50: Lock Tree.right is requested while hold-
ing lock Tree.right, with other thread hold-
ing Tree.right and requesting lock Tree.right.
Tree.java:16: Loop 2: invocation of synchro-
nized method Tree.visit() can cause deadlock.
Tree.java:17: Loop 3: invocation of synchro-
nized method Tree.visit() can cause deadlock.
Verification completed: 3 reported messages.
```

Like ESC/Java, Jlint now prints a spurious warning about a possible Deadlock. This warning does not parse the content of the local variables yet, so it may look rather confusing.

ESC/Java

```
Tree: wiggleWoggle() ...
------------------------------------------------------------
Tree.java:50: Warning: Possible deadlock (Deadlock)
          synchronized (x) {        // line (d)
                       ^
Execution trace information:
    Executed else branch in "Tree.java", line 34, col 6.
    Executed else branch in "Tree.java", line 37, col 8.
    Executed then branch in "Tree.java", line 49, col 23.
------------------------------------------------------------
```

Execution time: 3.8 s

Annotations: 7 (given in the example)

The ESC/Java manual documents the problems with the `wiggleWoggle` method:[1]

> "The problem is that the axiom is assumed to apply at the start of the routine, and thus to apply to the values of `.left` and `.right` at the start of the routine. According to the lock order thus defined, the lock acquired at the line `(d)` would precede that acquired at `(b)`. [Despite these caveats, our experience with ESC for Modula 3 suggests that axioms like the ones above will do the right thing surprisingly often and rarely cause problems.]
>
> The preceding example also illustrates a possible source of unsoundness in ESC/Java's treatment of race detection. If the lines marked `(c)` and `(e)` are deleted, and if deadlock checking is left disabled, then ESC/Java will accept line `(d)` without complaint, ignoring the possibility that some other thread might have taken advantage of the window between lines `(a)` and `(b)` to synchronize on "v" and set its `.right` field to null."

VisualThreads

Again, this program was not a full executable, so one would have to have written a test driver in order to check it with VisualThreads. Since the program had not actual deadlock or race condition, this test was omitted.

E.2.9 Buffer

Jlint

```
BufferWorks.java:49: Method BufferWorks$Consumer.run() imple-
menting 'Runnable' interface is not synchronized
BufferWorks.java:32: Method BufferWorks$Producer.run() imple-
menting 'Runnable' interface is not synchronized
Verification completed: 2 reported messages
```

[1] Listing D.8 with the annotated source is on page 94.

Execution time: very short (< 0.05 s)

Two warnings about bad code design, but no false positives.

New Jlint

Still only design guide warnings, no false positives.

ESC/Java

```
BufferWorks$Producer: run() ...
------------------------------------------------------------------------
BufferWorks.java:38: Warning: Precondition possibly not established (Pre)
            buffer.enq(name);
                       ^
Associated declaration is "BufferWorks.java", line 82, col 8:
    //@ requires \max(\lockset) <= this;
         ^
Execution trace information:
    Reached top of loop after 0 iterations in "BufferWorks.java", line 34, col 1.
    Executed else branch in "BufferWorks.java", line 38, col 5.
------------------------------------------------------------------------
BufferWorks$Consumer: run() ...
------------------------------------------------------------------------
BufferWorks.java:53: Warning: Precondition possibly not established (Pre)
          buffer.deq();
                 ^
Associated declaration is "BufferWorks.java", line 99, col 8:
    //@ requires \max(\lockset) <= this;
         ^
Execution trace information:
    Reached top of loop after 0 iterations in "BufferWorks.java", line 52, col 1.
------------------------------------------------------------------------
```

These two warnings make sense; after all, they are issued at the critical sections of the code. It is not possible to assure (at least not directly) that one consumer (or producer) thread will always eventually make sure that the buffer does not stay full (or empty, respectively).

Execution time: 7.1 s

Annotations: 11

VisualThreads

Two runs were made, with different outputs:

Run 1: No violations detected.

```
Mutex BufferWorks$Buffer was highly contended.
The Overall wait/locked ratio was 1.62314 which ex-
ceeds the analyses threshold of 1.00000.
Threads spent a total of 0.42480 seconds waiting for mu-
tex Buffer-
Works$Buffer which was only locked for 0.26171. On aver-
age, waiters took 0.07080 seconds to acquire the lock.
```

Run 2: No violations detected. [No warnings.]

Execution time: 21 s (in both cases)

It is interesting that ESC/Java detects a possible performance problem in one case and not in another. Nevertheless, it did not report a real fault for this program, so the output is correct in both cases.

E.2.10 BufferIf

Jlint

Jlint produces the same warnings as above, also in the same time, and misses the subtle fault.

New Jlint

Jlint still misses the fault.

ESC/Java

The result is the same as for the Buffer program (see Section E.2.9).

VisualThreads

The tool was run three times, and the fault went undetected each time. This shows that subtle faults that require an unusual timing by the thread scheduler cannot be detected by VisualThreads.

Execution time: 20 s (in all cases)

E.2.11 BufferNotify

Jlint

Again, Jlint produces the usual two warnings without recognizing the fault.

New Jlint

Jlint still misses the fault.

ESC/Java

The result is the same as for the Buffer program (see Section E.2.9).

VisualThreads

- State 1 (at $T = 20s$):

Thread 2	Ready	Owned: java.lang.Class
Thread 3	Block on java.lang.Class	

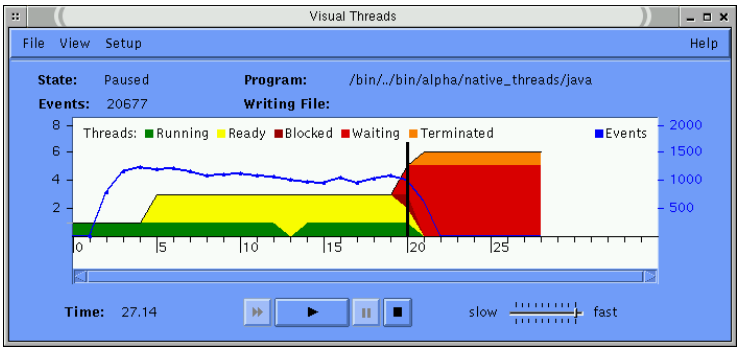

Figure E.3: Graph for BufferNotify produced by VisualThreads

- State 2 (at $T \geq 22s$):

Thread 2	Terminated
Thread 3	Block on Buffer$Notify
Thread 4	Block on Buffer$Notify

Execution time: 22+ s (no state change afterwards)

The result is similar to the DeadlockWait example (Listing D.3), where the `notify` message never receives its correct destination. It is surprising that this failure occurred during dynamic execution. It is likely that the different timing in Java's slowed down thread scheduler was the reason for this.

E.2.12 BufferSem

Jlint

```
BufferTest.java:38: Method BufferTest$Consumer.run() imple-
menting 'Runnable' interface is not synchronized
BufferTest.java:21: Method BufferTest$Producer.run() imple-
menting 'Runnable' interface is not synchronized
Verification completed: 2 reported messages
```

Execution time: very short (< 0.05 s)

The same as for the first correct version of the buffer program: two warnings, no false warnings reported.

New Jlint

Two design warnings, no false positives.

ESC/Java

No warnings.

Annotations: 17 (plus 5 annotations in the Semaphore class)

Execution time: 7.1 s

With the initial set of annotations, ESC/Java would produce a warning that a precondition in the Semaphore class may be violated when the Producer acquires the "empty" (`nonfull`) semaphore. It was then tried to express some dynamic conditions using model variables. However, these properties are too complex to be expressed in the ESC/Java annotation language. In the end, this turned out *not* to be the reason for the issued warnings; the reason was much simpler. Jim Saxe from Compaq gave me advice on how to work around this problem:

> "The problem is that ESC/Java doesn't "know" that a newly-forked thread holds no locks. This is fixed in the new version of ESC/Java, which we plan to release soon. For now, you can work around the problem. (...)
>
> Second, inform ESC/Java that `null` precedes all actual objects in the locking order. (...) The next release of ESC/Java will deal with all this automatically."

VisualThreads

No violations were detected (which is correct).

Execution time: 19 s

E.2.13 PhilosopherDeadlock

Jlint

```
PhilosopherDeadlock.java:36: Method PhilosopherDeadlock.run()
implementing 'Runnable' interface is not synchronized
Verification completed: 1 reported messages
```

Four additional warning messages about a debugging information in the classes Status and Semaphore were also produced; they were always of the type "Field f of class C can be accessed from different threads and is not volatile". While Jlint gives some helpful warnings, it misses the deadlock.

Execution time: very short (< 0.1 s)

New Jlint

Jlint still misses the fault.

ESC/Java

```
PhilosopherDeadlock: take_forks() ...
------------------------------------------------------------
PhilosopherDeadlock.java:50: Warning: Precondition possi-
bly not established (Pre)
    fork[i].down(i);
                ^
Associated declara-
tion is "/home/cartho/java/ipc/Semaphore.java", line 27, col 6:
```

```
  //@ requires \max(\lockset) <= this;
      ^
------------------------------------------------------------
PhilosopherDeadlock.java:58: Warning: Precondition possi-
bly not established (Pre)
    fork[i].up(i);
            ^
Associated declara-
tion is "/home/cartho/java/ipc/Semaphore.java", line 21, col 6:
  //@ requires \max(\lockset) <= this;
      ^
------------------------------------------------------------
```

Execution time: 5.1 s

Annotations: 10

In order to avoid warnings about program properties that were not of concern, the check was run with

```
escjava -warn Deadlock -warn Race -nowarn Exception \
-nowarn IndexNegative -nowarn IndexTooBig PhilosopherDeadlock.java
```

Most of the annotations, where several approaches were tried, deal with warnings about possible `null` pointers. Others are a first approach trying to get rid of "IndexNegative" and "IndexTooBig" warnings. The annotations are certainly not complete enough in order to allow ESC/Java to give a more precise warning. The given warnings may be right, but they originate from insufficient knowledge about the program rather than a thorough analysis. However, expressing the dynamic requirements for being able to obtain a fork (and also the circular structure of the locks) is probably not possible in the ESC/Java annotation language.

VisualThreads

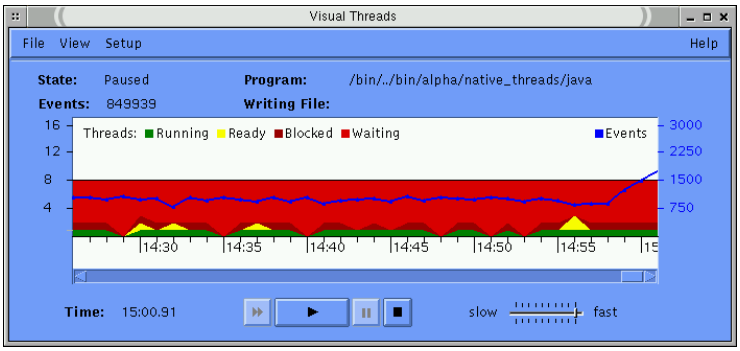

Figure E.4: Alternating thread states in VisualThreads.

Two runs were made with VisualThreads, for 10 and 15 minutes, respectively. During that time, a deadlock would have happened long ago under normal circumstances. Figure E.4 shows, though, that the different threads

just continuously change states, without ever locking each other out for an extended period of time. This is a very strong indicator that the thread scheduling is heavily influenced by the overhead caused by VisualThreads.

Execution time: 10 and 15 minutes (no termination)

E.2.14 Philosopher

Jlint

The reported warnings are the same as in the faulty version, and the execution time is equal.

New Jlint

Output still correct.

ESC/Java

See above.

VisualThreads

The program was run for 5 and 10 minutes, respectively, with the same result as in the previous example. Since this version does not deadlock, this result was to be expected.

E.2.15 PhilosopherHost

Jlint

```
Host.java:17: Method wait is called from non-synchronized method
```

Execution time: very short (< 0.1 s)

There are seven more warnings, which are the same ones as in the previous instances of this problem. This false warning is quite interesting, though: as one can see in Listing D.16 on page 100, the `wait` statement is within a `synchronized(this)` block. Therefore that instance holds a lock to itself, making the `wait` call safe. Again, Jlint misses the synchronization statement.

New Jlint

The spurious warning shown above has been removed. The output is now correct.

ESC/Java

```
Host: request() ...
-------------------------------------------------------------------------
Host.java:15: Warning: Possible assertion failure (Assert)
    /*@ assert \lockset[this] || \max(\lockset) < this */
        ^
-------------------------------------------------------------------------
PhilosopherHost: run() ...
-------------------------------------------------------------------------
PhilosopherHost.java:50: Warning: Precondition possibly not established (Pre)
      host.release();
                  ^
Associated declaration is "Host.java", line 29, col 6:
  //@ requires \max(\lockset) <= this
      ^
Execution trace information:
    Reached top of loop after 0 iterations in "PhilosopherHost.java", line 44, col 4.
-------------------------------------------------------------------------
PhilosopherHost: take_forks() ...
-------------------------------------------------------------------------
PhilosopherHost.java:56: Warning: Precondition possibly not established (Pre)
    fork[i].down(i);
                ^
Associated declaration is "/home/cartho/java/ipc/Semaphore.java", line 27, col 6:
  //@ requires \max(\lockset) <= this;
      ^
-------------------------------------------------------------------------
PhilosopherHost: put_forks() ...
-------------------------------------------------------------------------
PhilosopherHost.java:64: Warning: Precondition possibly not established
    fork[i].up(i);
              ^
Associated declaration is "/home/cartho/java/ipc/Semaphore.java", line
:
  //@ requires \max(\lockset) <= this;
      ^
-------------------------------------------------------------------------
```

Execution time: 5.3 s

Annotations: 15

The `synchronized(lock)` block, which works on a static member, was rather difficult to annotate. The current annotations do not remove spurious warnings about possible deadlocks.[2] Again, the problem is too complex to be expressed in the ESC/Java annotation language.

VisualThreads

Like in the other two examples, VisualThreads did not detect any violations after 5 and 10 minutes time.

[2]It is of course possible that the implementation of the host (`Host.java`) may indeed exhibit a deadlock under certain circumstances, but this has not occurred yet in dynamic testing.

Appendix F

Results of new Jlint

This appendix shows the test results of applying the improved Jlint to various packages. In some cases, the old version of Jlint was used as well to obtain a quantitative comparison.

F.1 Extra Jlint examples

In order to ensure the correctness of the new Jlint features, a few new test programs were written in Java. These programs exhibit various race conditions or deadlock problems.

F.1.1 Race conditions, `wait` problems

Wait.java

Listing F.1 Two faults regarding a `wait` call: Calling `wait` without owning the right lock, while owning other locks (the latter could lead to a deadlock).

```
public class Wait {
  public void a() {
    Object lock = new Object();
    Object lock2 = new Object();
    synchronized(lock) {
      try {
        lock2.wait();
      }
      catch (InterruptedException e) { }
    }
  }
}
```

Old Jlint

```
Wait.java:17: Method wait is called from non-
synchronized method
```

New Jlint

```
Wait.java:17: Method '<new>1.wait' is called without synchro-
nizing on '<new>1'.
Wait.java:17: Method wait() can be invoked with moni-
tor of other object locked.
Wait.java:17: Holding 1 lock(s): <new>0.
```

The old Jlint did not support synchronized blocks; therefore its warning is very general, and frequently a warning of this type is incorrect (although not in this case). The new Jlint detects both faults and reports a precise warning. It has also been successfully tested against correct versions of this program[1], with synchronizations on `this`, instance or class variables.

F.1.2 Deadlocks

Inter-method deadlock

Listing F.2 Deadlock scenario among two methods: methods `foo` and `bar` acquire locks a and b in a conflicting order.

```
public class Deadlock {
  Object a = new Object();
  Object b = new Object();

  public void foo() {
    synchronized (a) {
      synchronized (b) { }
    }
  }

  public void bar() {
    synchronized (b) {
      synchronized (a) { }
    }
  }
}
```

Old Jlint

No warnings.

New Jlint

```
Deadlock.java:13: Lock Deadlock.a is requested while hold-
ing lock Deadlock.b, with other thread holding Dead-
lock.a and requesting lock Deadlock.b.
Deadlock.java:7: Lock Deadlock.b is requested while hold-
ing lock Deadlock.a, with other thread holding Dead-
lock.b and requesting lock Deadlock.a.
```

[1] In these versions, the `wait()` call applies to the object that was just synchronized on, and no other monitors are owned at that time.

Note that since the deadlock analysis starts after all class files have been processed, the order in which the deadlocks are reported may not correspond to the one in the source file.

Listing F.3 More complicated version of the same deadlock: Two threads may call `foo(boolean ab)` with different values of `ab`, entering the two montors in conflicting order.

```
public class Deadlock2 {
  Object a = new Object();
  Object b = new Object();

  public void foo(boolean ab) {
    if (ab) {
      synchronized (a) {
        bar(!ab);
      }
    } else {
      synchronized (b) {
        bar(!ab);
      }
    }
  }

  public void bar(boolean ab) {
    if (ab) {
      synchronized (a) { }
    } else {
      synchronized (b) { }
    }
  }
}
```

Old Jlint

```
Deadlock2.java:12: Comparison always produces the same result
```

Line 12 is the second method call to `bar(!ab)`. Sun's version 1.3 of the Java compiler duplicates the check for the value of `ab`; therefore this warning is caused by inefficient code generation of that compiler rather than a real fault.

New Jlint

```
Deadlock2.java:12: Comparison always produces the same re-
sult.
Deadlock2.java:21: Lock Deadlock2.b is requested while hold-
ing lock Deadlock2.a, with other thread holding Dead-
lock2.b and requesting lock Deadlock2.a.
Deadlock2.java:19: Lock Deadlock2.a is requested while hold-
ing lock Deadlock2.b, with other thread holding Dead-
lock2.a and requesting lock Deadlock2.b.
```

The improved Jlint now also finds this bug, although it does not evaluate the value of the boolean variable `ab`; it does not perform a data flow analysis of

that flag and cannot determine which of the two branches in `bar` would be executed; hence it tries both. This means it would have reported that warning, too, if `bar` was (correctly) called without inverting `ab` first.

F.1.3 Lock reference changes

Example

Listing F.4 Assigning a new value to a lock variable.

```
public class Deadlock3 {
  Object a = new Object();
  Object b = new Object();

  public void foo(boolean ab) {
    if (ab) {
      synchronized (a) {
        bar(!ab);
      }
    } else {
      synchronized (b) {
        bar(!ab);
      }
    }
  }

  public void bar(boolean ab) {
    a = new Object();
    if (ab) {
      synchronized (a) { }
    } else {
      synchronized (b) { }
    }
  }
}
```

Old Jlint

Same output is in previous example.

New Jlint

Additionally to the output of the previous example, the following warning is printed:

```
Deadlock4.java:18: Value of lock a is changed while (poten-
tially) owning it.
```

Again, Jlint cannot distinguish between the two cases where the lock on `a` is already owned and where it is not. (Such an analysis would also greatly slow down the analyzer.)

F.2 Trilogy's source code

The goal was to find as many potential faults as possible, and to avoid spending time checking spurious warnings. Therefore, no test was made using the old Jlint. It is always clear which warnings were given because of the extensions. On the other hand, the number of suppressed warnings due to improved lock analyses is not quantifiable using only the output if the new Jlint.

Because the number of warnings was sometimes very high, usually a filter was applied, in the following form:

```
#!/bin/bash
find . -name '*.class' | xargs jlint -not_overridden \
-redundant -weak_cmp -bounds -zero_operand -string_cmp -shadow_local | \
grep -v 'Field .class\$' | grep -v 'can be .*ed from different threads' | \
grep -v 'Method.*Runnable.*synch'
```

This filter suppresses any warnings that refer to design issues or are likely spurious warnings due to insufficient data flow analysis. In the detailed the following tables, several warnings referring to exactly the same variable were counted as one; warnings that occurred due to Jlint bugs were not counted. The totals in the tables are therefore not consistent with the raw count of filtered warnings. Two packages (catalogsvc and tce2) were not examined because their compilation required third party packages, which were not installed on the system used. Moreover, those two packages did not employ multi-threading as much as the ones analyzed below.

F.2.1 MCC Core (ffcaf)

Unfiltered: 447 warnings.

Filtered: 32 warnings.

See table F.1 for a list of warnings that referred to distinct places in the source code. Out of these warnings, two were confirmed as bugs. Several `null` pointer warnings were in the same module, which was already deprecated and will likely be entirely removed soon because of these warnings.

F.2.2 Cerium (hec)

Unfiltered: 166 warnings.

Filtered: 4 warnings.

Table F.2 shows the warnings that referred to distinct places in the source code.

F.2.3 Log player (sbjni)

Unfiltered: 41 warnings.

Filtered: 1 warning. This warning is because the same lock is acquired twice in two different methods (which is OK). This is incorrectly interpreted by Jlint as two different locks.

Warning category	#	Comment
Lock variable change outside constructor or synchronization.	2	Initialization method; unsafe code but no bug.
Missing `super.finalize()` call.	1	Confirmed bug; fixed.
Shift << with count $\geq$ 32. Possibly incorrect type cast.	1	Two warnings for the same bug: the expansion from `int` to `long` comes *after* the left shift.
Possible loop in locking graph (synchronized methods).	4	Two cases that are probably OK but cannot be checked statically; two cases that need further investigation by someone working on that code.
Possible `NULL` pointer reference because parameter is not checked.	9	Three warnings are definitely wrong from the context; the other warnings are "just unsafe code."

Table F.1: Analysis of Jlint warnings for MCC Core.

Warning category	#	Comment
Lock variable change outside constructor or synchronization.	1	Initialization method; unsafe code but no bug.
Possible `NULL` pointer reference because parameter is not checked.	3	One warning was for a method where `NULL` is a valid parameter; yet that reference was still used later. Maybe no execution path for that scenario exists?

Table F.2: Analysis of Jlint warnings for Cerium.

F.2.4 Java backbone (scbbjava)

Unfiltered: 647 warnings.

Filtered: 44 warnings.

Table F.3 shows the warnings that referred to distinct places in the source code.

Warning category	#	Comment
Lock variable change outside constructor or synchronization.	1	Possibly a bug; not yet verified.
Missing `super.finalize()` call.	11	Bad coding practice in the best case, possibly even a fault.
Synchronized method is overridden by non-synchronized method...	1	As a result of this, an assignment within the overriding method is not guarded by a synchronization; a possible bug.
Possible loop in locking graph (synchronized method).	5	Three false positives, two more complex cases that are probably also working correctly.
Possible `NULL` pointer reference because of unexpected input or state.	2	1) If URL syntax was not correct, `NULL` dereference could occur. Not sure whether URL syntax is tested elsewhere. 2) Happens when a function returns `false`, which is probably never the case under normal circumstances.
Possible `NULL` pointer reference because parameter is not checked.	4	One false positive, three cases of unsafe coding practice.

Table F.3: Analysis of Jlint warnings for the Java backbone.

F.2.5 Trilogy Insurance Calculation Engine (trilogyice)

Unfiltered: 30 warnings.

Filtered: 19 warnings.

See table F.4 for a list of warnings that referred to distinct places in the source code. Out of the four `null` pointer reference warnings, two were confirmed as bugs and fixed. Two `null` pointer warnings referring to the usage of `javax.beans.*` classes were not bugs, because the contract with these classes ensures that the "dangerous" method is never called if those pointers are `null`. This could not be determined statically.

From the seven `null` pointer warnings, three occurred in private methods whose usage is always correct; four occurred in methods where these cases were documented as "undefined behavior". The latter four cases were fixed nonetheless. The potential race condition was confirmed and fixed as well.

Jlint has been remarkably successful in this module: Out of only 19 warnings (some of which were about the same fault in different lines), seven warnings resulted in bug fixes!

Warning category	#	Comment
Possible loop in locking graph (synchronized methods).	4	Four times the same situation: a new instance does not have a reference to the instance that created it; therefore a deadlock in synchronized methods is impossible.
Possible race condition (usually filtered out)	1	Debug output, where a race condition can probably be tolerated.
Possible `NULL` pointer reference because of unexpected input or state.	4	Seem to be faults, but not yet verified.
Possible `NULL` pointer reference because parameter is not checked.	7	In three cases (container class), `NULL` pointer check probably omitted for performance reasons.

Table F.4: Analysis of Jlint warnings for trilogyice.

F.2.6 Summary

Table F.5 shows an overview of the frequency of *all* warnings that Jlint issued for Trilogy's code. While a rather large part of the warnings are not bug detections, most of these warnings are hints to existing trouble spots in the code. At least ten warnings lead to actual changes in the code; a few other warnings, however, resulted in extra comments in the code (which is also valuable).

Warning category	#
Lock variable change outside constructor or synchronization.	3
Missing `super.finalize()` call.	12
Possible loop in locking graph (synchronized methods).	13
Possible `NULL` pointer reference because parameter is not checked.	23
Possible `NULL` pointer reference because of unexpected input or state.	6
Other	4

Table F.5: Summary of Jlint's warnings in Trilogy's code

F.3 Concurrency package

Old Jlint

Unfiltered: 197 warnings.

Filtered: 6 warnings.

New Jlint

Unfiltered: 170 warnings.

Filtered: 82 warnings.

There is a striking difference between the number of warnings reported. This concurrency package makes heavy use of `synchronized` blocks. Hence the refined checks suppress a lot of false positives, while generating a large number of new warnings.

Out of the remaining 82 warnings of the new Jlint, 31 false positives occurred because a `synchronized` block was "interrupted" by in `if` or `return` statement. They were all of type

```
Method x.wait|notify is called without synchro-
nizing on x.
```

This caused several `monitorexit` statements to be present in the source code, although only one could be executed at a time. Even after a `return`, Jlint would continue its analysis with the old context, assuming the lock had already been released. Such false warnings could be eliminated with full flow control.

17 new warnings were obviously caused by a fault in Jlint, all of type

```
Lock x is acquired while holding lock y,
with other thread holding lock y and requesting x.
```

These were only 5 unique warnings, all with wrong line numbers. This failure could not be reproduced with other bytecode. Subtracting these warnings from the total, only 34 remain. Out of these, 26 warnings were potential deadlock warnings that were difficult to verify and probably went beyond the capabilities of a static analyzer (only four such warnings were issued by the original Jlint, because its analysis is coarser). The reason of 5 warnings about invoking `wait()` with having other monitors locked could not be found – possibly this is also due to a bug in Jlint. This leaves three warnings:

```
EDU/oswego/cs/dl/util/concurrent/BoundedLinkedQueue.java:276:
Value of lock last_ is changed while (potentially) owning it.
```

This warning occurs because a new node is inserted into a list – this case is difficult to analyze statically, but the code is correct.

```
EDU/oswego/cs/dl/util/concurrent/BoundedLinkedQueue.java:296:
Method wait() can be invoked with monitor of other object locked.
EDU/oswego/cs/dl/util/concurrent/BoundedLinkedQueue.java:296:
Holding 2 lock(s): <this>, putGuard_.
EDU/oswego/cs/dl/util/concurrent/LinkedQueue.java:70:
Value of lock last_ is changed while (potentially) owning it.
```

The correctness here depends on the correct values of certain counters and the correct functionality of other methods; it could not be verified easily.

Because this package is very complex, and showed many remaining weaknesses in Jlint, statistics of the output would too skewed by these problems.

F.4 ETHZ data warehousing tool

Unfiltered: 150 warnings.

Filtered: 23 warnings.

Most warnings given here were not verified (with the exception of the lock change), because the code was already two months old when the new Jlint was applied to it. The lock change was indeed a fault in the algorithm, and that fault was actually found by manual inspection of intermediate files generated for the statistical analysis (see Appendix A). This success was the reason why such a check was added to Jlint.

Warning category	#	Comment
Lock variable change outside constructor or synchronization.	2	Verified, and fixed.
Possible loop in locking graph (synchronized methods).	1	False warning? Method is not `synchronized`; maybe a fault in Jlint.
Missing `super.finalize()` call.	3	Not verified.
Possible `NULL` pointer reference because of unexpected input or state.	6	Not verified.
Possible `NULL` pointer reference because parameter is not checked.	2	Not verified.

Table F.6: Analysis of Jlint warnings for the ETH data warehousing tool.

Bibliography

[1] SPIN model checker http://netlib.bell-labs.com/netlib/spin/whatispin.html

[2] Bandera http://www.cis.ksu.edu/santos/bandera/

[3] ESC/Java http://research.compaq.com/SRC/esc/

[4] FeaVer http://cm.bell-labs.com/cm/cs/who/gerard/abs.html

[5] Java PathFinder http://ase.arc.nasa.gov/jpf/

[6] SLAM http://research.microsoft.com/slam/

[7] Flavers http://laser.cs.umass.edu/tools/flavers.html

[8] Jlint http://www.ispras.ru/~knizhnik/jlint/ReadMe.htm

[9] Forte for C http://www.sun.com/forte/c/

[10] MC http://hands.stanford.edu/

[11] Modeling and Checking http://www.cis.upenn.edu/~rtg/mac/

[12] Rivet http://sdg.lcs.mit.edu/rivet.html

[13] Verisoft http://www1.bell-labs.com/project/verisoft/

[14] VisualThreads http://www5.compaq.com/products/software/visualthreads/

[15] Java Modeling Language http://www.cs.iastate.edu/~leavens/JML.html

[16] SCARP example repository http://laser.cs.umass.edu/verification-examples/

[17] LOOP tool development http://www.cs.kun.nl/~bart/LOOP/loop_tool.html

[18] Isabelle theorem prover http://www.cl.cam.ac.uk/Research/HVG/Isabelle/

[19] PVS Specification and Verification System http://pvs.csl.sri.com/

[20] Symbolic Model Verifier (SMV) `http://www.cs.cmu.edu/~modelcheck/`

[21] Symbolic Analysis Laboratory `http://verify.stanford.edu/DARPA/sal.html`

[22] `http://www.cs.wisc.edu/~solomon/cs537/processes.html`

[23] Doug Lea's concurrency package `http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html`

[24] HEDC - HESSI Experimental Data Center `http://www.hedc.ethz.ch/`

[25] Graphviz `http://www.research.att.com/sw/tools/graphviz/`

[26] GNU General Public License (GPL) `http://www.gnu.org/copyleft/gpl.html`

[27] Real-Time Systems Group, *MaCware User Manual,* University of Pennsylvania, USA 2000.

[28] I. Lee, S. Kannan, M. Kim. O. Sokolsky, M. Viswanathan, *Runtime Assurance Basd On Formal Specifications,* University of Pennsylvania, USA 1999.

[29] Derek Bruening. *Systematic Testing of Multi-threaded Java Programs.* Master's Thesis, MIT, May 1999

[30] Patrice Godefroid, *VeriSoft Reference Manual,* Bell Laboratories, Lucent Technologies, USA 2000.

[31] Patrice Godefroid, *Model Checking for Programming Languages using VeriSoft,* Bell Laboratories, Lucent Technologies, USA 2000.

[32] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson, *Proceedings of the 16th ACM Symposium on Operating System Principles,* pp. 27-37, Saint Malo, France, October 1997.

[33] Gerard J. Holzmann, *Design And Validation Of Computer Protocols,* Prentice Hall, USA 1991.

[34] Corina S. Pasareanu, Matthew B. Dwyer, and Willem Visser, *Finding Feasible Counter-examples when Model Checking Abstracted Java Programs,* USA 2000.

[35] J.C. Corbett, M.B. Dwyer, J. Hatcliff, and Robby, *Bandera: A source-level interface for model checking Java programs.* In Proc. 22nd International Conference on Software Engineering, June 2000.

[36] K. Rustan M. Leino, Greg Nelson, and James B. Saxe. *ESC/Java User's Manual.* Technical Note 2000-002, Compaq Systems Research Center, October 2000.

[37] K. Rustan M. Leino, James B. Saxe, and Raymie Stata, *Checking Java programs via guarded commands.* Technical Note 1999-002, Compaq Systems Research Center, May 1999.

[38] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe, *Extended Static Checking,* Compaq Systems Research Center, December 1998.

[39] Gerard J. Holzmann, Margaret H. Smith, *A Practical Method for Verifying Event-Driven Software,* Bell Laboratories, USA 2000.

[40] Klaus Havelund, *Java PathFinder User Guide,* NASA Ames Research Center, USA 1999.

[41] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, *Model Checking Programs,* NASA Ames Research Center, USA 1999.

[42] Guillaume Brat, Klaus Havelund, SeungJoon Park, Willem Visser, *Java PathFinder, Second Generation of a Java Model Checker,* NASA Ames Research Center, USA 2000.

[43] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem, *Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions,* Stanford University, USA 2000.

[44] Thomas Ball, Sagar Chaki, Sriram K. Rajamani, *Parametrized Verification of Multithreaded Software,* Microsoft Research, USA 2000.

[45] Gary T. Leavens, Clyde Ruby: *Safely Creating Correct Subclasses without Seeing Superclass Code,* OOPSLA 2000 proceedings, USA 2000.

[46] James C. Corbett, *Evaluating Deadlock Detection Methods for Concurrent Software.* IEEE transactions on software engineering, Vol. 22, No. 3, March 1996.

[47] Scott Oaks, Henry Wong, *Java Threads,* O'Reilly, USA 1997.

[48] David Flanagan, *Java In A Nutshell,* 3rd Ed., O'Reilly, USA 1999.

[49] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, *The Java Language Specification,* 2nd Ed., Addison-Wesley, USA 2000.

[50] Tim Lindholm, Frank Jellin, *The Java Virtual Machine Specification, Second Edition,* Addison-Wesley, USA 1999.

[51] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns. Elements of Reusable Object-Oriented Software,* Addison-Wesley, USA 1994.

[52] McCabe, T., *A software complexity measure,* IEEE transactions on software engineering SE-2(4), pp. 308–20, 1976.

[53] Abraham Silberschatz, Greg Gagne, Peter Baer Galvin, *Applied Operating Systems Concepts,* 1st edition, USA 2000.

[54] Eleftherios Koutsofious, *Drawing graphs with* dot. AT&T Bell Laboratories, USA 1996.

[55] Norman E. Fenton, Shari Lawrence Pfleeger, *Software Metrics – a rigoruous & practical approach,* 2nd Ed., Thomson Computer Press, USA 1996.

[56] IEEE, *IEEE Standard 729: Glossary of Software Engineering Terminology,* IEEE Computer Society Press, 1983.

[57] Turing, A. M., *On computable numbers with an application to the Entscheidungsproblem,* Proc. London Math. Soc. 42, 230–265 and 43, 544–546.

Index

aliasing problem, 5, 27, 36, 53
applicability, 5

Bandera, 9, 77

call graph, 4, 35–40
    extended, 39, 46

deadlock, 2, 15–18, 22, 24, 27, 36–40
dynamic checkers, 11, 51, 74–76
dynamic checking, 3, 59

Eraser, 76
error, 1
ESC/Java, 9, 23, 33, 78
    annotations, 25

failure, 1
fault, 1
FeaVer, 9, 79
Flavers, 9, 79

incompleteness, 4

Java PathFinder, *see* JPF
Jlint, 9, 26, 33, 80
    application, 47
    code changes, 40
    extensions, 35
        implementation, 36
    future extensions, 55
JML/LOOP, 10, 84
JPF, 10, 81

livelock, 2
lock, 2, 6
    change analysis, 36, 40
    graph, 2
LockLint, 10, 82

MaC, 8, 20, 74
MC, 10, 82
model checking, 4

non-determinism, 1, 51
`notify`, 18, 35, 38

race condition, 2, 16–18, 22, 24, 27, 36–40
Rivet, 8, 21, 32, 74

scheduling, 2, 21, 51
SLAM, 10, 83
software metrics, 14
soundness, 5
Spin, 9, 77
static checkers, 11, 52, 77–84
    design, 56
    usage, 53
static checking, 4, 60
statistics, 28, 63–71
`synchronized` block, 27–32, 36–40, 58, 85
`synchronized` method, 27–32

testing, 1
theorem proving, 4
thread-safe, 1

Verisoft, 8, 75
VisualThreads, 9, 22, 33, 76

`wait`, 18, 35, 38