RICE UNIVERSITY

A Framework for Testing Concurrent Programs

by

Mathias Guenter Ricken

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Science

Approved, Thesis Committee:

Robert Cartwright, Chairman
Professor of Computer Science

Walid Taha
Assistant Professor of Computer Science

William Scherer III
Faculty Fellow

Houston, Texas

October 2007

A Framework for Testing Concurrent Programs

Mathias Guenter Ricken

Abstract

Incremental, test-driven development is sweeping the software industry, elevating testing from an ancillary activity to an integral part of the programming process. Unfortunately, in our recent experience developing production programs in Java, unit testing has only proven effective in assuring the reliability of code with a single thread of control; it is much less effective in concurrent programs. To facilitate the development of concurrent programs, we are developing:

1. An extension of the JUnit framework that actively supports the developer by treating tests that could silently ignore failures in auxiliary threads as test errors;

2. A lightweight Java annotation language that can be used to specify and check the threading invariants of both existing and new code;

3. A testing framework that can record and analyze the schedules of unit tests, detect deadlocks, and run the tests using modified schedules, increasing the likelihood that concurrency problems are discovered.

Acknowledgments

I would like to thank my advisor, Robert "Corky" Cartwright. Without his guidance, advice and patience, this project would not have been possible. I thank Corky for his support and understanding, given in regard to this project as well as in private matters. I owe many thanks to the members of my thesis committee, Walid Taha and Bill Scherer, for their crucial suggestions, comments, and encouragements.

I am indebted to Joe Warren for piquing my curiosity and getting me interested in research – and for experiencing informative failure for the first time: not achieving my goal, but learning a lot about it. I thank Jim Kinsey, Don Johnson, and Jim Pomerantz for organizing the Rice Undergraduate Scholar Program, showing me more about research and graduate school, and ultimately tipping the scale.

I also thank Dung "Zung" Nguyen, Stephen Wong, and my other teachers, the department staff, my office mates James Sasitorn and Moez A. Abdel-Gawad, my lab mate Dan Smith, and all my fellow students – my friends – for the often fruitful discussions and all the help along the way. I thank Justin Crites for diligently proofreading and commenting on my thesis and even helping me understand future problems better. His curiosity and energy are infectious, and I hope, for everyone's sake, that it will always stay that way.

I thank the National Science Foundation, the Texas Advanced Technology Program, the School of Engineering and the Department of Computer Science at Rice University for funding this project. I thank Corky for giving me the opportunity

to attend the "Summer School on Language-Based Techniques for Concurrent and Distributed Software" in 2006 in Oregon, ten days of learning, laughing, and hiking to completely new points of view. It was an invaluable experience, and I thank Jeff Foster, Dan Grossman and Jim Allen for making it such a success.

I thank a good friend for being there for me so often. She has been much more important to me than I have been able to show, and perhaps more important than I can comprehend.

Finally, I thank my mother. Words cannot express the love and respect I feel when I think of you. Everything that is good in me can be traced back to you.

Thank you for believing in me, Mama. Ich liebe Dich, so wie ein Sohn nur die beste Mutter der Welt lieben kann.

Every designing mind which has mastered the earthly
Loves in the figure's swing nothing more than the turning point.
(Rainer Maria Rilke, from *Sonnets to Orpheus*)

Contents

Abstract ii

Acknowledgments iii

List of Figures viii

List of Tables ix

List of Listings x

1 Introduction 1

1.1 Review of Prior Work 1

1.2 Motivation 4

1.3 Organization 4

2 Problems Testing Concurrent Programs 6

2.1 Inadequacies of Existing Frameworks 9

2.2 Enforcement of Threading Disciplines 11

2.3 Tractability of Schedule-Based Testing 12

2.4 Common Problems in Concurrent Software 15

3 Improvements to JUnit 21

3.1 Default Exception Handler 22

3.2 Enforcement of Thread Joins 26

3.3 Results 28

4 Annotations for Concurrency Invariants 31

4.1 Examples of Invariants 33

4.2 Annotations and Inheritance 36
4.3 Primitive Annotations 40
4.4 Predicate Annotations 43
4.4.1 Predicate Link Annotations 45
4.4.2 Combine Annotations 52
4.4.3 Predicate Annotations Using Reflection 58
4.5 External Annotations in XML Format 59
4.6 Comparison to Assertions 61
4.7 Subtyping Relation for Annotations 62
4.7.1 Extended Syntax 66
4.7.2 Featherweight Java Subtyping Relation 66
4.7.3 Auxiliary Definitions 67
4.7.4 Combined Subtyping Relation 72
4.7.5 Extension of Featherweight Java Typing 73
4.7.6 Type Preservation and Progress 74
4.7.7 Implementation Differences 74
4.8 Problems Caused By Subclassing: When Is Something Really "Thread-Safe"? 75
4.9 Results 79

5 Bytecode Rewriting Framework 85

5.1 Offline and On-the-Fly Instrumentation 88
5.2 Local and Global Instrumentation 89
5.3 Marker Methods for Instrumentation 91
5.4 Other Uses of Bytecode Rewriting 91
5.4.1 Recording Schedules 92
5.4.2 Deadlock Monitor 94
5.4.3 Random Delays and Yields 95

5.5 Results 101

6 Conclusion **103**

Bibliography **104**

A Pre-Defined Predicate Annotations **109**

B XML Annotation Syntax **122**

B.1 Class-Based XML Format 122

B.2 Invariant-Based XML Format 123

B.3 XML Invariant Notation 124

B.3.1 XML `@PredicateLink` Notation 125

B.3.2 XML `@Combine` Notation 126

B.3.3 XML Argument Notation 126

C Suggestions for Improving Java Annotations **129**

C.1 Annotations of Data 129

C.2 Repeated Annotations 131

C.3 Subtyping for Annotations 133

C.4 Generics for Annotations 135

Figures

2.1 Formula for Number of Schedules 14

4.1 Set of Annotations 40
4.2 Stack for Calling `@PredicateLink` Annotation Predicate 47
4.3 Invariant Checker-Extended Featherweight Java (TCFJ) Syntax . . . 66
4.4 Featherweight Java Subtyping Relation 66
4.5 TCFJ - Extract the Method Declarations from a Class 67
4.6 TCFJ - Extract the Annotations from a Class 67
4.7 TCFJ - Extract the Annotations from a Method 67
4.8 TCFJ - Extract the Name from a Method Declaration 67
4.9 TCFJ - Extract Only Predicate Annotations 68
4.10 TCFJ - Extract Only `@NotRunBy` Annotations 68
4.11 TCFJ - Extract Only `@OnlyRunBy` Annotations 68
4.12 TCFJ - Method Introduction 69
4.13 TCFJ - Annotations Lookup for `Object` 70
4.14 TCFJ - Annotation Lookup for Introduction 70
4.15 TCFJ - Annotation Lookup for Overriding 71
4.16 TCFJ - Check for Annotation Contradictions 71
4.17 TCFJ - Combined Subtyping Relation 72
4.18 TCFJ - Static Invariant Violation 73

5.1 Time for n Synchronized Blocks 102

Tables

4.1 Sample Invariants 63
4.2 More Sample Invariants 64
4.3 Unit Tests 80
4.4 Invariant Checks and Violations 81
4.5 Other Information 82
4.6 Invariant Checker Benchmarks 84

5.1 Schedule Codes 93

A.1 Predicate Annotation Library: `AnySynchronized*` 109
A.2 Predicate Annotation Library: `NoneSynchronized*` 111
A.3 Predicate Annotation Library: `OnlySynchronized*` 112
A.4 Predicate Annotation Library: `NotSynchronized*` 113
A.5 Predicate Annotation Library: `*EventThread` 114
A.6 Predicate Annotation Library: `OnlyThreadWithName*` 115
A.7 Predicate Annotation Library: `NotThreadWithName*` 116
A.8 Predicate Annotation Library: `OnlyThreadWithGroupName*` 117
A.9 Predicate Annotation Library: `NotThreadWithGroupName*` 118
A.10 Predicate Annotation Library: Miscellaneous 119

B.1 XML `@PredicateLink` Member Type Codes 127

Listings

2.1 A Simple Unit Test 8
2.2 A Simple Class to Test 8
2.3 A Flawed Class to Test 9
2.4 A Unit Test Using JUnit 4.2 9
2.5 Program with Race Condition 16
2.6 Shared Data Protected By Lock 17
2.7 Dining Philosophers: Deadlock 18
2.8 Dining Philosophers, Locks in Right Order 19
3.1 Uncaught Exception and Assertion 22
3.2 Uncaught Exception and Assertion in an Auxiliary Thread 23
3.3 Uncaught Exception in an Auxiliary Thread Reached Too Late 27
3.4 Main Thread Waits for Auxiliary Thread to Finish 28
3.5 Concurrency Problems in DrJava Detected by Improved JUnit 30
4.1 Possible Deadlock Due to Lock Acquisition Order 35
4.2 Deadlock Avoided By Correct Order 35
4.3 Listing 4.1 with Annotations to Detect Deadlock 36
4.4 Synchronized Methods Before Transformation 37
4.5 Methods With Synchronized Blocks After Transformation 37
4.6 Annotated Method 38
4.7 Method Annotated in Superclass 38
4.8 Annotated Classes 39
4.9 `@ThreadDesc` Annotation for Use with Primitive Annotations 41
4.10 Illustration of Primitive `@NotRunBy` Annotations 42

4.11 Illustration of Primitive `@OnlyRunBy` Annotations 43
4.12 `@PredicateLink` Meta-Annotation 45
4.13 Memberless Annotation, 0 Parameters, Arguments not Passed 47
4.14 Annotation Definition of Listing 4.13 48
4.15 Predicate Method of Listing 4.13 48
4.16 Usage Site of Listing 4.13 48
4.17 Annotation with Members, 1 Parameter, Arguments not Passed . . . 49
4.18 Memberless Annotation, 2 Parameters, Arguments Passed 50
4.19 Annotation with Member, 2 Parameters, Arguments Passed 50
4.20 Predicate Accessing Private Field Using Nested Class 51
4.21 Predicate Accessing Private Field Using Reflection 52
4.22 Ideal, But Unachievable Usage Example 52
4.23 `@Combine` Meta-Annotation 53
4.24 A `@Combine` Annotation With Default Values 56
4.25 A `@Combine` Annotation Used Twice 57
4.26 XML File in Class-Based Format 60
4.27 XML File in Invariant-Based Format 60
4.28 Listing 4.30 with an Assertion 62
4.29 Benign Program 75
4.30 Modified Behavior by Subclassing 76
4.31 Modified Behavior by Call to Overridden Method 77
4.32 Listing 4.30 with Annotation (Excerpt) 79
4.33 Listing 4.31 with Annotation (Excerpt) 79
4.34 Invariant Checker Output for Listing 4.32 79
4.35 Invariant Checker Output for Listing 4.33 80
5.1 `IInstrumentationStrategy` Source 86
5.2 `ConditionalStrategy` Source 87
5.3 `IScannerStrategy` Interface 87

5.4 “Too Late” Faulty Program 97
5.5 Fault Detected by Random Wait 98
5.6 “Too Early” Faulty Program 99
5.7 Possible Atomicity Hazard 100
B.1 Class-Based XML Header 122
B.2 Class-Based XML Footer 122
B.3 Class-Based XML Body: Class 123
B.4 Class-Based XML Body: Methods 123
B.5 Invariant-Based XML Header 123
B.6 Invariant-Based XML Footer 123
B.7 Invariant-Based XML Body: Method 124
B.8 `@PredicateLink` Annotation in XML 125
B.9 `@Combine` Annotation in XML 126
B.10 Non-Array `<arg>` XML Node 126
B.11 Array `<arg>` XML Node 128
C.1 Using a Dummy Method to Check Data 131
C.2 Annotation for Listing C.1 131
C.3 Predicate for Listing C.2 131
C.4 Invariant Checker Output for Listing C.1 132
C.5 Repeated Annotations 132
C.6 Regular Expressions as Alternative 133
C.7 Annotation Array as Alternative 133
C.8 Extending Annotations 133
C.9 Subtyping for Annotations 134
C.10 Annotation Definitions for Listing C.9 134
C.11 Usage Example for Listing C.9 135
C.12 Generics for Annotations 135

Chapter 1

Introduction

The software industry is embracing test-driven development, popularized under the banner of "Extreme Programming" [1, 2]. When tests for each program unit are written before the unit is implemented, testing can influence the design of a program and catch bugs early in its development. Together with automated build systems like Ant [3], unit tests also provide a basis for refactoring a program while preserving its reliability.

Incremental design and unit testing have proven to be effective for programs with a single thread of control; however, our experience with programming larger applications, such as DrJava [4], has shown that this approach is much less reliable at detecting bugs in programs with multiple threads: Thread scheduling is non-deterministic and machine-specific, making it difficult to create meaningful unit tests.

These problems are exacerbated as CPU clock frequencies stagnate and multi-core processors become more widespread. To benefit from these newer CPUs, programs have to run several computations in parallel. Unless there is a breakthrough in CPU design and manufacturing, writing and testing concurrent code is bound to change from an expert activity to a skill that programmers use every day.

1.1 Review of Prior Work

The topic of testing programs and detecting bugs has been extensively discussed in the literature; recent papers have even covered debugging concurrent programs [5, 6, 7]. The aspect of unit testing concurrent programs, though, is still largely unexplored.

There have been several articles in the Java trade press [8, 9, 10, 11], but none of these publications provides a framework for thoroughly unit testing concurrent code.

In his Master's thesis [12] and a technical report [13], Bruening approaches testing concurrent programs by using schedule-based execution. Using a lockset algorithm based on Eraser [14], his framework ensures that the program is race-free. In parallel, the framework then identifies the critical operations in the program where one thread can have an impact on the computation of other threads, and if necessary, the framework re-executes the program from that point on with a different scheduling choice. As long as the program is race-free, executing all possible arrangements of program blocks delimited by these critical operations is sufficient for covering all possible concurrent behaviors.

Bruening's work focuses on testing entire applications when they are nearly complete; in essence, his framework allows acceptance testing for concurrent programs. To do this, the framework has to have the ability to "roll back" to a previous point, since re-executing the entire program to that point may be too expensive or even impossible. The emergence of "Extreme Programming" [1, 2], however, has put a greater emphasis on testing smaller program units at an earlier stage of the development process. Since unit tests are short and predictable, it seems like the dynamic techniques of schedule-based execution developed by Bruening and the dynamic race detection by Savage et al may be easier to utilize with unit tests. There are several frameworks available that support unit testing, JUnit [15] and TestNG [16] for Java for example, but none of these frameworks takes concurrency into account. Without an improved unit testing framework, applying the research on schedule-based execution is a compromised endeavor.

Aside from these dynamic techniques to improve the reliability of concurrent programs, several static approaches have been suggested. Many of them [17, 18], however, rely on manual annotations to the program to guide the static analysis. This by itself is not necessarily bad: After all, this very thesis proposes an annotation language for

detecting concurrency problems as well. It is nonetheless important to identify this cost because static guarantees at compile time are often seen as getting something for free. Some statement is guaranteed to always be true, and the program itself does not need to ensure that this is actually the case. The fact that these static analyzers require annotations as guidance should serve as a reminder that they, in fact, do not solve the problem of bug detection for free.

It is interesting to point out that Java Specification Request (JSR) 305 [19] proposes the use of Java annotations to specify invariants and detect violations. This request, however, is still in its infancy.

Other approaches to ensuring program correctness, even in concurrent situations, use specialized type systems [20, 21, 22, 23]. The benefits of a highly developed type system, though, are often paid for by greater difficulties in implementing the desired algorithms; complex types can become hurdles that the programmer needs to overcome. Furthermore, these type systems are often experimental and not close to being integrated into a language commonly used by the industry, such as C, C++ or Java.

Another approach to simplifying concurrent software completely eschews locks: These techniques are typically grouped as lock-free or wait-free algorithms [24, 25, 26, 27, 28]. At the core, these algorithms share an optimistic notion about the work they perform; each thread assumes it will succeed without interference from other threads. Without the help of locks, the thread performs all of its work, and in a final step uses a device like an atomic N-way compare-and-swap to produce either change at a large scale or a failure due to interference. In that latter case, the thread will redo its work.

Libraries providing some lock-free algorithms are available for many platforms (Java 5.0 introduced the `java.util.concurrent` package, for example). However, lock-free algorithms need to be re-developed for each standard data structure to which we have become accustomed, and not all data structures are supported yet.

While *using* lock-free algorithms may be easier than using one with locks, *designing* a lock-free algorithm is a complex task. We doubt locks can be completely eliminated in application programs. Furthermore, lock-free algorithms require robust testing in concurrent situations; therefore, the tools developed for this thesis can be helpful to the developer of lock-free algorithms.

1.2 Motivation

Our work was motivated by the problems the programmers of DrJava face during the development of a large program with many threads of control. Our goal was to provide a set of tools that help developers write better concurrent software by enabling them to write correct unit tests, document and verify the concurrency invariants in a piece of software, and simulate the execution of the unit tests using a number of parameters that influence thread scheduling. This framework should not remain a theoretical exercise but be of immediate use to programmers.

1.3 Organization

The first part of this thesis, chapter 2, examines the problems during the development of concurrent programs. It introduces the basics of unit testing, the shortcomings of existing unit testing frameworks, and the difficulties in documenting and maintaining invariants crucial for executing programs correctly. Chapter 2 also discusses how feasible it is to comprehensively test concurrent programs.

Chapter 3 and chapter 4 describe the contributions of this thesis:

- an improved unit testing framework based on JUnit (chapter 3)
- an annotation language for documenting and verifying the concurrency invariants of a program (chapter 4)

Some of the more technical issues and details of the implementation are relegated to chapter 5. This chapter also describes some of the other tools we have created

as part of this project, for example a utility to record the schedule a program has executed, ways to change the schedule by inserting random yields or delays, and a deadlock monitor.

The conclusion in chapter 6 contains a summary of this work and examines opportunities for future research.

Appendix A documents the library of pre-written annotations that have been developed for this thesis. Appendix B describes the syntax of XML files to specify invariants. Appendix C discusses some of the problems related to Java annotations that we encountered during our implementation and makes several suggestions how Java annotations could be improved.

Chapter 2

Problems Testing Concurrent Programs

Unit testing at an early stage has several advantages when compared to traditional acceptance testing, which usually happens when the code of the application is complete:

- By focusing on individual parts, such as methods or classes, unit tests check code segments that are much smaller than the entire application. This automatically localizes errors in the program, since there is no interference from other code segments. Unit testing is particularly effective in combination with code coverage tools like Clover [29], which can ensure that the unit tests actually execute every line of the program's code.

- Unit tests are executable programs, so testing can easily be automated and test failures be reported. Since acceptance testing often involves programs with a graphical user interface (GUI), automation is possible, but much harder, since the actions of a user, such as mouse clicks or keystrokes, have to be simulated.

- By beginning to test earlier and only committing code that passes all unit tests, flaws in the code are found earlier and usually by the same developer who is writing the code. This reduces the importance of being familiar with all parts of the program, facilitating concurrent development by multiple programmers.

- Since unit tests remain in place even when the application code is changed, the program can be refactored safely. Should a change introduce an error, a unit test will fail, giving the developers the opportunity to change the code and make it adhere to previous specifications.

- It is common practice to first write a unit test that exhibits a reported bug – a test that reliably fails because of the problem – and only then fix the bug. The new unit test is added to the code repository together with the bug fix and therefore prevents the problem from occurring again. Should a developer re-introduce the problem, the new unit test will fail.

- Unit tests also provide usage examples of the code they test and can therefore be considered part of the documentation. Studying the unit tests can greatly simplify understanding how a program unit should be used and what its common problems are.

Open-source unit testing frameworks are available for most languages; for Java, JUnit [15] and TestNG [16] are frequently used.

Listing 2.1 shows a simple unit test using JUnit 3.8.2 that tests the `Square.get` method, provided in listing 2.2. The unit test shows a typical list of assertions for this kind of method: A couple of positive integers, a positive non-integral number, and a negative number are used as inputs to the function.

The expected output of each function call is given as first argument to `assert Equals`, the actual result of the function call as second argument, and since this test has to work with inexact floating-point arithmetic, the acceptable deviation from the expected value is given as third argument.

Together with the `Square` class shown in listing 2.2, the unit test will pass since all assertions are met. If, however, an incorrect implementation of squaring is used, like the one shown in listing 2.3, one or more assertions, and the unit test as a whole, will fail. In the particular example in listing 2.3, the function works correctly for integral input values only, and the assertion expecting a result of `.25` for the input value `.5` will fail.

This is only a simple demonstration of unit testing and perhaps seems unnecessary. Our experience with production programming for DrJava [4], however, has shown that

```
import junit.framework.TestCase;
public class SquareTest extends TestCase {
  public void testSquare() {
    // parameters are: expected value, actual value
    // and acceptable deviation (delta)
    assertEquals(4.0, Square.get(2), 0.001);
    assertEquals(9.0, Square.get(3), 0.001);
    assertEquals(6.25, Square.get(2.5), 0.001);
    assertEquals(16.0, Square.get(-4), 0.001);
  }
}
```

Listing 2.1: A Simple Unit Test

```
public class Square {
  public static double get(double x) {
    return x*x;
  }
}
```

Listing 2.2: A Simple Class to Test

unit testing simple methods is an invaluable tool for ensuring that more complicated, composed methods function correctly.

JUnit 3.8.2 and older versions impose some restrictions on the developer: All methods to be run as unit tests have to be `public void` and their names have to begin with `test`. The class containing the methods also has to be a subclass of `junit.framework.Test`. TestNG and JUnit since version 4.0 freed the developer of these limitations by using Java annotations to mark test methods. The disadvantage of this approach is lost compatibility with Java versions prior to 5.0. We have provided improved versions of JUnit 3.8.2 and 4.2, so programmers using Java 1.4 or older can still benefit from this work, while programmers using Java 5.0 or newer may claim all the advantages provided by JUnit 4.2. Listing 2.4 shows the same unit test from listing 2.1 written using JUnit 4.2.

```
public class Square {                                          1
  public static double get(double x) {                         2
    // note: this only works for integral values of x!         3
    return (int)(x*x);                                         4
  }                                                            5
}                                                              6
```

Listing 2.3: A Flawed Class to Test

```
import static org.junit.Assert.assertEquals;                   1
import org.junit.Test;                                         2
public class SquareTest4 { // no need to subclass              3
  @Test // annotation to mark test methods                     4
  public void doSquare() { // test prefix not required         5
     assertEquals(4.0, Square.get(2), 0.001);                  6
     assertEquals(9.0, Square.get(3), 0.001);                  7
     assertEquals(6.25, Square.get(2.5), 0.001);               8
     assertEquals(16.0, Square.get(-4), 0.001);                9
  }                                                            10
}                                                              11
```

Listing 2.4: A Unit Test Using JUnit 4.2

2.1 Inadequacies of Existing Frameworks

As shown above, powerful unit testing frameworks are already available. All of these frameworks are designed for testing code with a single thread of control, though. There are some extensions that address multithreading to a certain degree, GroboUtils [30] and parallel-junit [31], for example, but they focus on running *several unit tests concurrently*, not on running a *concurrent unit test*. Running several tests in parallel may expose some flaws that occur in concurrent situations, but even with these extensions, the current frameworks are deficient: They cannot reliably unit test concurrent software.

Unit testing depends on the fact that the input and output are known in advance and that the entire execution is deterministic. In concurrent programs, however, thread scheduling introduces an element of nondeterminism. To get the full value of unit testing, the process must be deterministic. The approach that we suggest is

schedule-based execution combined with race detection. We define a schedule as a total order over all synchronization actions during the execution of the unit test. If several schedules exist for a test, the unit testing framework executes all of them to ensure that the unit test succeeds regardless of the schedule chosen.

The race detection algorithm that is run in parallel ensures that programs are data-race-free: All accesses to a shared variable are consistently protected by the same lock. The Java Memory Model [32] guarantees that data-race-free Java programs exhibit sequential consistency [33, 34]; in other words, every possible execution of such a program, regardless of the number of processors, corresponds to an "as-if-serial" execution for some schedule. Requiring correctly synchronized programs is a reasonable demand: Good programming practices already insist on correct synchronization, as a data race often is the result of a bug. Furthermore, the Java Memory Model does not guarantee sequential consistency for a program with a data race, making it harder to reason about the program's behavior.

In order to produce meaningful results using schedule-based execution, the framework first needs to reliably test programs in a single schedule, the one that was nondeterministically chosen by the platform for this run. This is not an easy task with the existing frameworks: Exceptions and failed assertions that occur in threads other than the main thread are ignored, and the frameworks do not ensure that child threads terminate before the test is declared a success.

The software developed as part of this project includes an improved version of the JUnit framework, available as plug-in for JUnit 4.2 and self-contained replacements of JUnit 3.8.2 and 4.2. It adds the ability to detect exceptions and failed assertions in all threads and ensures that spawned threads have terminated when the test ends. These improvements are described in detail in chapter 3.

2.2 Enforcement of Threading Disciplines

To prevent possible data loss, developers of concurrent software often need to follow a certain discipline of locking data structures or calling methods only from certain threads. At the same time, they have to be cautious when using locks, since a lock owned by one thread may prevent other threads from making progress. In the worst case, several threads own locks and wait for other threads to release their locks, preventing all threads from making progress. This situation is called deadlock.

The second of the additions to JUnit described above, ensuring that all spawned threads have terminated when the test ends, can be viewed as maintaining such a threading discipline.

Java's GUI frameworks, AWT and Swing, require developers to follow an especially complex discipline. Some methods may only be called in the event thread, a special thread that serializes changing the GUI and reacting to user input. This is necessary because concurrent access by a regular thread and the event thread handling user input may cause data loss or put the program in an inconsistent state.

On the other hand, some methods may not be called from within the event thread, because doing so would cause a deadlock. Many methods fall in between these two extremes: Some methods may be called from any thread as long as the GUI has not been displayed yet, and other methods are even considered completely "thread-safe" (a discussion of "thread-safe" methods can be found in section 4.8).

Additionally, applications add their own classes and data structures that have to be kept synchronized, so they define their own threading disciplines. In DrJava, for example, an extra lock has to be acquired before text is inserted into a document: The application maintains two representations of the data, and both have to be consistent.

Unfortunately, these threading disciplines are often only poorly documented, perhaps in Javadoc comments, regular comments inside a method, a white paper, or not at all. This makes it difficult for developers to educate themselves about the required threading discipline and harder to ensure that they actually adhere to it.

Chapter 4 describes a light-weight language, based on Java 5.0 annotations, that allows programmers to describe the required threading discipline in the form of invariants. The language is concise, extensible and easy to use. It serves both as authoritative documentation of the threading discipline and basis for the automated verification that the threading discipline is indeed being followed. For this verification, our tools extract the invariants from the annotations and rewrite the bytecode of the class files to perform the necessary checks. While some problems are already detected at the time of rewriting, most checks are performed at runtime and produce a log detailing which invariant was violated, where the violation happened, and why.

2.3 Tractability of Schedule-Based Testing

The improvements outlined in the previous two sections are only able to detect problems in the schedule under which the test was actually executed. They do not detect problems in all schedules.

The desire to obtain such a general statement – that the unit tests pass no matter what schedule is chosen – was what motivated this project in the first place. If the contributions of this thesis only find problems in one schedule, then the improvements fall short of the goal. Why are these enhancements nonetheless important, even crucial, for testing concurrent programs?

The enhancements of JUnit described here are important, for without them, the framework is unable to detect all unit test failures even in the one schedule that was chosen; failures are only detected if they occur in the main thread. Except for the rare exceptions noted, the improved version of JUnit records uncaught exceptions and failed assertions in any thread.

The same is true for the verification of threading invariants: Without the annotation language, it is difficult to verify whether a program is obeying the threading discipline in the schedule nondeterministically chosen. With the annotation language

and the invariant checker in place, the claim can be made that at least this one schedule did not violate any invariants.

Armed with these strong claims about one particular schedule, the execution of the program under several different schedules can provide even stronger claims. If the tests are executed under all possible schedules, then the general statement can be made that the unit tests succeed regardless of the scheduling choice. The detection of all uncaught exceptions, failed assertions and violated invariants in one particular schedule therefore is a necessity for detecting them in all possible schedules.

Unfortunately, the number of schedules grows exponentially with the number of threads and the number of times a context switch is performed. For simplicity's sake, we assume in the following calculation that all threads are interrupted by a context switch the same number of times, and that they do not interact with each other. We also assume that the scheduler is free to schedule the threads in any order and does not have to maintain some measure of fairness. On an actual system, the scheduler is more constrained and the calculation more complex.

Let t be the number of threads and s be the number of time slices per thread. The number of possible schedules N can then be calculated as a product of s-combinations. The formula for N, given in figure 2.1, has been derived by first choosing the s time slices, out of the total ts time slices, during which the first thread executes; there are $\binom{ts}{s}$ different choices. Then the s time slices for the second thread are chosen, out of the remaining $ts - s$ ones; there are $\binom{ts-s}{s}$ ways to do that. This process continues until only s time slices remain for the last thread.

Stirling's approximation, $n! \approx \sqrt{2\pi n}\frac{n^n}{e^n}$, demonstrates that the number of schedules grows exponentially with both the number of threads t and the number of time slices s: The square root factor grows polynomially; in the second factor, both the numerator and the denominator grow exponentially, but the numerator dominates.

For example, if there are two threads, and each thread consists of two time slices ($t = 2, s = 2$), then there are $\frac{4!}{(2!)^2} = \frac{24}{4} = 6$ different schedules. If there are three

$$
\begin{aligned}
N &= \prod_{x=0}^{t-1} \binom{(t-x)s}{s} \\
&= \binom{ts}{s}\binom{(t-1)s}{s}\binom{(t-2)s}{s}\cdots\binom{2s}{s}\binom{s}{s} \\
&= \frac{(ts)!}{s!(ts-s)!}\frac{((t-1)s)!}{s!(ts-2s)!}\frac{((t-2)s)!}{s!(ts-3s)!}\cdots\frac{(2s)!}{s!s!}\frac{s!}{s!\,0!}
\end{aligned}
$$

The second term in the denominator of one fraction cancels out the term in the numerator of the next fraction, resulting in the simpler term.

$$
\begin{aligned}
&= \frac{(ts)!}{s!}\frac{1}{s!}\frac{1}{s!}\cdots\frac{1}{s!}\frac{1}{s!} \\
&= \frac{(ts)!}{(s!)^t}
\end{aligned}
$$

Figure 2.1 : Formula for Number of Schedules

threads ($t = 3, s = 2$), there are already $\frac{6!}{(2!)^3} = \frac{720}{8} = 90$ different schedules. It becomes apparent that the execution of all schedules quickly becomes intractable.

The number of possible schedules should be taken with a grain of salt:

- While the problem of executing all schedules is intractable for large programs, unit tests are much smaller than entire applications, resulting in a significant drop of possible schedules.

- If each shared variable is protected by at least one lock when accessed, then it is sufficient to test all interleavings of blocks between accesses to shared variables, acquiring or releasing locks, and other actions that influence concurrent behavior [12]. Adherence to this so-called mutual exclusion locking discipline can be verified by running a lockset algorithm in parallel [14].

- Unit tests are written to be non-interactive, so it is not as crucial to provide results immediately as it is with programs that undergo acceptance testing. This allows unit tests to be run unattended over night.

Even if the execution of the unit tests under all schedules takes a long time, the benefits of finding problems that only occur in rare schedules outweigh the costs of

debugging the program after a release, especially as computers become faster and concurrent programming even more frequent.

2.4 Common Problems in Concurrent Software

One of the most common problems in concurrent software is a race condition. A race conditions occurs when two or more threads access the same data at the same time and at least one of them is a write access. This situation is more common than one might think, since even a simple increment operation like `++counter` contains both a read and a write operation, as shown here in pseudo-code: { `move counter to register; increment register; move register to counter;` }.

In listing 2.5, ten threads each increment a shared counter 100,000 times. At the end, the expected value of the counter is 1,000,000, but we obtained 894,936 and many other values as results.

This deviation is a result of unfortunate and nondeterministic thread switches that occur in the middle of the `++counter` operation. Here is one sequence of operations that results in data loss: Thread A moves 0, the counter's value, into a register, but is then preempted by thread B. Thread B itself also moves 0, the counter's value, into a register, increments the value to 1, and writes 1 back into the counter. Now thread A resumes, increments the value it had stored in its register, which was 0, and writes the result back into the counter. After two increments, the counter contains 1, not 2 as expected. Thread A has written stale data.

The different threads mutate shared data, and whenever that is the case, the threads need to be synchronized in some way to prevent interference due to race conditions. In Java, each object has a "lock" (or "monitor") that only one thread can acquire at a time, and only the owner is allowed into a `synchronized` block or method protected by the lock. When a thread encounters a `synchronized` block and no other thread owns the lock, the thread acquires the lock, executes the code inside the block, and releases the lock again upon leaving. If a thread encounters

```
public class RaceCondition {                                        1
  static int counter = 0;                                           2
  static int threadNum = 10;                                        3
  static int iterationsPerThread = 100000;                          4
  static class Incrementor implements Runnable {                    5
    public void run() {                                             6
      for(int i=0; i<iterationsPerThread; ++i) {                    7
        ++counter;                                                  8
      }                                                             9
    }                                                               10
  }                                                                 11
  public static void main(String[] args) {                          12
    Thread[] t = new Thread[threadNum];                             13
    for(int i=0; i<threadNum; ++i) {                                14
      t[i]=new Thread(new Incrementor()); t[i].start(); }           15
    try { for(int i=0; i<threadNum; ++i) { t[i].join(); } }         16
    catch(InterruptedException e) { }                               17
    System.out.println("Counter: "+counter);                        18
    System.out.println("Expected: "+                                19
                       (threadNum*iterationsPerThread));            20
  }                                                                 21
}                                                                   22
```

Listing 2.5: Program with Race Condition

a `synchronized` block, but some other thread already owns the lock, the thread is added to a set of threads waiting to acquire the lock; when the lock becomes available again, one of the waiting threads is allowed to acquire it. Listing 2.6 shows how a `synchronized` method can prevent the race condition from listing 2.5: Instead of incrementing the counter directly, the threads now call a `synchronized` method `inc()` to perform the increment operation.

The changes in listing 2.6 remove the race condition using a lock. Locks, however, have their own set of problems, most notably deadlock.

A deadlock is created if two or more threads each own a resource (like a lock), but they need to acquire another resource to continue, which is already owned by another thread: Thread A owns the first lock, Thread B owns the second lock. To proceed, thread A needs to acquire the second lock, already owned by thread B, and thread B

```
static synchronized void inc() { ++counter; }
static class Incrementor implements Runnable {
  public void run() {
    for(int i=0; i<iterationsPerThread; ++i) inc();
  }
}
```

Listing 2.6: Shared Data Protected By Lock

needs to acquire the first lock, already owned by thread A. Neither thread can make any progress, both are doomed to wait forever.

One of the most famous problem that illustrates the concept of deadlock is the "Dining Philosophers" problem: The philosophers have spaghetti for lunch, and they each grab the fork to their right. The spaghetti is so slippery, though, that it is impossible to eat with just one fork. Each philosopher decides to borrow the fork of the philosopher on his left once it becomes available, but since all philosophers hold on to the fork on their right, the university is forced to hire a completely new Philosophy department.

Listing 2.7 shows a program simulating the "Dining Philosophers" that is likely to produce a deadlock. Each philosopher is represented by a separate thread that first grabs the lock (=fork) to the right (=at the specified index). Then it attempts to grab the lock to the left (=at index+1, or if that is outside the array, at 0), and when it has both locks, the thread terminates (=eats). By the time a thread attempts to acquire the lock to the left, it is likely that another thread has already acquired it, creating a deadlock.

One attempt to solve this problem is to let a philosopher realize he cannot eat because he only has the right fork and cannot acquire the left fork, and then make him drop the right fork. This should allow the person sitting to the right of that self-sacrificing philosopher to pick up the fork and eat, but this may not always work either. It is possible that all philosophers realize this at the same time, drop their forks, and then start from the beginning by grabbing the fork on their right. Since

```
public class Philosophers {
  static final int N = 5;
  static Object[] forks = new Object[5];
  static class Philosopher implements Runnable {
    int right, left;
    public Philosopher(int i) { right = i; left = (right+1)%N; }
    public void run() {
      synchronized(forks[right]) {
        System.out.println("Philosopher "+right+" has "+
          "fork "+right+", tries to get fork "+left);
        synchronized(forks[left]) {
          System.out.println("Philosopher "+right+" eats!");
  } } } }
  public static void main(String[] args) {
    Thread[] philosopher = new Thread[N];
    for(int i=0; i<N; ++i) { forks[i] = new Object();
      philosopher[i] = new Thread(new Philosopher(i));
      philosopher[i].start();
    }
    try { for(int i=0; i<N; ++i) { philosopher[i].join(); } }
    catch(InterruptedException e) { }
    System.out.println("All done eating.");
  }
}
```

Listing 2.7: Dining Philosophers: Deadlock

the philosophers keep doing something, there is no deadlock, but no progress is made either. This situation is called livelock.

The main problem in this implementation is the order in which the locks are acquired. Thread 0 first acquires lock 0, then tries to acquire lock 1; thread 4 first acquires lock 4, then tries to get lock 0, which is already owned by thread 0. In the program's wait-for graph (a graph that shows the threads, which locks they own and which they need), this order of always grabbing the right lock first completes a cycle and therefore the program enters a deadlock.

The most common solution for avoiding deadlock lets threads always acquire locks in the same order. In this example, thread 4 should have tried to acquire lock 0 first, not lock 4. This enabled thread 3 to acquire lock 4 and proceed. Listing 2.8 shows the necessary changes.

```
synchronized(forks[right<left?right:left]) {                     1
  System.out.println(‘‘Philosopher ‘‘+right+’’ has ‘‘+            2
    ‘‘fork ‘‘+right+’’, tries to get fork ‘‘+left);               3
  synchronized(forks[right<left?left:right]) {                    4
    System.out.println(‘‘Philosopher ‘‘+right+’’ eats!’’);        5
  }                                                               6
```

Listing 2.8: Dining Philosophers, Locks in Right Order

Another solution has just one lock for the entire table, one that guards access to forks in general. All philosophers attempt to acquire that lock, but only one will. That philosopher dines and releases the table lock when he is done. This enables someone else to acquire the table lock and eat.

This solution works, but it is not very efficient: Only one philosopher can eat at a time. In essence, the table lock has serialized all the threads, and they behave as if there were just one thread at all. The locking scheme can have a profound impact on the performance of the system.

A possible improvement releases the table lock once two forks have been picked up. Then the next philosopher can acquire the table lock and attempt to pick up two forks, which may or may not work, depending on who else is eating at the time. If the philosopher has two forks available, he can drop the table lock and begin to eat; if the philosopher cannot acquire two forks, he does not eat, releases the table lock and gets back in line to re-acquire the lock.

This scheme also works, but it is not without flaws either: The system does not guarantee fairness. An unfortunate philosopher might always find himself with fewer than two forks.

Solving concurrency problems is not always as simple as in the “Dining Philosophers” example, with the array of locks in plain sight. Developers therefore need to be particularly careful when they combine two or more operations that each use locks, and when different means of communication between threads are mixed. In order to avoid deadlocks and livelocks, to maintain high performance and to guaran-

tee fairness, the threading discipline has to be meticulously documented. Since many programs use unit testing and have extensive test suites, these properties should be checked at the same time the unit tests are run.

Chapter 3

Improvements to JUnit

Despite the proven effectiveness of current unit testing frameworks for programs with a single thread of control, we have found them difficult to use in multithreded programs. Writing good unit tests for concurrent programs is hard, for several reasons:

- Thread scheduling is non-deterministic and machine-specific, so the outcome of a unit test may change from one run to the next.
- The non-determinism makes it hard to reproduce problems that only occur in particular schedules, and even harder to ensure that the unit tests pass under all possible schedules.
- Attempting to simulate a test under different schedules by adding additional locks or `wait-notify` communication between the test and the framework results in unwieldy code.

As a result, a successful unit test provides only little assurance, and only a unit test failure imparts tangible information to the developer. A failure proves that a problem exists in the program, but a unit test success does not prove that the unit test will always succeed. To make matters worse, the existing unit testing frameworks do not correctly deal with concurrent programs and may completely ignore errors that occur in threads other than the main thread. This chapter describes two defects in current unit testing frameworks and then introduces Concutest-JUnit, an improved version of JUnit that does not share these deficiencies.

```
import junit.framework.TestCase;                                     1
public class TestInMainThread extends TestCase {                     2
  public void testException() {                                      3
   // uncaught, causes failure                                        4
   throw new RuntimeException();                                      5
  }                                                                   6
  public void testAssertion() {                                      7
    // fails, causes failure                                          8
    assertTrue(false);                                                9
  }                                                                   10
}                                                                     11
```

Listing 3.1: Uncaught Exception and Assertion

3.1 Default Exception Handler

The most serious problem of using JUnit to test concurrent programs is the lack of reporting uncaught exceptions in auxiliary threads. A new child thread does not have a default exception handler installed; therefore, unless a thrown exception is caught somewhere in the program, the exception will unwind the thread's stack, invoke Java's own exception handler to print a message, and then terminate the thread. Other threads are not automatically notified of this, so an uncaught exception in an auxiliary thread will be completely unnoticed by the main thread and JUnit, even though the same code executed in the main thread would lead to a test failure. Since JUnit's assertions, like `assertEquals` shown in listings 2.1 and 2.4, are implemented using exceptions, failed assertions will not be noticed either.

Listings 3.1 and 3.2 illustrate this problem in greater detail: The unit tests in both listings contain two methods, `testException` and `testAssertion`. In both listings, `testException` throws an exception that is not caught anywhere, while `testAssertion` makes an assertion that is guaranteed to fail. Therefore, in both listings 3.1 and 3.2, the two test methods should produce failures. In listing 3.2, however, an auxiliary thread throws the exception and makes the assertion, not the main thread as in listing 3.1. JUnit is never informed of the uncaught exception or the failed assertion and declares both test methods successful.

```
import junit.framework.TestCase;                                  1
public class TestInOtherThread extends TestCase {                 2
  public void testException() {                                   3
    new Thread(new Runnable() {                                   4
      public void run() {                                         5
        // uncaught, should cause failure but does not            6
        throw new RuntimeException();                             7
      }                                                           8
    }).start();                                                   9
  }                                                               10
  public void testAssertion() {                                   11
    new Thread(new Runnable() {                                   12
      public void run() {                                         13
        // fails, should cause failure but does not               14
        assertTrue(false);                                        15
      }                                                           16
    }).start();                                                   17
  }                                                               18
}                                                                 19
```

Listing 3.2: Uncaught Exception and Assertion in an Auxiliary Thread

To remedy this problem, our modified Concutest-JUnit framework creates a new thread group with an overridden `ThreadGroup.uncaughtException` method. The framework then spawns a new thread in this thread group, executes the test in the new thread, and waits for the test to complete. If an uncaught exception is thrown in the test's thread or any of its child threads, the overridden `uncaughtException` method stores the exception to make it accessible to Concutest-JUnit. When the Concutest-JUnit framework resumes execution, it checks whether an uncaught exception has occurred and deals with the exception appropriately, just as if it had occurred in the main thread.

The use of a thread group is essential for two reasons:

- When a parent thread spawns a child thread, the parent's thread group is inherited by the child thread (unless a specific thread group is passed to the child's constructor; more about this below). This assigns the same thread group to auxiliary threads as to the test's main thread; therefore, uncaught exceptions

in auxiliary threads also invoke the overridden `uncaughtException` method, and uncaught exceptions and failed assertions are no longer ignored.

- Before Java 5.0 introduced the `setDefaultUncaughtExceptionHandler` method, using a thread group was the only way to catch exceptions in threads other than the current thread. While the feature introduced with Java 5.0 is easier to use, thread groups offer compatibility with older versions of Java. Thread groups are also more robust than the new `setDefaultUncaughtExceptionHandler` method: There is only one default uncaught exception handler, and in order to function correctly, the Concutest-JUnit framework would have to prevent the program from changing it. A program that creates thread groups, on the other hand, is not affected because of the hierarchical nature of thread groups.

Thread groups were introduced to the Java API to process threads and their descendants collectively. The Concutest-JUnit framework uses this feature to record uncaught exceptions in all threads that a test spawns.

There are a few problems with this approach:

1. Programmers can supply their own thread groups when creating threads, thereby overriding the thread group installed by Concutest-JUnit. This is normally unproblematic, since a thread group created in one of the test's threads is a descendant of Concutest-JUnit's thread group, which is still informed about all the uncaught exceptions it should know about. Which exceptions need to be recorded by Concutest-JUnit depends on where they occur and whether the `uncaughtException` method of the new thread group has been overridden:

   a. If the `uncaughtException` method has been overridden by the programmer, then the intent has been declared that the program should handle uncaught exceptions itself. These uncaught exceptions may therefore not be recorded by Concutest-JUnit.

   b. The only place where uncaught exceptions should be reported to Concutest-JUnit is in the overridden `ıuncaughtException` method itself. Unfortunately, uncaught exceptions thrown there do not get processed by Java at all. We argue that this is an oversight in the Java Language Specification [35] and that the parent thread group's `uncaughtException` method should be invoked.

   c. If the `uncaughtException` method has not been overridden, then the basic behavior of the `ThreadGroup.uncaughtException` method will automatically call the `uncaughtException` method in the parent thread group, and the exceptions are correctly registered.

2. It is possible to create a new thread group that is not a descendant of the current thread group. If the programmer deliberately creates a thread group that does not descend from Concutest-JUnit's thread group, then exceptions could go unnoticed by our Concutest-JUnit framework.

3. Uncaught exceptions thrown in the `uncaughtException` method of an application's `Thread.UncaughtExceptionHandler` cannot be processed, since the Java virtual machine ignores them [36]. Again, we believe this is an oversight in the Java Language Specification [35] and that the `uncaughtException` method of the thread group should be invoked.

While the problems described in 1.b., 2. and 3. are real, the probability of accidentally ignoring uncaught exceptions is low: Most code does not use thread groups at all (in March 2007, Koders [37], a source code search engine, found 913 matches for "ThreadGroup" in the Java source code it had scanned, compared to 49,329 matches for "Thread"), does not override the `uncaughtException` method (in March 2007, Koders reported 32 method definitions as matches for "uncaughtException"), and does not create thread groups that do not descend from the current thread group. For nearly all programs, the Concutest-JUnit framework can report all uncaught ex-

ceptions. Furthermore, all uncaught exceptions reported by the original framework are reported by our improved framework.

It is important to understand that these improvements will not detect all uncaught exceptions that *could* occur; only the uncaught exceptions thrown in the chosen schedule are found. It is possible that the program can be executed under a different schedule and fail. Adding a default exception handler to Concutest-JUnit is nonetheless a crucial step in creating a framework suitable for testing concurrent programs.

3.2 Enforcement of Thread Joins

If JUnit is modified as described in section 3.1, the Concutest-JUnit framework is now able to detect uncaught exceptions in auxiliary threads. Unfortunately, listing 3.2 exhibits another problem often found in tests of concurrent software: The test does not ensure that the auxiliary threads spawned in `testException` and `testAssertion` finish before the test ends and is declared a success. The two test methods in listing 3.2 may or may not be successful, but the test method in listing 3.3 is almost guaranteed to succeed even though it should fail: Because of the call to `Thread.sleep`, the auxiliary thread is unlikely to reach its point of failure in time.

It is clear that this problem is caused by the test's main thread not waiting for auxiliary threads to finish before the test ends. A correctly written test should ensure this by having a `Thread.join` call for every thread that it spawned; therefore, it is logical to declare as invalid a test that finishes before all of its child threads have terminated.

To address this issue and increase the framework's ability to detect problematic tests, the Concutest-JUnit framework checks if any child threads are still running. The framework enumerates the remaining threads in the Concutest-JUnit thread group and declares the test a failure if running threads are found after the test ended. To

```
import junit.framework.TestCase;
public class TestInOtherThreadSleep extends TestCase {
  public void testException() {
    new Thread(new Runnable() {
      public void run() {
        // sleep for 10 seconds
        try { Thread.sleep(10*1000); }
        catch(InterruptedException ioe) { /* ignore */ }
        // uncaught, should cause failure but does not
        throw new RuntimeException();
      }
    }).start();
    // test's main thread exits immediately
  }
}
```

Listing 3.3: Uncaught Exception in an Auxiliary Thread Reached Too Late

help the programmer fix the incorrectly written test case, still running child threads are listed, along with their current stack traces.

Some threads are excluded from the list of threads and are allowed to outlive the test's main thread:

- Daemon threads are automatically shut down once all non-daemon threads of an application have terminated. In a unit test, they continue to run, though, because the unit test is not a stand-alone application. It is therefore reasonable to allow daemon threads to remain active even after the test has ended.

- Some system threads, namely those part of the AWT and RMI (remote method invocation) libraries, may be created inside the thread group, but this happens automatically and without the programmer's knowledge. Just like daemon threads, they would terminate automatically once the application finishes. For the reasons explained above, these threads are allowed to remain active after the test has ended:

  - `AWT-Shutdown`

```
import junit.framework.TestCase;
public class TestInOtherThreadSleepJoin extends TestCase {
  public void testException() {
    final Thread t = new Thread(new Runnable() {
      public void run() {
        // sleep for 10 seconds
        try { Thread.sleep(10*1000); }
        catch(InterruptedException ie) { /* ignore */ }
        // uncaught, causes test to fail
        throw new RuntimeException();
      }
    });
    t.start();
    // main thread waits for spawned thread to finish
    try { t.join(); }
    catch(InterruptedException ie) { /* ignore */ }
  }
}
```

Listing 3.4: Main Thread Waits for Auxiliary Thread to Finish

- AWT event threads, i.e. threads starting with `AWT-EventQueue-`
- `RMI Reaper`
- `DestroyJavaVM`

The improvements added in this section find the active child thread in listing 3.3 and declare the test a failure. Listing 3.4 shows a correctly written version of the test: Even though the auxiliary thread sleeps for 10 seconds, the uncaught exception after the sleep is detected and causes the test to fail since the main thread waits for the auxiliary thread to terminate.

3.3 Results

We used the 3.8.2 version of Concutest-JUnit as drop-in replacement for the `junit.jar` that was previously used in the DrJava build process. Most tests passed without problems; however, several tests that used RMI failed, and the requirement for child threads to terminate first had to be relaxed. This can be done by attaching `_NOJOIN`

to a method name in the 3.8.2 version or setting `@Test(forceThreadJoin=false)` in the modified version of JUnit 4.2. Other than that, two problems were found, as shown in the unit test output in listing 3.5.

Both problems are minor: In the first problem, a thread communicating with a second virtual machine is waiting for more input and is not terminated. In the second problem, there are several reader and writer threads, and the test does not ensure that all of them finish before the test is over.

It was easy to integrate Concutest-JUnit into the existing DrJava project and use it instead of the original `junit.jar`. We did not expect many problems, since the DrJava developers had already created a much more complicated solution to deal with multithreading and exceptions in other threads.

It has always been possible to write concurrent unit tests, but it has been very difficult to write them well. Despite the complicated DrJava code and extensive unit tests, the Concutest-JUnit library was able to detect several flaws.

Concutest-JUnit works on all three major platforms:
Windows, Linux, and Mac OS X.
All source code is open source and available at: `http://www.concutest.org/` [38].

```
Thread Timer-0 (java.util.TimerThread) is still alive: state=
   WAITING
        java.lang.Object.wait(Native Method)
        java.lang.Object.wait(Object.java:474)
        java.util.TimerThread.mainLoop(Timer.java:483)
        java.util.TimerThread.run(Timer.java:462)
Thread Wait for Interactions to Exit Thread (edu.rice.cs.util.
   newjvm.AbstractMasterJVM$1) is still alive: state=RUNNABLE
        java.lang.ProcessImpl.waitFor(Native Method)
        edu.rice.cs.util.newjvm.AbstractMasterJVM$1.run(
           AbstractMasterJVM.java:197)

Testcase: testInterpretCurrentInteractionWithIncompleteInput(edu.
   rice.cs.drjava.model.repl.InteractionsModelTest):      Caused
   an ERROR
The test did not perform a join on all spawned threads.


Thread Thread-2 (edu.rice.cs.util.
   ReaderWriterLockTest$PrinterReaderThread) is still alive:
   state=TERMINATED

Testcase: testMultipleReaders(edu.rice.cs.util.
   ReaderWriterLockTest):   Caused an ERROR
The test did not perform a join on all spawned threads.
```

Listing 3.5: Concurrency Problems in DrJava Detected by Improved JUnit

Chapter 4

Annotations for Concurrency Invariants

The improvements described in the previous chapter only address problems with the unit testing frameworks currently available. These frameworks are not designed for concurrency, even though concurrent programs are becoming more prevalent.

One of the areas that first became overwhelmingly concurrent was the graphical user interface (GUI) of a program. In AWT/Swing, Java's GUI frameworks, the main program could be doing one thing, such as setting up the user interface components, but once that has been done, a second thread, the `"AWT-EventThread"`, is invoked every time a button is pressed or a selection made. To maintain a responsive user interface, programs performing long computations use both the event thread and a worker thread in the background; therefore, they are concurrent. This serves as an example of how widespread concurrent programs have become.

Along with the problems of testing concurrent programs comes the task of defining, documenting, and enforcing a threading discipline. A threading discipline is defined as a set of rules that dictate which threads must acquire what sets of locks before they may access data.

There are many commonly used examples of these threading disciplines, and a reasonable variety is found in AWT/Swing: For example, the Javadoc documentation states that all the methods defined first in Java's `TreeModel`, `DefaultTreeModel`, `TreeNode`, `MutableTreeNode`, and many other classes associated with the model side of Swing's tree component may only be called from the event thread. The same applies to classes that belong to Swing's table model, and some methods involved in model-to-view coordinate conversion. On the other hand, calling `SwingUtilities.`

`invokeAndWait` from the event thread is a recipe for an instantaneous deadlock, since the event thread will wait until the specified task has been completed by the event thread – but the event thread will never even attempt to complete it, because it was told to wait. Both of these limitations define threading disciplines, and these examples merely came from the Swing GUI library; applications and other libraries usually have their own disciplines that need to be followed.

Unfortunately, these threading disciplines are often undocumented, hidden in source code comments, or only found in a white paper about the library. Many times, the authors of concurrent code use several of these approaches to communicate the necessary circumstances for safe access, but there is no way the threading disciplines are enforced. Disobeying a library's threading discipline often does not result in an informative error message, but instead goes unnoticed until much later, when the code has grown and changed, clouding the actual cause of the problem.

This chapter specifies a light-weight language, using Java 5.0 annotations, that allows threading invariants to be applied to methods which require the caller or subclass to adhere to a threading discipline. The invariants are directed outward: They specify contracts that code using the annotated methods must uphold. Most of the time, the verification of adherence is done at runtime, but some verification can be done statically at compile time.

Both the verification of complicated threading invariants at runtime, and the limited static analysis at compile time offer a great benefit to library developers and users: A library developer can precisely describe the required threading invariants, not in comments or white papers, but as Java annotations that can be automatically checked. The users of a library can determine whether they are violating the library's invariants at runtime, if not even statically at compile time. Considering that writing and using extensible, multi-threaded libraries is one of the most challenging programming tasks, these annotations add a lot of value to a library.

This project provides two kinds of annotations: primitive annotations, described in section 4.3, and predicate annotations, described in section 4.4. Primitive annotations allow the detection of contradicting invariants and perform in a very predictable way, but they may not be able to codify all the invariants a program requires. Primitive annotations can only express invariants based on thread names, IDs and whether a thread is an event thread. Predicate annotations offer more flexibility to the programmer, but require judicious use.

The verification of a program's conformity with the threading discipline described by the annotations is performed by automatically inserted bytecode. Since this bytecode "applies instruments" to measure certain aspects of the program, the act of modifying the program's code is called "instrumentation". Different ways of rewriting the program are encapsulated in "instrumentation strategies".

While working extensively with Java 5.0 annotations, we found them to be quite limited in many aspects and have provided suggestions on how they can be improved in appendix C.

4.1 Examples of Invariants

We first consider some of the threading invariants that an application may need to maintain. Invariants are often library- or application-specific, but there are some invariants that occur commonly:

- The method has to be executed in the event thread. This is the case for many methods in Java's AWT and Swing GUI libraries.
- A common relaxation to the preceding invariant is that any thread may call a method of a GUI component before it has been displayed, or "realized" in Java jargon. Once a component has been realized, only the event thread may call a method.

- On the other hand, some methods must not be executed in the event thread. `SwingUtilities.invokeAndWait` has already mentioned as an example.

- Both threads and thread groups can be given names in Java. It is conceivable that some tasks may only be executed by threads or thread groups with a certain name. For example, an application may want to limit the execution of certain methods to threads whose name starts with `"SecureThread"`. Code loaded from an untrusted source could be run in a thread with a different name and therefore be prevented from calling the annotated methods.

- Commonly, only one specific thread may execute a method. The above example uses names, but this convention lacks the precision to specify exactly one instance of a thread. Using custom annotations, the programmer can compare the current thread to a thread stored in a field and allow only that thread to execute the method.

- Often, a certain lock has to be acquired before a method may be executed. This is done automatically in `synchronized` methods, but in many cases, it may be preferable to leave a method unsynchronized and use explicit `synchronized` blocks to limit the time a lock is held. This is particularly true if several locks have to be acquired: If the wrong order is used, the application may deadlock. With automatically acquired locks, the programmer does not have direct control over the order, and a simple program could deadlock under an unfortunate schedule. If threads `t1` and `t2` in listing 4.1 are interrupted each time after they acquire a lock, the program will deadlock. Listing 4.2 leaves the methods unsynchronized and avoids this problem.

- Similarly, the annotations can check that a certain lock is not yet held, as the programmer may know that this would lead to a deadlock. Listing 4.3 shows how listing 4.1 can be annotated to express the requirement that no other thread owns `lock1` or `lock2`, respectively, because the annotated method will need it.

```
public class SyncMethodOrder {                                          1
  public static interface BoolRunnable {                                2
    public void run(boolean stop); }                                    3
  static BoolRunnable lock1 = new BoolRunnable() {                      4
    public void run(boolean stop) { // own lock1, try lock2             5
      if (!stop) { lock2.run(true); } }                                 6
  };                                                                    7
  static BoolRunnable lock2 = new BoolRunnable() {                      8
    public void run(boolean stop) { // own lock2, try lock1             9
      if (!stop) { lock2.run(true); } }                                 10
  };                                                                    11
  public static void main(String[] args) {                              12
    Thread t1 = new Thread(new Runnable() {                             13
      public void run() { lock1.run(false); }                           14
    }); t1.start();                                                     15
    Thread t2 = new Thread(new Runnable() {                             16
      public void run() { lock2.run(false); }                           17
    }); t2.start();                                                     18
    try { t1.join(); t2.join(); }                                       19
    catch(InterruptedException ioe) { }                                 20
  }                                                                     21
}                                                                       22
```

Listing 4.1: Possible Deadlock Due to Lock Acquisition Order

```
public class SyncBlockOrder {                                           1
  // no synchronization here                                            2
  static BoolRunnable lock1 = new BoolRunnable() {                      3
    public void run(boolean s) { if (!s) lock2.run(true); }             4
  };                                                                    5
  static BoolRunnable lock2 = new BoolRunnable() {                      6
    public void run(boolean s) { if (!s) lock1.run(true); }             7
  };                                                                    8
  public static void main(String[] args) {                              9
    Thread t1 = new Thread(new Runnable() {                             10
      public void run() { // synchronization here                       11
        synchronized(lock1) { synchronized(lock2) {                     12
          lock1.run(false); } } } });                                   13
    Thread t2 = new Thread(new Runnable() {                             14
      public void run() { // and here, in same order                    15
        synchronized(lock1) { synchronized(lock2) {                     16
          lock2.run(false); } } } });                                   17
    t1.start(); t2.start();                                             18
    try { t1.join(); t2.join(); }                                       19
    catch(InterruptedException ioe) { }                                 20
  }                                                                     21
}                                                                       22
```

Listing 4.2: Deadlock Avoided By Correct Order

```java
public class SyncMethodOrder {
  static BoolRunnable lock1 = new BoolRunnable() {
    @NoneSynchronizedField(SyncMethodOrder.class, "lock2")
    public void run(boolean stop) { /* ... */ }
  };
  static BoolRunnable lock2 = new BoolRunnable() {
    @NoneSynchronizedField(SyncMethodOrder.class, "lock1")
    public void run(boolean stop) { /* ... */ }
  };
  // ...
}
```

Listing 4.3: Listing 4.1 with Annotations to Detect Deadlock

4.2 Annotations and Inheritance

To express the invariants, we have elected to use Java annotations, a facility introduced with Java 5.0 to store meta-data, information about the program itself.

Compared to the alternative approach of using comments with special formatting, annotations have the advantage of being part of the Java language; therefore, their syntax is checked by the Java compiler, and their content is accessible by using the Java API or by reading the well-defined format of a Java class file. Furthermore, it is possible to restrict the places where annotations can be applied: The annotations designed for the invariant checker can only be applied to types (classes and interfaces), methods and constructors. An annotation facility based on comments would involve much more processing to parse annotations and detect badly placed ones.

The invariants introduced in this section are checked at the beginning of a method, right when it is entered, except in constructors, which make a `super` call first. Synchronized methods are changed to regular methods with a `synchronized` block around the entire method body. This allows the invariant checks to be executed before the lock is acquired: The `synchronized` methods in listing 4.4 are transformed into the methods shown in listing 4.5. All of this is done using bytecode rewriting, either by an offline tool or on-the-fly by a custom class loader at the time a class is loaded into memory.

```
class C {                                                         1
  synchronized void nonS() { /* some code */ }                    2
  static synchronized void s() { /* some code */ }                3
}                                                                 4
```

Listing 4.4: Synchronized Methods Before Transformation

```
class C {                                                         1
  void nonS() { synchronized(this) { /* some code */ } }          2
  static void s() { synchronized(C.class) { /* some code */ } }   3
}                                                                 4
```

Listing 4.5: Methods With Synchronized Blocks After Transformation

A method can acquire an invariant in three different ways:

1. The method itself is annotated. Listing 4.6 shows and example of this.

2. The same method in a superclass or one of the implemented interfaces has been annotated. Once a method has been assigned an invariant, all overriding implementations will be assigned the same invariant. An example can be found in listing 4.7.

3. The class or interface in which the method is first introduced is annotated. The invariant will be assigned to the same method in all classes or interfaces that extend or implement the annotated class. Examples are shown in listing 4.8. Note that a class annotation is only shorthand for annotating all methods in the class.

```
class C {
 @OnlyEventThread void someMethod() {
    // Method may only be run in the event thread
} }
```

Listing 4.6: Annotated Method

```
class C {
  @OnlyEventThread public void someMethod() {
    // Method may only be run in the event thread
  }
}
interface I {
  // Method may only be run in the event thread
  @OnlyEventThread public void otherMethod();
}
class D extends C implements I {
  public void someMethod() {
    // Method may only be run in the event thread
    // D.someMethod() hasn't been annotated, but C.someMethod(),
    // the same method in the superclass, has
  }
  public void otherMethod() {
    // Method may only be run in the event thread
    // D.otherMethod() hasn't been annotated, but
    // I.otherMethod(), the same method in an implemented
    // interface, has
  }
}
```

Listing 4.7: Method Annotated in Superclass

```
class B { public void nothing() { /* no annotation at all */ } }
@OnlyEventThread class C extends B {
  public void nothing() { // no annotation at all
    // even though class C is annotated, because nothing()
    // already defined in the superclass
  }
  public void someMethod() { // May only run in event thread
    // C.someMethod() hasn't been annotated, but the class
    // in which it was introduced has
} }
@OnlyEventThread interface I { // May only run in event thread
  // I.otherMethod() has not been annotated, but the interface
  // in which it was introduced has
  public void otherMethod();
}
class D extends C implements I {
  public void nothing() { /* still no annotation at all */ }
  // someMethod() and otherMethod() may only run in event thread
  // they weren't annotated itself, but the superclass or
  // interface where they were already defined, were annotated
  public void someMethod() { /* ... */ }
  public void otherMethod() { /* ... */ }
}
```

Listing 4.8: Annotated Classes

More formally, let A be an annotation, let m be a method, and let C, D, S be classes or interfaces. Let $m(C)$ be the set of methods defined (i.e. introduced or overridden) in class C. Let $a(C)$ be the set of annotations that are directly attached to class C, i.e. that appear in front of C's class definition, and let $a(C, m)$ be the set of annotations that are directly attached to method m in class C, i.e. that appear in front of m's method definition in the class definition of C. Then $\mathit{annotations}(C, m)$, shown in figure 4.1, is the set of annotations that are applied to a method m in class C, either because the method was directly annotated or because the annotations were somehow inherited.

The first subset contains all annotations attached directly to methods. Because of the reflexive property of subtyping $<:$, this subset contains both annotations from

$$\begin{aligned}
annotations(C,m) \quad = \quad & \{A: \quad \exists S \text{ such that } C <: S, A \in a(S,m)\} \cup \\
& \{A: \quad \exists D \text{ such that } C <: D \wedge \\
& \qquad \nexists S \text{ such that } D <: S, D \neq S, m \in m(S), \\
& \qquad A \in a(D)\}
\end{aligned}$$

Figure 4.1 : Set of Annotations

methods in superclasses and class C itself. The second subset contains annotations are attached to methods because the class in which they were introduced was annotated. More precisely, the second subset consists of all the annotations that meet the following three criteria: the method exists in the class, superclass, or an implemented interface; that class, superclass, or interface is annotated with the annotation; and the method has not already been introduced in a class or interface higher up.

This structure of inheritance of invariants is crucial in enforcing invariants in subclasses. It allows library developers to design APIs that users can extend while still ensuring that the original threading discipline is maintained.

The decision to let annotations on classes only affect methods that are first introduced in that class or one of its subclasses helps localize the effect of an annotation. The users of a library are free to introduce their own invariants, in addition to the library's invariants, without accidentally strengthening the invariants of methods that have already been defined in a library superclass or interface. It should be noted that an annotation on a class is just shorthand for annotating every method in the class and its subclasses.

4.3 Primitive Annotations

Initially, we decided to only allow invariants that use the currently executing thread to determine whether a certain method is allowed to be run or not, using the `@OnlyRunBy` and `@NotRunBy` annotations. Threads are distinguished by

```
public @interface ThreadDesc {                                                1
    /** @return regular expression for the thread name. */                   2
    String name() default "";                                                 3
    /** @return regular expressions for thread group name. */               4
    String group() default "";                                                5
    /** @return ID number as returned by Thread.getID(). */                  6
    long id() default -1;                                                     7
    /** @return designates if the event thread has been                      8
      * selected (for use inside @OnlyRunBy). */                             9
    OnlyRunBy.EVENT_THREAD eventThread()                                      10
      default OnlyRunBy.EVENT_THREAD.NO;                                      11
}                                                                             12
```

Listing 4.9: `@ThreadDesc` Annotation for Use with Primitive Annotations

- name
- group name
- id
- whether the thread was an AWT event thread

The `@OnlyRunBy` and `@NotRunBy` annotations store an array of `@ThreadDesc` annotations, which actually describe the thread using one of the four criteria listed above. This way, it is possible to specify more than one invariant per method. Listing 4.9 shows the implementation of the `@ThreadDesc` annotation.

The thread descriptions of an `@OnlyRunBy` annotation are combined using “or”; that means they are seen as acceptable alternatives, and only one of their invariants has to be met. The descriptions of a `@NotRunBy` annotation are combined using “and”, and all of the invariants must be met for an application to not violate the threading discipline.

The Concutest framework uses bytecode rewriting to implement the invariant checks: An offline tool or custom class loader inserts calls to pre-written methods at the beginning of all affected constructors and methods. Inside those methods, the current thread’s information is compared to the data in the annotation, and if a

```
@NotRunBy(@ThreadDesc(name="foo"))                          1
public class ThreadCheckSample {                            2
    @NotRunBy({@ThreadDesc(group=".*ai.*")})                3
    public ThreadCheckSample() {                            4
        System.out.println("ctor");                         5
    }                                                       6
    @NotRunBy({                                             7
      @ThreadDesc(name="bar"),                              8
      @ThreadDesc(id=1)})                                   9
    public void run() {                                     10
        System.out.println("run");                          11
    }                                                       12
    @NotRunBy({                                             13
      @ThreadDesc(name="main"),                             14
      @ThreadDesc(name="fum")})                             15
    public static void main(String[] args) {                16
        System.out.println("main");                         17
        (new ThreadCheckSample()).run();                    18
    }                                                       19
}                                                           20
```

Listing 4.10: Illustration of Primitive `@NotRunBy` Annotations

violation is detected, it is added to a log file. For `@NotRunBy` annotations, there is one call per `@ThreadDesc`, and the description from the annotation is passed as parameter and compared immediately. Since threads described in an `@OnlyRunBy` annotation are seen as alternatives, a conclusion about whether a violation has occurred can only be made once all thread descriptions have been compared. The implementation of `@OnlyRunBy` therefore is more complex: There is one call per `@ThreadDesc`, which stores the thread description in a list, and then another final call to a method that evaluates all stored descriptions.

It is worth noting that, since it is possible that multiple threads evaluate an `@OnlyRunBy` invariant at the same time, there is one list of thread descriptions for each thread. Without this measure, two concurrently executed invariant checks could interfere with each other.

Listings 4.10 and 4.11 show examples of these primitive annotations. It is apparent that they are not very flexible or extensible.

```
@OnlyRunBy(@ThreadDesc(name="main2"))
public class ThreadCheckSample3 {
    @OnlyRunBy({
      @ThreadDesc(name="main"),
      @ThreadDesc(name="main3"),
      @ThreadDesc(eventThread=OnlyRunBy.EVENT_THREAD.ONLY),
      @ThreadDesc(id=3),
      @ThreadDesc(id=4)})
    public void run() {
        System.out.println("run");
    }
    @OnlyRunBy({
      @ThreadDesc(name="main3"),
      @ThreadDesc(group="foo"),
      @ThreadDesc(group="bar"),
      @ThreadDesc(id=3),
      @ThreadDesc(id=4)})
    public static void main(String[] args) {
        System.out.println("main");
        (new ThreadCheckSample3()).run();
    }
}
```

Listing 4.11: Illustration of Primitive `@OnlyRunBy` Annotations

4.4 Predicate Annotations

After implementing the primitive annotations, we noticed that only a small set of the desirable invariants could be expressed, particularly when annotating the model object associated with a GUI component: For instance, it is easy to specify that a `JTable` should only be accessed by the event thread once the component has been realized, but how would a developer do the same for the `TableModel` object that contains the data displayed in the table? The model object does not have a reference to the GUI component.

Several alternatives were considered:

1. Defining the semantics of "event thread only after realized" for non-component classes such that any thread can call this method until the event thread calls it the first time; after that, only the event thread may call the method.

2. Linking model and view classes with a `HashMap<String,Boolean>` by letting the view set the value indexed by some unique name to `true` and defining "event thread only after realized" as allowing any thread to run the method as long as the value indexed by the unique name is `false`; once it is `true`, only the event thread is allowed. This idea could be generalized to allow arbitrary Boolean expressions.

3. Specifying a Java method by name in the annotation and calling it.

4. Allowing arbitrary Java code as a string in the annotation, extracting, compiling and then inserting it in the right place.

The first alternative seemed too limited and the second one too brittle, because it involved assigning unique names. The fourth option seemed very attractive, but it would have involved a major effort to correctly implement parsing and access to classes, methods, fields, and annotation values. We therefore chose the third option, which lead to the development of predicate annotations.

A predicate annotation is an annotation that itself is annotated by one of the two meta-annotations `@PredicateLink` and `@Combine`. These meta-annotations have the dual purpose of marking an annotation as predicate annotation and specifying the way the predicate annotation behaves.

Annotations marked as `@PredicateLink` may only contain primitive data members, strings, class objects, enumerations, and arrays of the types just mentioned; they may not contain other annotations or arrays of annotations as members. Annotations marked with `@Combine`, on the other hand, are the dual and may only contain other annotations or arrays of annotations (in fact, the annotations must even be predicate annotations, i.e. they must have either a `@PredicateLink` or a `@Combine` meta-annotation); they may not contain primitive data members, strings, class objects, enumerations, or arrays of those types.

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.ANNOTATION_TYPE})
public @interface PredicateLink {
    /** Class containing the predicate method.
      * @return class with the predicate method. */
    Class value();
    /** Name of the predicate method. If not
      * specified, the name "check" is assumed.
      * @return name of the predicate method. */
    String method() default "check";
    /** Whether method arguments should be passed
      * to the predicate.
      * @return true if arguments should be passed. */
    boolean arguments() default false;
}
```

Listing 4.12: `@PredicateLink` Meta-Annotation

4.4.1 Predicate Link Annotations

Listing 4.12 shows the definition of the `@PredicateLink` meta-annotation. The `@PredicateLink` meta-annotation establishes a link between the annotation it is marking and a static method returning a `boolean` (the predicate method, specified by a `Class` instance and a method name). This method is called, and if it returns `false`, a violation has occurred. The predicate method must be in a completely static context, which means the method cannot be in an anonymous inner class or a non-static inner class; methods in static inner classes can be used. To be precise, it has to be possible to call the predicate method from the beginning of every method that is affected by the annotation.

The method is specified using the `@PredicateLink.value()` and `@PredicateLink.method()` members. The former specifies the class that contains the predicate method. The latter is optional and specifies the method name; if no method name is specified, `"check"` is assumed. The third member, `@PredicateLink.arguments()` decides whether the method arguments should be passed.

The predicate method is required to have a very particular signature: It has to return `boolean` and have at least one parameter of type `Object`. That first `Object` parameter contains the value of `this` if the method whose invariant is checked is non-static, or `null` if the method is static.

Additionally, if `@PredicateLink.arguments()` is `true`, the predicate method must accept an `Object[]` array as second parameter which contains the argument values of the annotated method. If the method has no arguments, the length of the array will be 0. Primitive parameters like `int` or `boolean` will automatically be converted to their respective boxed versions, such as `Integer` and `Boolean`.

Finally, the method has to have one parameter for each member of the annotation that the `@PredicateLink` meta-annotation is marking, in the same order as they appear in the annotation, with the same types and the same names. For example, if the annotation has three members (`int m1; float m2; String m3;`) and `@PredicateLink.arguments()` is `false`, the method has to accept four parameters (`Object this0, int m1, float m2, String m3`); if `@PredicateLink.arguments()` is `true`, the method needs to have five parameters (`Object this0, Object[] args, int m1, float m2, String m3`).

When an invariant check is performed at the beginning of a method, the bytecode is rewritten to first load `this` or `null` on the stack, depending on whether the method is non-static or static. Then, if `@PredicateLink.arguments()` is `true`, an array of the method arguments is built and put on the stack; the left-most argument is at array index 0. Finally, the values of the annotation members are read out and put on the stack and the specified predicate method is called. Figure 4.2 shows the stack layout.

The predicate method returns a `boolean`, and if it is `false`, a violation has occurred, which is added to a log. Listings 4.13 to 4.19 show examples of how annotations, predicate methods, and methods to which annotations are applied fit together.

(top of stack)	
annotation member value n	values from the annotation members
⋮	
annotation member value 2	
annotation member value 1	
method argument array	optional, only if `@PredicateLink.arguments()` is `true`
`this` or `null`	for non-static or static methods

Figure 4.2 : Stack for Calling `@PredicateLink` Annotation Predicate

```
@PredicateLink(value=Predicates.class)                    1
public @interface OnlyEventThread { }                     2
class Predicates {                                        3
  public static boolean check(Object thisObject) {        4
    // this0 is the bar method's this                     5
    return EventQueue.isDispatchThread();                 6
  }                                                       7
}                                                         8
class Foo {                                               9
  @OnlyEventThread public void bar() { }                  10
}                                                         11
```

Listing 4.13: Memberless Annotation, 0 Parameters, Arguments not Passed

In listing 4.13, the annotation `OnlyEventThread` does not have any members, and the method, `Foo.bar` does not have any parameters either. Therefore, the predicate method `Predicates.check` (`check` because no other name was specified) has only one parameter, `this0`, which is the `bar` method's `this`, since the method is non-static. The annotation `OnlyEventThread` can be used to mark methods that must be executed in the event thread.

Predicate annotations consist of three individual parts: The annotation definition, the predicate method, and the actual usage site of the annotation. While listings 4.13,

```
@PredicateLink(value=Predicates.class)
public @interface OnlyEventThread { }
```

Listing 4.14: Annotation Definition of Listing 4.13

```
class Predicates {
  public static boolean check(Object thisObject) {
    // this0 is the bar method's this
    return EventQueue.isDispatchThread();
  }
}
```

Listing 4.15: Predicate Method of Listing 4.13

and 4.17 to 4.19 show all three parts together, the reader should keep in mind that the annotation definition and the predicate method are usually written by the library developer, and the application developer only needs to work with the annotation usage.

To show the simplicity of this scheme, the individual parts from listing 4.13 have been separated and are shown in listings 4.14, 4.15 and 4.16.

Listing 4.17 has an annotation `OnlyThreadWithName` with two members, `String value` and `boolean regex`, and the method `Foo.bar` also has a parameter. However, since arguments are not passed, the `Foo.bar`'s `x` argument is not passed. The predicate method `Predicates.checkName` has three parameters, `this0` (which is `null` this time, since the `bar` method is static), `String value` and `boolean regex` from the annotation. The `OnlyThreadWithName` annotation in listing 4.17 is used to restrict execution of the `Foo.bar` method to threads with then name “main”.

```
class Foo {
  @OnlyEventThread public void bar() { }
}
```

Listing 4.16: Usage Site of Listing 4.13

```
@PredicateLink(value=Predicates.class, method="checkName")        1
public @interface OnlyThreadWithName {                            2
  String value(); boolean regex() default false;                  3
}                                                                 4
class Predicates {                                                5
  public static boolean checkName(Object this0,                   6
    String value, boolean regex) {                                7
    // this0==null                                                8
    // value, regex contain the values in the annotation          9
    final String name = Thread.currentThread().getName();         10
    return regex?(name.matches(value)):(name.equals(value));      11
  }                                                               12
}                                                                 13
class Foo {                                                       14
  @OnlyThreadWithName(value="main")                               15
  public static void bar(int x) { }                               16
}                                                                 17
```

Listing 4.17: Annotation with Members, 1 Parameter, Arguments not Passed

Listing 4.18 has an annotation `NoNullArguments` without members and a method with two parameters. Since argument passing has been enabled, the predicate method `Predicates.checkNoneNull` has two parameters, `this0` (which is `bar`'s `this`, since the method is non-static) and an two-element array `args`, containing the values of the arguments passed when `bar` is called. The annotation `NoNullArguments` in this example reports a violation if `bar` is called and at least one arguments is `null`.

In listing 4.19 an annotation with a member, `NotNullArgument`, is applied to a method with two parameters, `Foo.bar`. The predicate method, `Predicates.check NotNullArg`, therefore has three arguments: `this0` (which is the value of `this` in the `bar` method), an array `args` with two elements (the values of `dummy0` and `dummy1`), and an `int` with the value of `i` from the annotation. The annotation `NotNullArgument` in this example is used to restrict the parameter with index 0 (`dummy0`) to non-zero values.

Since the predicate methods are regular static Java methods, developers can perform any operation allowed in any static Java method: access static fields, call static methods, and, using the first parameter, even access `this` if the call came from a

```
@PredicateLink(value=Predicates.class, method="checkNoneNull",
               arguments=true)
public @interface NoNullArguments { }
class Predicates {
  public static boolean checkNoneNull(Object this0,
    Object[] args) {
    // this0 is bar's this, args.length()==2
    for(int i=0; args<args.length; ++i)
      if (args[i]==null) return false;
    return true;
  }
}
class Foo {
  @NoNullArguments
  public void bar(String dummy0, Object dummy1) { }
}
```

Listing 4.18: Memberless Annotation, 2 Parameters, Arguments Passed

```
@PredicateLink(value=Predicates.class,
  method="checkNotNullArg", arguments=true)
public @interface NotNullArgument { int i(); }
class Predicates {
  public static boolean checkNotNullArg(Object this0,
    Object[] args, int i) {
    // this0 is bar's this, args.length()==2
    // i is from the annotation
    if (i>=args.length) return false;
    return args[i]!=null;
  }
}
class Foo {
  @NotNullArgument(i=0)
  public void bar(String dummy0, Object dummy1) { }
}
```

Listing 4.19: Annotation with Member, 2 Parameters, Arguments Passed

```
@PredicateLink(value=Foo.Predicates.class)
@interface AccessPrivate { }
class Foo {
  // static inner class
  public static class Predicates {
    public boolean check(Object this0) {
      if (this0==null) return false;
      return (((Foo)this0)._privateField==0)
    }
  }
  private int _privateField = 1;
  @AccessPrivate public void bar() { }
}
```

Listing 4.20: Predicate Accessing Private Field Using Nested Class

non-static method. If `@PredicateLink.arguments()` is `true`, then the predicate method also receives the values of the arguments that were used to call the annotated method. Private fields are not directly accessible, unless the predicate is contained in an inner class, as shown in listing 4.20. Reflection can also be used to access private fields of any object, but the predicate method may have to call `AccessibleObject.setAccessible(true)` first, as demonstrated in listing 4.21.

Predicate methods and reflection can be a powerful diagnostic tool, but with power also comes responsibility. The predicate methods should be kept as short and simple as possible: Since any Java code can be executed, a predicate annotation could easily introduce an infinite loop or a deadlock into an otherwise unproblematic program. If a predicate method throws an uncaught exception, this is recorded as incorrectly written predicate. Exceptions are not allowed to escape from a predicate method, as that would alter the behavior of the program being tested.

The library that is part of the invariant checker contains a large number of predefined annotations. These are discussed in appendix A.

```
@PredicateLink(value=Predicates.class)
@interface AccessPrivateReflect { }
// not an inner class
class Predicates {
  public boolean check(Object this0) {
    if (this0==null) return false;
    Field field = this0.getClass().
      getDeclaredField("_privateField");
    boolean isAccessible = field.isAccessible();
    field.setAccessible(true); // access private
    Object value = field.get(thisObject);
    field.setAccessible(isAccessible); // reset access
    return value.equals(0);
  }
}
class Foo {
  private int _privateField = 1;
  @AccessPrivateReflect public void bar() { }
}
```

Listing 4.21: Predicate Accessing Private Field Using Reflection

4.4.2 Combine Annotations

`@Combine` meta-annotations were created because we longed for the ability to perform Boolean operations on predicate annotations to create larger, more complex compound annotations. The example shown in 4.22 would have been the ideal usage, but as discussed in section C.3 in the appendix, this was not possible with Java's primitive annotation system. The `@Combine`-style annotations presented in this section are the best solution we could find.

```
@And({
  @Or({
    @NotThreadWithName("foo"),
    @NotThreadWithName("bar")}),
  @Not(@NotThreadWithGroupName("main")),
  @NotThreadWithID(5)})
void someMethod() { /* ... */ }
```

Listing 4.22: Ideal, But Unachievable Usage Example

The definition of the `@Combine` meta-annotation can be found in listing 4.23. The meta-annotation has two members, `@Combine.value()` and `@Combine.arguments()`.

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.ANNOTATION_TYPE})
public @interface Combine {
    public static enum Mode {
        AND, OR, XOR, NOT, IMPLIES;
    }
    /** Operation used to combine the member predicate
      * annotations, i.e. AND, OR, XOR, NOT, IMPLIES.
      * @return combining operation */
    Mode value() default Mode.OR;
    /** Whether method arguments should be passed
      * to the predicate.
      * @return true if arguments should be passed. */
    boolean arguments() default false;
}
```

Listing 4.23: `@Combine` Meta-Annotation

The former decides if the member annotations of the annotation marked by the `@Combine` meta-annotation should be merged using “and”, “or”, “xor”, “not”, or “implies”. The second member, `@Combine.arguments()`, determines whether the method arguments should be passed to the predicate methods, just like `@PredicateLink.arguments()` did.

Since `@Combine`-style annotations are used to create compounds of other predicate annotations, the only members allowed in a `@Combine`-style annotation are other predicate annotations or arrays of predicate annotations; primitive data, strings, class objects, enumerations, or arrays of the aforementioned types are not allowed. The member annotations or the elements in the member array must also have been annotated with one of the meta-annotations `@PredicateLink` or `@Combine`. There is a further restriction: If a `@Combine`-style annotation has `arguments()` set to `false`, then none of the member annotations may have `arguments()` set to `true`, because that information simply is not available to the parent annotation's predicate method. If, however, `@Combine.arguments()` is `true`, then the member annotations are free to either have the member arguments be passed to their predicate methods or to ignore them.

At the usage site, there is no difference between using a `@PredicateLink`- style or a `@Combine`-style annotation: For both kinds of annotations, `this` or `null` is first put on the stack, depending on whether the method is non-static or static. Then, if `arguments()` is `true`, an array is created and the arguments are stored in the array. As third step, the annotation's member values are put on the stack. Finally, the annotation's predicate method is called.

A `@Combine`-style annotation, however, does not have a designated predicate method, like the one specified by a `@PredicateLink` meta-annotation. A `@Combine`-style annotation may have members that have predicate annotations, though, but even that is not guaranteed: A `@Combine`-style annotation may combine several other `@Combine`-style annotations. The annotations generate a call graph that is tree-shaped, with `@Combine`-style annotations as interior nodes and `@PredicateLink`-style annotations as leaves. Arrays of annotations are treated specially: They are not treated as a single entity; instead, each array element becomes a child annotation of the annotation that contains the array. Since programs are finite, these call trees are finite too, and eventually there will be predicate methods at the leaves that can be called.

When the invariant checker encounters a usage site of a `@Combine`-style annotation, it automatically generates a predicate method in a reserved package, by default `edu.rice.cs.cunit.threadCheck.predicates`. In doing so, it recursively explores the call tree in a depth-first fashion and automatically generates predicate methods for member annotations that are also `@Combine`-style annotations. Once it reaches the leaves, it accumulates the member values of the `@PredicateLink`-style annotation. When the tree has been fully explored, all predicate methods for `@Combine`-style children have been generated, and a list of the member values of the `@PredicateLink`-style annotations at the leaves has been built. In essence, the tree has been flattened, and all of the member values present in the leaves are available to the root.

This flattening has the negative side effect that a predicate method for a parent `@Combine`-style annotation has to accept all the arguments for all its children. If, for example, an annotation `@Parent` is a `@Combine`-style annotation with five members, which are all `@PredicateLink` annotations with two members, then the predicate method for `@Parent` must have ten parameters, and the parameter for `this` or `null`, and perhaps the `Object[]` array for method arguments.

The flattening of all member values made it easy to integrate `@Combine`-style annotations with `@PredicateLink` annotations. We considered two alternatives: instantiating instances of the annotations, which basically means creating anonymous inner classes on-the-fly; and creating wrapper classes that contain the same information as the annotations but are placed in a completely separate class hierarchy and package. Both approaches seemed to involve more work and particularly higher memory usage than the flattening solution.

The automatically generated predicate method first loads a start value into a `boolean` accumulator: `true` for "and", `false` for "or" and "implies", and `0` for "xor". It then prepares the stack for calling the first member predicate method, in the same way `@PredicateLink`-style predicate methods were called: It puts `this` or `null` on the stack, depending on whether the method that was annotated with this `@Combine`-style annotation was non-static or static. If `@Combine.arguments()` is `true`, then an array with the method arguments is created. Then all the member values for the child annotation are put on the stack, and finally the predicate method for the child annotation is called.

When the call returns, the predicate's `boolean` return value is processed. If the mode is "not", the return value is negated and returned. If the mode is "and" or "or", the return value is merged with the accumulator using the appropriate Boolean operation. If the mode is "xor", then the number of times `true` is returned is counted, and `true` is returned if and only if that count is one. For the "implies" mode, the Boolean equivalence $a \rightarrow b = (\neg a) \vee b$ is used: The first return value is negated

```
@Combine(value=Combine.Mode.AND)                              1
public @interface CombineWithDefault {                        2
  NotThreadWithName name()                                    3
    default @NotThreadWithName("foo");                        4
  NotThreadWithGroupName group()                              5
    default @NotThreadWithGroupName("bar");                   6
}                                                             7
@CombineWithDefault                                           8
void someMethod() { /* ... */ }                               9
@CombineWithDefault(name=@NotThreadWithName("fee"))           10
void someMethod() { /* ... */ }                               11
```

Listing 4.24: A `@Combine` Annotation With Default Values

and the disjunction with the second return value is returned. Since "implies" is not commutative, but Java does not specify the order in which annotation members appear in a class file, "implies" requires a two-element array of annotations. The array imposes an order for the operands.

When the predicate methods of all child annotations have been called, the value of the accumulator is returned and represents the success or failure of the `@Combine`-style annotation.

The use of default values for members is crucial in achieving ease of use: If all members of a `@Combine`-style annotation have default values, then the annotation can be used as a parameterless annotation. In listing 4.24, the annotations `@NotThreadWithName` and `@NotThreadWithGroupName` are combined to form the `@CombineWithDefault` annotation. The members of `@CombineWithDefault` are all given default values, so when `@CombineWithDefault` is actually used, it can be done without specifying any arguments, as in line 8, or with arguments, overriding the default values, as in line 10.

The automatic generation of predicate methods for `@Combine` annotations is quite complex, right now requires disk write access, and due to the recursive nature can be a bit time-consuming. Fortunately, the predicate methods can be cached, so they only need to be generated once for each annotation usage. However, since a

```
@Combine(value=Combine.Mode.AND, arguments=true)        1
public @interface TestCombine {                         2
  NotThreadWithName[] value();                          3
}                                                       4
@TestCombine({@NotThreadWithName("foo")})               5
void someMethod() { /* ... */                           6
@TestCombine({                                          7
  @NotThreadWithName("foo"),                            8
  @NotThreadWithName("bar")})                           9
void otherMethod() { /* ... */ }                        10
```

Listing 4.25: A `@Combine` Annotation Used Twice

`@Combine`-style annotation may be used in more than one way it may be necessary to generate more than one predicate method. Listing 4.25 illustrates this problem: The first time `@TestCombine` is used, the array contains only one `@NotThreadWithName` annotation; the second time `@TestCombine` is used, there are two array entries. Since the first use in line 5 needs to make only one call to a child predicate, but the second use in line 7 needs to make two, each use requires its own predicate method. To distinguish the different predicate methods, the names of the automatically generated predicate methods contain the encoded sizes of the arrays. The predicate method name for the first use in line 5 would be `check$value$$$1` and the name for the second use in line 7 `check$value$$$2`. In case of arrays nested inside of arrays, a path-like string using one or more `'$'` as separator encodes the nesting. For example, `check$value$$$1$value$$0$nested$$$4` indicates that `nested` is an array with four elements, and it is stored in `value[0]`, and the `value` array has only one element. Every time the array sizes, the numbers following the triple `'$'`, or the indices of nested arrays, the numbers following the double `'$'`, change, a new predicate method has to be generated. At the usage site, the invariant checker analyzes the annotation, generates the encoded method name, and then either generates that predicate method or re-uses a cached, already existing one.

4.4.3 Predicate Annotations Using Reflection

We have also written an instrumentation strategy for invariant checking that relies less on bytecode rewriting and more on reflection. Every time a predicate annotation is encountered, a call is made to a library method, passing just the class, the method name and descriptor, and possibly the method arguments. There is no need to recursively generate predicate methods for `@Combine` annotations and to flatten all the values of the annotations, since that is done using reflection in the library method.

Our framework locates the `Method` object for the method that was entered and processes all `@PredicateLink` and `@Combine` annotations that should affect the method, using the rules specified in figure 4.1. For a `@PredicateLink`, the reflection-based invariant checker requests the `Method` instance for the predicate method and then calls `Method.invoke` with the appropriate arguments. For a `@Combine` annotation, the member annotations are processed recursively.

Using the reflection-based invariant checker instead of the one that performs extensive bytecode rewriting has two advantages:

- Since no predicate methods need to be generated, `@Combine`-style annotations using reflection have less initial overhead.
- The annotation is an actual object retrieved using reflection and can therefore be printed out more easily, giving the developer more information about why the violation occurred.

Reflection has downsides, too. Line number information is harder to retrieve, and since reflection operates at runtime, the annotations need to be available at runtime as well. That means that their retention policy has to be runtime, while the invariant checker that uses bytecode rewriting only needs class retention. If the annotated program uses reflection itself, then a runtime retention policy could interfere with the program's desired behavior. Furthermore, the predicate methods generated by bytecode rewriting alone are typically small and may be inlined by a JIT compiler.

4.5 External Annotations in XML Format

The information that can be supplied to the invariant checker in the form of predicate or primitive annotations, as described in the previous two sections, requires the modification of source code: To specify a concurrency invariant, a class or method has to be prefixed by a Java 5.0 annotation.

The software developed as part of this thesis has the ability to convert source code-based annotations into a separate, stand-alone XML file expressing the same information. The XML file can be imported back again, resulting in instrumented bytecode that checks the same concurrency invariants without having had to modify source code.

The structure of the XML file is simple enough to be written by hand and can use two different formats, one sorted by Java packages and classes, the other sorted by invariant. Listings 4.26 and 4.27 show examples of XML files expressing annotation information in both the class- and invariant-based formats. Tools to convert from one format to another are also available.

Internally, the XML annotations are converted to the class-based format because it corresponds more closely with the behavior of the instrumentation strategy that rewrites the class files: It moves from package to package, class to class, method to method. The invariant-based format, though, is typically easier to create by hand, and it is used in merging and minimizing sets of invariants.

By using individual XML files that are not directly connected to source code, programmers can annotate and add invariants to programs and class libraries even if the source code is not available or cannot be recompiled.

As part of the Concutest project, we have started an online community website [39] that contains information about the concurrency invariants of the Standard Java API and other commonly used libraries, sorted both by class and by invariant. The content is accessible to anyone, and any programmer can register, add content, and participate

```
<?xml version="1.0" encoding="UTF-8"?>
<concutest><threadcheck>
  <javax><swing><text>
    <DefaultCaret>
      <class>
        <method sig="repaintNewCaret()V">
          <eventThread type="only"/>
        </method>
      </class>
    </DefaultCaret>
    <JTextComponent>
      <class>
        <method sig="setText(Ljava/lang/String;)V">
          <name type="only" value=".*"/>
        </method>
      </class>
    </JTextComponent>
  </text></swing></javax>
</threadcheck></concutest>
```

Listing 4.26: XML File in Class-Based Format

```
<?xml version="1.0" encoding="UTF-8"?>
<concutest>
  <threadcheck:def>
    <invariant>
      <name type="only" value=".*"/>
    </invariant>
    <method name="javax.swing.text.JTextComponent"
            sig="setText(Ljava/lang/String;)V"/>
  </threadcheck:def>
  <threadcheck:def>
    <invariant>
      <eventThread type="only"/>
    </invariant>
    <method name="javax.swing.text.DefaultCaret"
            sig="repaintNewCaret()V"/>
  </threadcheck:def>
</concutest>
```

Listing 4.27: XML File in Invariant-Based Format

in the discussion. We plan to survey the information that is entered by community volunteers and add it to freely available XML files.

The format of the XML files is discussed in appendix B.

4.6 Comparison to Assertions

The annotations introduced in this chapter are very similar to assertions already found in Java since version 1.4. They correspond to assertions placed at the top of a method.

Assertions, however, are written for one specific purpose: To ensure that a certain property is met at one, and only one, place in the program code. Assertions are not inherited by overriding methods or attached to methods first introduced in annotated classes. Assertions are therefore very suitable for performing sanity checks of relatively static code, but they are unsuitable for ensuring that concurrency invariants of extensible frameworks are maintained.

In listing 4.28, the assertion in line 3 successfully protects the application from deadlocking, as it did in listing 4.30, but the overridden method in line 8 does not execute the assertion and is therefore vulnerable to the same deadlock as the code in listing 4.30.

The annotation shown in listing 4.32 is passed on to the overridden method and will detect the problem. Compared to assertions, the annotations introduced in this chapter are therefore more suitable in extensible libraries like those found in modern GUI frameworks.

Another product, Contract4J [40] provides similar capabilities to specify preconditions and postconditions. However, Contract4J requires an aspect-oriented version of Java, which makes the insertion of invariant checks at certain points easier. AspectJ [41], an aspect-oriented version of Java, is not used very frequently, though. The expressiveness of the invariants in this chapter already surpasses that of Contract4J, and an extension to postconditions or data verification is possible.

```
public class MalignSubClassAssert extends Benign {              1
  public void run(Object o1, Object o2) {                       2
    assert(o1!=o2);                                             3
    // ...                                                      4
  }                                                             5
  public static class MalignSubClassNoAssert                    6
    extends MalignSubClassAssert {                              7
    public void run(Object o1, Object o2) { /* ... */ }         8
  }                                                             9
  private static String s = "foo";                              10
  public static void main(String[] args) {                      11
    Benign b = new MalignSubClassAssert(); b.run(s, s);         12
    b = new MalignSubClassNoAssert(); b.run(s, s);              13
  }                                                             14
}                                                               15
```

Listing 4.28: Listing 4.30 with an Assertion

4.7 Subtyping Relation for Annotations

In many cases, the invariant checker is able to statically determine problems in a program's source code by detecting subtyping violations. As long as adding an annotation to a method makes the invariants stricter, then annotations on methods define a subtyping relation similar to the one defined by Java.

With `@NotRunBy` annotations, it is quite clear that each annotation adds an invariant (or duplicates one already present), thus making the invariants for a method in a subclass stricter (or keeping it at least as strict). `@OnlyRunBy`, however, are implemented to represent alternatives of which only one has to be `true`: A method may be executed only by threads named `"foo"` or `"bar"`, for example. Adding an `@OnlyRunBy` annotation to a method in a subclass therefore makes the invariants weaker, so for the purpose of statically detecting potential concurrency problems, they need to be excluded.

For predicate annotations, it is not as obvious that each annotation makes the invariant stricter, but it should be clear that adding another required invariant, even if it is universally true, cannot make the overall invariant weaker; at worst, the re-

Class	`A`	`B extends A`
New invariant	(none)	not event thread
Combined invariants	(none)	not event thread
Effective invariants	(none)	not event thread

Table 4.1 : Sample Invariants

quirements will remain just as strict as without the added predicate annotation. Therefore, it is reasonable to assume that the subtyping described below applies to both `@NotRunBy` annotations and predicate annotations.

Consider the following two classes `A` and `B`, with `class B extends A`, and the invariants shown in table 4.1. In this table, "new invariant" refers to the invariant added for a class, "combined invariant" shows the new invariant combined with the invariant inherited from the superclass, and the "effective invariant" is a simplified restatement of the combined invariant.

Remember that the invariants from the superclass are passed on to the subclass. It is clear that class `A` is less restrictive than class `B`: The methods in `A` may be run by any thread, but those in `B` may be run by all those except the event thread. The invariant for method execution have been made stricter for class `B`.

We now use the lambda calculus to illustrate and formalize the behavior of the invariant checker. We pretend that the set of invariants restricting the execution of a method is a special input parameter passed to every function. In the extended typed lambda calculus described in sections 11.7 and 15.2 in Pierce's "Types and Programming Languages" [42], a record storing unit can be used. The subtyping rules for records tell us that the wider record is a subtype of the narrower record, and

subtyping for function types suggests that there should be contravariant behavior for function parameters. Since the set of invariants I_A that constrains methods in class `A` contains fewer elements (none, in fact) than the set of invariants I_B that constrains class `B`, I_B is the subtype of I_A: $I_B <: I_A$.

Let $<@$ be the invariant subtyping relation to distinguish it from the $<:$ symbol that is used for classical subtyping. When we consider the contravariant behavior of the special function parameter containing the set of invariants, it follows that class `A` is the invariant subtype of class `B`: `A` $<@$ `B`. It becomes clear that this runs counter to the direction of subtyping set up by Java with `class B extends A`, `B<:A`.

A second, larger example uses the classes `W`, `X`, `Y` and `Z`, with `class X extends W`, `class Y extends X`, and `class Z extends Y`, and the invariants in table 4.2.

Class	`W`	`X extends W`	`Y extends X`	`Z extends Y`
New invariant	(none)	not event thread	not `"foo"`	not regex `".*"`
Combined invariants	(none)	not event thread	not event thread not `"foo"`	not event thread not `"foo"` not regex `".*"`
Effective invariants	(none)	not event thread	not event thread not `"foo"`	no thread fits

Table 4.2 : More Sample Invariants

Again, the invariants acquired by a superclass get passed on to its subclasses, changing

the effective invariant: While class `W` may be executed by any thread, class `X` may be executed by all threads except the event thread, and class `Y` narrows it further down to threads not named `"foo"`. When class `Z` adds the invariant that it may not be executed by any thread whose name matches the regular expression `".*"`, there are no more threads left that satisfy all these invariants.

When considering the sets of invariants that constrain the execution of the methods of these classes, then the set of invariants I_W of class `W` is empty and therefore the smallest, the set I_X of class `X` contains one element, I_Y two elements, and set I_Z is the largest set, containing three elements. The subtyping of records indicates that $I_Z <: I_Y <: I_X <: I_W$. From that and the contravariant relationship of function parameters, we can establish the invariant subtyping relationship `W` <@ `X` <@ `Y` <@ `Z`. Again, the invariant subtyping runs exactly in the opposite direction as the subtyping established by Java.

The invariant checker can use these observations to statically determine that the programmer is likely extending an annotated class library in a way that does not maintain substitutability. To make this example concrete: Assume that `JComponent`'s methods may be executed by any thread; they do not have any concurrency requirements. The methods in `JTextField`, which is a subclass of `JComponent`, may only be accessed in the event thread. If programmers work abstractly at the level of `JComponent`, but are actually dealing with an instance of `JTextField`, they may not be aware of the invariants of that class. Furthermore, a second subclass, `JTable`, could have a completely contradictory set of invariants. Trying to deal with both of them virtually forces the programmer to deal with a class like `Z` above: The invariants of the classes just cannot be met.

By analyzing the Java subtyping hierarchy and the hierarchy created by invariant annotations, the invariant checker can statically determine these situations and direct the attention of the programmer to areas of the program or class library that will likely be problematic under concurrent situations.

4.7.1 Extended Syntax

The syntax and subtyping relation introduced here is based on Featherweight Java [43].

$$
\begin{array}{lll}
PD & ::= & \text{Predicate declarations} \\
 & & \texttt{@PredicateLink(}value = C, method = "m", \\
 & & \quad arguments = \{\texttt{true}|\texttt{false}\})\ \texttt{@interface P}\ \{\ \overline{C\ m();}\ \} \\
 & & \texttt{@Combine(}value = \{\texttt{AND}|\texttt{OR}|\texttt{XOR}|\texttt{NOT}|\texttt{IMPLIES}\}, \\
 & & \quad arguments = \{\texttt{true}|\texttt{false}\})\ \texttt{@interface P}\ \{\ \overline{P\ m();}\ \} \\
A & ::= & \texttt{@}P\texttt{(}\overline{x}\texttt{)} \qquad \text{Invariant Checker annotations} \\
 & & \texttt{@NotRunBy(}\overline{x}\texttt{)} \\
 & & \texttt{@OnlyRunBy(}\overline{x}\texttt{)} \\
CL & ::= & \texttt{class}\ \overline{A}\ C\ \texttt{extends}\ C\ \{\overline{C\ f};\ K\ \overline{M}\} \qquad \text{class declarations} \\
K & ::= & \overline{A}\ C(\overline{C\ f})\ \{\ \texttt{super(}\overline{f};\texttt{)}\ \texttt{this.}\overline{f} = \overline{f};\} \qquad \text{constructor declarations} \\
M & ::= & \overline{A}\ C\ m(\overline{C\ x})\ \{\ \texttt{return}\ t;\} \qquad \text{method declarations}
\end{array}
$$

Figure 4.3 : Invariant Checker-Extended Featherweight Java (TCFJ) Syntax

The rest of the syntax is identical to that of Featherweight Java.

4.7.2 Featherweight Java Subtyping Relation

Reflexive property

$$C <: C$$

Transitive property

$$\frac{C <: D \qquad D <: E}{C <: E}$$

$$\frac{CT(C) = \overline{A}\ \texttt{class}\ C\ \texttt{extends}\ D\{...\}}{C <: D}$$

Figure 4.4 : Featherweight Java Subtyping Relation

4.7.3 Auxiliary Definitions

Extraction

The following rules allow the extraction of methods from a class, annotations from a class or method, and the name of a method:

$$\frac{CT(C) = \overline{A}\ \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{C_f\ f};\ K\ \overline{M}\}}{extract_M(C) = \overline{M}}$$

Figure 4.5 : TCFJ - Extract the Method Declarations from a Class

$$\frac{CT(C) = \overline{A}\ \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{C_f\ f};\ K\ \overline{M}\}}{extract_A(C) = \overline{A}}$$

Figure 4.6 : TCFJ - Extract the Annotations from a Class

$$\frac{\overline{A}\ R\ m(\overline{P\ x}) \in extract_M(C)}{extract_A(m, C) = \overline{A}}$$

Figure 4.7 : TCFJ - Extract the Annotations from a Method

$$\frac{M = \overline{A}\ R\ m(\overline{P\ x})}{name_M(M) = m}$$

Figure 4.8 : TCFJ - Extract the Name from a Method Declaration

Get only the $\texttt{@}P(\overline{x})$ predicate annotations, the $\texttt{@NotRunBy(}\overline{x}\texttt{)}$ or $\texttt{@OnlyRunBy(}\overline{x}\texttt{)}$ annotations of an annotations list, respectively:

$$\begin{array}{lcl}
predAnnot(\cdot) & = & \cdot \\
predAnnot(\texttt{@}P\texttt{(}\overline{x}\texttt{)}, \overline{A}) & = & \texttt{@}P(\overline{x}), predAnnot(\overline{A}) \\
predAnnot(\texttt{@NotRunBy(}\overline{x}\texttt{)}, \overline{A}) & = & predAnnot(\overline{A}) \\
predAnnot(\texttt{@OnlyRunBy(}\overline{x}\texttt{)}, \overline{A}) & = & predAnnot(\overline{A})
\end{array}$$

Figure 4.9 : TCFJ - Extract Only Predicate Annotations

$$\begin{array}{lcl}
notRunBy(\cdot) & = & \cdot \\
notRunBy(\texttt{@NotRunBy(}\overline{x}\texttt{)}, \overline{A}) & = & \texttt{@NotRunBy(}\overline{x}\texttt{)}, notRunBy(\overline{A}) \\
notRunBy(\texttt{@OnlyRunBy(}\overline{x}\texttt{)}, \overline{A}) & = & notRunBy(\overline{A}) \\
notRunBy(\texttt{@}P\texttt{(}\overline{\overline{x}}\texttt{)}, \overline{A}) & = & notRunBy(\overline{A})
\end{array}$$

Figure 4.10 : TCFJ - Extract Only `@NotRunBy` Annotations

$$\begin{array}{lcl}
onlyRunBy(\cdot) & = & \cdot \\
onlyRunBy(\texttt{@OnlyRunBy(}\overline{x}\texttt{)}, \overline{A}) & = & \texttt{@OnlyRunBy(}\overline{x}\texttt{)}, onlyRunBy(\overline{A}) \\
onlyRunBy(\texttt{@NotRunBy(}\overline{x}\texttt{)}, \overline{A}) & = & onlyRunBy(\overline{A}) \\
onlyRunBy(\texttt{@}P\texttt{(}\overline{x}\texttt{)}, \overline{A}) & = & onlyRunBy(\overline{A})
\end{array}$$

Figure 4.11 : TCFJ - Extract Only `@OnlyRunBy` Annotations

Furthermore, let $subAnnot(A) = predAnnot(A) \cup notRunBy(A)$ be the set of annotations to which the new subtyping relation applies.

Method Introduction

Find the class D, such that $C <: D$, that introduces a method m, i.e. the superclass of C containing a declaration of m that does not have any superclasses that also contains a declaration of m. If the method m is not defined in C or any of its superclasses, $\cdot$ is returned.

$$introduced(m, \texttt{Object}) = \cdot$$

$$\frac{CT(C) = \overline{A}\ \texttt{class}\ C\ \texttt{extends}\ D\ \{...\} \qquad m \text{ defined in } extract_M(C) \qquad introduced(m, D) = \cdot}{introduced(m, C) = C}$$

$$\frac{CT(C) = \overline{A}\ \texttt{class}\ C\ \texttt{extends}\ D\ \{...\} \qquad introduced(m, D) = S \qquad S \neq \cdot}{introduced(m, C) = S}$$

Figure 4.12 : TCFJ - Method Introduction

Annotation Lookup

Methods of class `Object` do not have any annotations.

$$annot_M(m, \texttt{Object}) = \cdot$$

Figure 4.13 : TCFJ - Annotations Lookup for `Object`

The annotations of a method m that is introduced in class C are the union of the class annotations on C and the annotations mentioned in the method declaration of m in C.

$$\frac{\begin{array}{c} introduced(m, C) = C \\ extract_A(C) = \overline{A_C} \\ extract_A(m, C) = \overline{A_M} \end{array}}{annot_M(m, C) = \overline{A_C} \;\cup\; \overline{A_M}}$$

Figure 4.14 : TCFJ - Annotation Lookup for Introduction

The annotations of a method m that is overridden in class C, i.e. method m was introduced in a superclass of C, are the union of the class annotations on C, the annotations mentioned in the overriding method declaration of m in C, and the annotations of the same method m in the superclass of C.

$$\frac{\begin{array}{c}\neg introduced(m, C) \\ CT(C) = \overline{A_C}\ \texttt{class}\ C\ \texttt{extends}\ D\ \{...\} \\ extract_A(m, C) = \overline{A_M} \\ annot_M(m, D) = \overline{A_D}\end{array}}{annot_M(m, C) = \overline{A_C}\ \cup\ \overline{A_M}\ \cup\ \overline{A_D}}$$

Figure 4.15 : TCFJ - Annotation Lookup for Overriding

Check for Annotation Contradictions

Check that a set of annotations does not contain contradictions.

$\boxed{\overline{A}\ \texttt{OK}}$

Figure 4.16 : TCFJ - Check for Annotation Contradictions

$$\frac{}{\cdot\ \texttt{OK}} \qquad \frac{\overline{A}\ \texttt{OK}}{\texttt{@}P\texttt{(}\overline{x}\texttt{)},\ \overline{A}\ \texttt{OK}} \qquad \frac{\begin{array}{c}\overline{A}\ \texttt{OK} \\ \texttt{@OnlyRunBy(}\overline{x}\texttt{)} \notin \overline{A}\end{array}}{\texttt{@NotRunBy(}\overline{x}\texttt{)},\ \overline{A}\ \texttt{OK}} \qquad \frac{\begin{array}{c}\overline{A}\ \texttt{OK} \\ \texttt{@NotRunBy(}\overline{x}\texttt{)} \notin \overline{A}\end{array}}{\texttt{@OnlyRunBy(}\overline{x}\texttt{)},\ \overline{A}\ \texttt{OK}}$$

These rules check that a set of annotations does not contain any obvious contradictions, i.e. an annotation that specifies a method must be run by a thread with name `"foo"`, but at the same time may not be run by a thread named `"foo"`. Since predicate annotations are distinct from primitive annotations and may execute arbitrary Java methods, the invariant checker cannot statically check for contradictions, so predicate annotations of the form $\texttt{@}P\texttt{(}\overline{x}\texttt{)}$ are assumed to be noncontradictory.

4.7.4 Combined Subtyping Relation

Let $<:@$ be the combination of the Java subtyping relation and the invariant subtyping relation $<@$:

Reflexive property

$$C <:@ C$$

Transitive property

$$\frac{C <:@ D \qquad D <:@ E}{C <:@ E}$$

$$\frac{\begin{array}{c} C <: D \\ extract_M(D) = \overline{M} \\ name_M(\overline{M}) = \overline{m} \\ subAnnot(extract_A(\overline{m}, C)) = \overline{\overline{A_C}} \\ subAnnot(extract_A(\overline{m}, D)) = \overline{\overline{A_D}} \\ \text{for each}(\overline{A_C}, \overline{A_D}) \in (\overline{\overline{A_C}}, \overline{\overline{A_D}}) : \overline{A_C} \subseteq \overline{A_D} \end{array}}{C <:@ D}$$

Figure 4.17 : TCFJ - Combined Subtyping Relation

For $C <:@ D$, it is required that $C <: D$ and that the sets of annotations of all of C's methods are subsets of the corresponding sets of D's methods. If this were not the case, if a method m in C contained a `@NotRunBy("foo")` annotation, but the same method m in D did not, then the set of threads allowed to invoke m has been narrowed. Code working only with D at an abstract level may expect that m can be invoked by thread `"foo"`, because the contract for m in D did not state otherwise.

More formally, this problem can be expressed as the equivalence in figure 4.18.

If Java compiles the source code for classes C and D, but the invariant checker emits a static subtyping warning, then it has detected potential problems when working with an instance of C, but treating it abstractly as a D. A method in class C has

$$
\begin{aligned}
C <: D \text{ but } C \not<:@\ D \quad &\Leftrightarrow \quad \exists \overline{A_{Ci}} \in \overline{\overline{A_C}} \text{ such that } A_{Ci} \not\subseteq A_{Di} \\
&\Leftrightarrow \quad \exists A \in A_{Ci} \text{ such that } A \notin A_{Di}
\end{aligned}
$$

Figure 4.18 : TCFJ - Static Invariant Violation

an additional invariant to maintain, but the code that uses class D abstractly is not aware that this invariant needs to be maintained.

For the classes and invariants shown in tables 4.1 and 4.2, the <: Java subtyping relation holds for the classes, and the <@ subtyping relation holds for the invariants, albeit in the reverse order. In both tables, the combined <:@ subtyping relation does not hold; therefore, subtyping warnings should be emitted in both cases.*

4.7.5 Extension of Featherweight Java Typing

To check whether a program is well-typed both using the original <: subtyping relation and using the <:@ subtyping relation defined above, the <:@ relation should be used instead of <:. The combined subtyping relation <:@ is more stringent and completely subsumes <: as one if its antecedents.

Furthermore, the $\boxed{M\ \texttt{OK in}\ C}$ and $\boxed{C\ \texttt{OK}}$ rules should be extended to also perform an $\overline{A}$ `OK` check.

The invariant checker uses the <:@ subtyping relation to detect potential problems with maintaining the concurrency discipline, because a violation here indicates that a subclass has more stringent invariants than the superclass, and therefore the superclass cannot be used abstractly. Doing so would risk breaking the invariants of the subclass.

*Bill Scherer named <:@ "ice cream cone". I hope this name becomes commonly used.

4.7.6 Type Preservation and Progress

The changes to the Featherweight Java syntax are mostly cosmetic: This extension adds predicate annotation declarations and annotations to the syntax, and classes and methods can be prefixed with annotations. The substantial change is the use of the stricter <:@ subtyping relation instead of the conventional <: subtyping relation.

Since <:@ subsumes <:, any program that is correct using the <:@ subtyping relation is also correct using the <: subtyping relation. If P is the set of all programs that are correct using <:@ , and Q the set of all correct programs using <:, then $P \subset Q$.

Therefore, it is not necessary to prove type preservation and progress for the extended version of Featherweight Java presented here, because the proofs given in the original Featherweight Java paper [43] still hold.

4.7.7 Implementation Differences

In the current implementation, a few things differ from the syntax and typing described above. Full Java allows overloaded types, so distinguishing methods just by name n is not sufficient; instead, the full signature is used: $C\ m(\overline{P\ x})$.

The class `Object` is treated specially in Featherweight Java and is assumed to contain no fields or methods; in the same spirit, we decided to disallow annotations on the class `Object` or its methods, even though the invariant checker implementation allows it.

The syntax of annotation is also slightly different: As Java does not allow multiple annotations of the same type on the same class or method, `@NotRunBy` and `@OnlyRunBy` actually contain arrays of annotations, which specify the thread. We felt it was unnecessary to reproduce the syntactic complexity here.

```
public class Benign {
  public void run(Object o1, Object o2) {
    System.out.println(o1.toString());
    System.out.println(o2.toString());
  }
  private static String s = "foo";
  public static void main(String[] args) {
    Benign b = new Benign(); b.run(s, s);
  }
}
```

Listing 4.29: Benign Program

4.8 Problems Caused By Subclassing: When Is Something Really "Thread-Safe"?

Another invariant that programmers may want to express is that a method can be called by any thread and does not need to meet any particular criteria, which can be done simply by not specifying any annotation, or by a gesture such as allowing all threads that match the regular expression `".*"` to run the method.

Superficially, this is true for most methods; however, closer inspection reveals that many apparently benign methods may still cause concurrency problems, especially in the face of subclassing or method calls to other objects. Listing 4.29 shows such a benign program that does not exhibit any concurrency problems.

Listings 4.30 and 4.31 show slight variations of that program that deadlock when executed and therefore are not as benign anymore.

- Listing 4.30 demonstrates that subclassing can affect a method's concurrency invariants. While the code in the `main` method is identical except for the fact that an instance of `MalignSubClass` is created instead of a `Benign` instance, the implementation of the `run` method differs: It acquires the first parameter's lock and prints out the first parameter. Then it starts `aux`, a second thread that acquires the second parameter's lock and print out the parameter's value.

```
public class MalignSubClass extends Benign {
  public void run(final Object o1, final Object o2) {
    synchronized(o1) {
      System.out.println(o1.toString());
      Thread aux = new Thread(new Runnable() {
        public void run() {
          synchronized(o2) {
            System.out.println(o2.toString());
          }
      } });
      aux.start();
      try { aux.join(); }
      catch(InterruptedException e) { }
    }
  }
  private static String s = "foo";
  public static void main(String[] args) {
    Benign b = new MalignSubClass(); b.run(s, s);
    // b.run(s, "bar"); would work
  }
}
```

Listing 4.30: Modified Behavior by Subclassing

The main thread waits for `aux` to finish and only relinquishes its hold on the first parameter's lock once `aux` has completed.

The `run` function described above prints out the same results as the one in listing 4.29 as long as the two arguments passed to the `run` method are distinct objects. In listing 4.30, however, the same object `s` is passed for both parameters, resulting in a deadlock, since the `aux` thread cannot acquire the object's lock until the main thread has released it, but that only happens once `aux` has completed.

The invariant that now needs to be maintained for the `MalignSubClass.run` method is that both arguments are distinct objects. Listing 4.30 therefore shows the danger of treating an object abstractly at the base class (`Benign`) even though the object may actually be an instance of a subclass (`MalignSubClass`).

```
public class MalignCall extends Benign {
  private static Object o = new Object() {
    public synchronized String toString() {
      return "foo";
  } };
  public static void main(String[] args) {
    final Benign b = new MalignCall();
    synchronized(o) {
      Thread aux = new Thread(new Runnable() {
        public void run() {
          b.run(o, o);
        }
      });
      aux.start();
      try { aux.join(); }
      catch(InterruptedException e) { }
    }
    // b.run(o,o); would have worked
  }
}
```

Listing 4.31: Modified Behavior by Call to Overridden Method

- In listing 4.31, the `run` method is left unchanged, but the arguments used impose an invariant: When the `toString` method is called, the argument's own lock has to be acquired. The program's `main` method has also been rewritten to call the `run` method in a second thread, `aux`.

  Listing 4.31 would have produced the same result as listing 4.29, but the main thread acquires the lock of the argument `o` before the second thread `aux` is started and does not release it until `aux` has finished running. Since `o`'s lock is already held by the main thread, `aux` cannot acquire it when its `run` method calls `o`'s `toString` method, and a deadlock ensues.

  Due to the kind of argument passed to it, the `MalignCall.run` method now needs to maintain the invariant that the locks of its arguments are not held at the time of the call. Listing 4.31 therefore exposes the problems caused by subclassed arguments, fields or return values.

The software described in this chapter is able to disarm the situation, though, as long as invariants are consistently annotated. If a method in a subclass introduces an invariant, but the method in a superclass does not, then the invariant checker can statically determine that a subtyping violation has been found. The invariant of the subclass method is stricter than the invariant of the superclass, and this violates the contravariance of preconditions. Such a subtyping violation typically indicates that the class hierarchy is used in an invalid way.

These static checks can help expose many concurrency problems, but overridden methods in subclasses or subclassed arguments remain problematic unless the subclasses and the classes of the arguments are properly annotated as well. If, for example, the `MalignSubClass.run` method in listing 4.30 expresses the invariant that the two arguments must be distinct, then the deadlock can be detected before it occurs.

If the overridden `toString` method in listing 4.31 expressed the invariant that the object's lock may not be owned by any thread yet, because the method itself will need to acquire it, the impending deadlock can also be detected. Listings 4.32 and 4.33 show the necessary annotations to express the aforementioned invariants. Listings 4.34 and 4.35 show the output produced by the software described in this chapter. The reports allow the programmer to pinpoint the location and nature of the violation.

This section established how dangerous it is to label a method as "thread-safe": If the method makes any calls to other methods that may have been overridden, or if the method itself is overridden in a subclass, the assumptions that lead to the determination of safety may be invalid. Instead of labeling a method as "thread-safe", it is better to turn the assumptions into annotations that specify the actual invariants.

```
@DistinctArguments({0,1})                                              2
public void run(final Object o1, final Object o2) { ... }              3
```

Listing 4.32: Listing 4.30 with Annotation (Excerpt)

```
@NoneSynchronizedThis                                                  3
public synchronized String toString() { return "foo"; }                4
```

Listing 4.33: Listing 4.31 with Annotation (Excerpt)

The only time a developer is really justified in calling a method "thread-safe" is when the method is final, and therefore prohibits overriding, and does not make any calls to methods that are not final as well.

4.9 Results

To evaluate how effective and easy to use the invariant checker and the annotations are, we annotated two different versions of DrJava [4], a version from March 26, 2004, and a version from September 2, 2006. These versions were chosen because they marked two stable releases of DrJava. Since the DrJava development team had recently made many changes to improve concurrent behavior of the application, we expected that the older version would have more invariant violations than the more recent version.

```
Thread Predicate Violation: (1 check, 1 violation)
        Current thread 'main', id 1, group 'main'
        Violated predicate @edu.rice.cs.cunit.threadCheck.
           predicates.DistinctArguments
        Method arguments
                'foo' : java.lang.String
                'foo' : java.lang.String
        at BenignSubClass.run (BenignSubClass.java:7)
        at BenignSubClass.main (BenignSubClass.java:22)
```

Listing 4.34: Invariant Checker Output for Listing 4.32

```
Thread Predicate Violation: (1 check, 1 violation)
        Current thread 'Thread-0', id 7, group 'main'
        Violated predicate @edu.rice.cs.cunit.threadCheck.
           predicates.NoneSynchronizedThis
        at BenignCall$1.toString (BenignCall.java:-1)
        at Benign.run (Benign.java:3)
        at BenignCall$2.run (BenignCall.java:14)
        at java.lang.Thread.run (Thread.java:613)
```

Listing 4.35: Invariant Checker Output for Listing 4.33

The process of annotating existing code was primarily guided by source code comments and Javadoc comments present in the source code. It should be noted that we probably missed some opportunities for annotation and only expressed a subset of the invariants actually present in the program. We also faced some problems building and running the 2004 version of DrJava, which was written before the final version of Java 5.0 was released; as a result, some unit tests could not be run.

Table 4.3 shows the number of unit tests for the two versions that passed successfully, failed, or could not be run because of problems with Java 5.0, as well as the total number of unit tests. It is evident that many tests were added to the unit testing suite of DrJava between the two versions.

	3/26/2004 Version	9/2/2006 Version
Unit Tests Passed	610	881
Unit Test Failures	36	0
Could Not Run	90	0
Total Unit Tests	736	881

Table 4.3 : Unit Tests

Table 4.4 shows the total number of invariant checks, the number of passed and failed checks, and the percentage of failed checks during the execution of the entire test suite. While there were more check failures in the 2006 version in absolute terms, the percentage of failed invariant checks was significantly lower in the newer version, reflecting the development team's perception that concurrent behavior had improved. The number of invariant checks was lower in the 2004 version, though, since there were fewer comments that dealt with concurrency. This made the annotation process for the 2004 version more difficult.

	3/26/2004 Version	9/2/2006 Version
Invariant Checks Failed	965	3796
Invariant Checks Passed	4161	30616
Total Invariant Checks	5116	34412
Percentage Failed	18.83	11.03

Table 4.4 : Invariant Checks and Violations

This is corroborated by the information in table 4.5: On the one hand, the source base and the number of unit tests grew substantially, and on the other hand, the term "event thread", a concept central to Java AWT/Swing threading disciplines, was mentioned hardly at all in the 2004 version, but frequently in the 2006 version.

It should also be noted that the 34,412 invariant checks passed in the 2006 version did not prolong the testing process measurably. In general, as long as there was sufficient information available to establish concurrency invariants, it was easy to add annotations and have them checked. It was also simple to annotate only a part of the codebase without annotating the whole codebase, and it was easy to carry out

	3/26/2004 Version	9/2/2006 Version
KLOC	107	129
Total Unit Tests	736	881
Mentions “event thread”	1	99

Table 4.5 : Other Information

the invariant checking in addition to running the unit testing suite. We expect that it will be much easier to annotate a program at the same time it is written, when the programmers are actually the most aware of the required invariants.

To compare the performance of reflection-based invariant checker and the bytecode-rewriting checker, we ran several benchmarks. The programs all made 2000 calls to a method that sleeps for 100 ms. The method was annotated with different invariants, and then run without the invariant checker, with the checker using bytecode rewriting, and with the checker using reflection. The invariants chosen for these benchmarks are guaranteed to succeed, since an invariant violation would cause an undesirable write to a log file.

The invariants are listed in the first column of table 4.6. A, B, C and D are placeholders for invariants of the form “not executed by thread with name...”. The second column indicates whether an instrumentation was performed, whether primitive or predicate annotations were used, and whether reflection was used as opposed to bytecode rewriting. The third column contains the time for 2000 method calls, generated on an Intel Core 2 Duo E6600 CPU (2.4 GHz, 1066 MHz FSB) with 2 GB of DDR2 SD RAM (667 MHz). The last column shows the normalized execution times, where the factor of 1.0 corresponds to the execution time without instrumentation, listed first in each group.

The factors show that the invariant checker that uses bytecode rewriting has negligible impact; the overhead caused by the reflection-based invariant checker is more noticeable. As the invariants become more complex, the advantage of bytecode rewriting increases. We also found that there was no performance penalty for using the more powerful predicate annotations instead of the primitive annotations.

The invariant checker presented in this chapter works on all three major platforms: Windows, Linux, and Mac OS X.
All source code is open source and available at: `http://www.concutest.org/` [38].

Invariant	Description	Time (ms)	Factor
A	primitive, not instrumented	201172	1.00000
	primitive, instrumented	201203	1.00015
A	predicate, not instrumented	201172	1.00000
	predicate, instrumented bytecode	201187	1.00007
	predicate, instrumented, reflection	201281	1.00054
$A \wedge B$	primitive, not instrumented	201172	1.00000
	primitive, instrumented	201203	1.00015
$A \wedge B$	predicate, not instrumented	201172	1.00000
	predicate, instrumented, bytecode	201187	1.00007
	predicate, instrumented, reflection	201297	1.00062
$((A \wedge B) \wedge (C \wedge D))$	predicate, not instrumented	201187	1.00000
	predicate, instrumented, bytecode	201188	1.00000
	predicate, instrumented, reflection	201328	1.00078

Table 4.6 : Invariant Checker Benchmarks

Chapter 5

Bytecode Rewriting Framework

Except for the improvements to JUnit described in chapter 3, all of the pieces of the framework use bytecode rewriting: Java source files are compiled with the regular Java compiler, if possible with debug information, and then analyzed and rewritten.

The alternative would have been to rewrite Java source code, but this approach has several disadvantages:

- It is more difficult to parse a Java source file due to all the variations that are allowed.
- Some of the instrumentations that are performed do not have a corresponding Java source equivalent. For example, it is impossible in Java to just emit a `monitorenter` instruction without a matching `monitorexit` instruction.
- Sometimes, the Java source simply is not available or it is not advisable to recompile it, for example the classes of the Java API.
- Performing the instrumentation on-the-fly, using a custom class loader, would have been much more difficult if the changes were made to Java source and not to class files.

Another option would have been to modify the Java compiler and runtime environment, but this was ruled out early on so the project could target as many platforms as possible and therefore maximize the potential user base.

What is analyzed and how the class files are changed depends on the required task. In general, though, class files are rewritten one at a time; the actual process has

```
public interface IInstrumentationStrategy {          1
  public void instrument(ClassFile cf);              2
  public void done();                                3
}                                                    4
```

Listing 5.1: `IInstrumentationStrategy` Source

been abstracted out using several object-oriented design patterns, namely strategy, decorator and composite [44], so that each instrumentation can implement the necessary changes as it sees fit. Very often, the instrumentation strategy cycles through all the methods in a class and changes them.

`IInstrumentationStrategy` is the base interface, and all instrumentation strategies need to implement its two methods shown in listing 5.1. `instrument` will be called once per class file; `done` is called only once, at the end of the instrumentation, when all classes have been processed.

There are several helpful classes and interfaces that allow better code reuse:

`CompoundStrategy` bundles several `IInstrumentationStrategy` instances and runs them one after the other. A `ConditionalStrategy`, shown in listing 5.2 also contains another `IInstrumentationStrategy`, but it will only execute the strategy if the `ConditionalStrategy.apply` method returns `true`.

Using a `ConditionalStrategy`, a developer can apply an instrumentation only to class files that, for example, reside in the `java.lang` package. There is also an additional interface that instrumentation strategies can implement, `IScannerStrategy`, which adds another method that needs to be implemented and whose purpose it is to return data gathered during the instrumentation. The interface's definition is shown in listing 5.3.

The classes implementing `IScannerStratey` typically cache data from the instrumentation of one class to the next, and possibly process it when the `done` method is called. Another common use of the `IScannerStrategy` is storing minor errors that

```
public abstract class ConditionalStrategy                              1
  implements IInstrumentationStrategy {                                2
  IInstrumentationStrategy _decoree;                                   3
  public ConditionalStrategy(IInstrumentationStrategy decoree) {       4
    _decoree = decoree;                                                5
  }                                                                    6
  public void instrument(ClassFile cf) {                               7
    // ...                                                             8
    if (apply(cf)) {                                                   9
      // ..                                                            10
      _decoree.instrument(cf);                                         11
    }                                                                  12
  }                                                                    13
  public void done() { _decoree.done(); }                              14
  public abstract boolean apply(ClassFile cf);                         15
}                                                                      16
```

Listing 5.2: `ConditionalStrategy` Source

```
public interface IScannerStrategy extends                              1
   IInstrumentationStrategy {
  public interface IScanResult {                                       2
      public String getPropertyName();                                 3
  }                                                                    4
  public List<? extends IScanResult> getScanResults();                 5
}                                                                      6
```

Listing 5.3: `IScannerStrategy` Interface

should be relayed to the user but that should not entirely terminate the instrumentation, as an exception would.

Most instrumentation strategies also take a `List<String>` as parameter in their constructors: This list allows the user to pass values to the instrumentation strategies and, for example, determine whether backup files should be created before a class file is changed.

To parse, analyze and modify Java class files, we decided to write our own library, even though alternatives existed: BCEL [45] was already very well developed and could have been considered the product of choice when this project started; however, BCEL seemed to force us into a certain development mold that felt too rigid and unnecessarily complicated. ASM [46], a newer, lighter-weight library, puts more of emphasis on speed, but when the Concutest project was started, ASM was still too incomplete to use.

Our framework consists of a mixture of high-level classes that employ object-oriented design patterns like strategies, compounds and visitors, as well as low-level constructs for the compact representation of intructions, methods and classes. All features of Java 5.0 and 6.0 are fully supported by the library.

It took some time to write the library, but we believe this effort was necessary to get acquainted with all parts of the Java class file format and the intricacies of the JVM.

5.1 Offline and On-the-Fly Instrumentation

Regardless of which instrumentation strategy is used, instrumentation can be done either offline, after compile time but before runtime, or at runtime using a custom class loader that rewrites the classes just as they are needed.

To instrument classes offline, a set of class files, directories, and jar files is passed as argument to the `FileInstrumentor` program, together with the name of the `IInstrumentationStrategy` that should be applied. The `FileInstrumentor` then

processes all class files that were specified and replaces the originals with instrumented copies. After the instrumentation, the classes can be used just as if nothing had changed.

Offline instrumentation is faster, safer, more accurate and more general than on-the-fly instrumentation: It is faster because caching changes is easier, and because the JVM does nothing else besides instrumenting class files.

Performing the instrumentation offline is safer and more accurate than changing the class files on-the-fly because the instrumentation cannot have any side effects on computations that happen concurrently. With on-the-fly instrumentation, care must be taken to minimize the impact the custom class loader has on the rest of the program. If the custom class loader affected the behavior of the program being instrumented and changed the outcome of a particular unit test, that would negate the effort of the framework.

Finally, several classes in the Java API are considered "protected" and cannot be changed on-the-fly. In order to change them, the instrumentation has to be performed offline. For all of these reasons, it is recommended to instrument the Java API (usually called `rt.jar` on Windows and Linux, or `classes.jar` and `ui.jar` on Mac OS X) offline and create instrumented copies to be used instead of the original API files. To use the instrumented copies, they have to be placed at the beginning of Java's boot classpath using the `-Xbootclasspath/p` option. Our framework provides GUI tools to assist the user with this.

5.2 Local and Global Instrumentation

Each instrumentation strategy, each change of the program, can be classified as either local or global, depending on what parts of the program need to be modified to achieve the desired effect.

The changes from chapter 4 to check whether the program has violated the threading discipline are an example of local instrumentation. Bytecode is inserted in one

place, at the beginning of a method, but the results are observable throughout the entire program, at every call site of the method. Converting `synchronized` methods to methods with a `synchronized` block is another example of local instrumentation. A change is made in one method only, the rest of the program does not have to be modified.

Not all changes can be made in this way, though. In order to record all synchronization events of a program, for example, it is necessary to know when a program calls the `Object.wait`, `Object.notify`, and `Object.notifyAll` methods. The easiest way to achieve this would be to insert the bytecode doing the recording into these methods; in the current Java API, however, these methods are native and therefore do not contain bytecode.

The next easiest way to be notified of every call to the methods above would be to rename the original methods, for example `Object.wait` to `Object.waitOriginal`, and then put a method with the original name in its place, a method that records the invocation and then forwards the call to the renamed method. That way, the changes would still be localized to the `Object` class alone; all other classes could remain unchanged and would nonetheless call the method we put in place. This works for some methods, but it does not help in the case of `Object.wait`, `Object.notify`, and `Object.notifyAll`. These methods are native, and linking the native code to the methods requires the methods to always have the original name.

To still be notified of calls to these methods, an instrumentation strategy has to create forwarding methods, for example `Object.waitForward`, that do the recording and then call the native methods. Unfortunately, all the other classes still call the original method, so now all call sites in all classes need to be changed to call the added forwarding method. That makes this kind of instrumentation global; the changes are not localized to a single class anymore, but affect every class that uses the changed method. Local instrumentation is preferable to global instrumentation since it reduces the number of times code has to be rewritten.

5.3 Marker Methods for Instrumentation

In some occasions, it may be necessary to place a specific bytecode or a sequence of instructions at a certain place in a Java method, but the Java language forbids this. One prime example is the strictly enforced pairing of `monitorenter-monitorexit` instructions. Even though they are individual bytecode instructions, they always appear together because they express one Java construct, the synchronized block.

In certain circumstances, we have found it desirable to emit only a `monitorenter` instruction without a following `monitorexit` opcode, or vice versa. An easy way of achieving this while working with high-level Java code is the use of marker methods.

These marker methods are empty `public static void marker(Object o)` methods and not meant to be executed. When the `MarkerInlineStrategy` instrumentation strategy encounters the call to such a method, the bytecode making the call is replaced by the instructions that the programmer wants to be to be inserted at that location, the lone `monitorenter` instruction for example.

Marker methods and an inlining instrumentation strategy allow the developer to work at a conceptionally high level with Java methods and classes, but violate the Java model by using features only available to programs written directly in bytecode.

5.4 Other Uses of Bytecode Rewriting

In addition to using bytecode rewriting for the annotation language to specify concurrency invariants in chapter 4, we have applied the same techniques to other problems: Recording a program's schedule, monitoring for deadlocks, and modifying a program's schedule using random delays and yields.

All of these tools are in one way or another tied to the schedule of a program and either monitor, record or attempt to change it. In order to prevent tampering with the schedule, the main algorithms are executed in a second JVM, called "master JVM", that is connected to the program's JVM, called "client JVM", using the Java

Debug Interface (JDI). The program's JVM is only required to handle an array of primitive data, a buffer in which the schedule is encoded.

5.4.1 Recording Schedules

To record a program's schedule, the program and the entire Java API are instrumented to add values to the buffer. Table 5.1 shows the codes used for the different events.

This table should be extended for other operations that can influence concurrent behavior, such as access to `volatile` variables or remote method invocation (RMI).

The smallest amount of data that must be recorded to allow scheduled replay, the next phase of this project, is the type of event and the thread that encountered the event. Therefore, at a bare minimum, the schedule consists of "thread ID"-"code" pairs.

In order to keep threads apart, the constructor of the `Thread` class is modified to keep a hidden counter that is incremented every time a new thread is created. As discussed before, the increment operation is a compound operation and thus prone to race conditions. To avoid those and ensure that thread IDs are assigned in a deterministic fashion, assigning and incrementing the counter is done in a synchronized block. Since that block did not exist in the original program but is nonetheless important for accurately representing the schedule, the codes 9 through 11 are used for this synchronized block, not 1 through 3 as in the rest of the program.

To fit into the pattern of using 1 through 3 for acquiring, releasing, and attempting to acquire regular locks, synchronized methods are rewritten to be normal unsynchronized methods with a synchronized block inside, as has been explained before. Another benefit of rewriting synchronized methods is that the instrumentation remains local: Only the method with the synchronized block needs to be rewritten. If synchronized methods were not converted to regular methods with a synchronized block, then the event that represents the attempt to acquire a lock (code 3) would have to be emitted before the call to the synchronized method is made; this would

Code	Event
1	after `monitorenter`, lock obtained
2	before `monitorexit`, about to release lock
3	before `monitorenter`, lock not yet obtained
4	thread started
5	thread terminated
6	after `monitorenter` when assigning a unique object ID
7	before `monitorexit` when assigning a unique object ID
8	before `monitorenter` when assigning a unique object ID
9	after `monitorenter` when assigning a unique thread ID
10	before `monitorexit` when assigning a unique thread ID
11	before `monitorenter` when assigning a unique thread ID
12	end of schedule

Table 5.1 : Schedule Codes

mean changing every call site, in every method, in every class, representing an instrumentation with global effects.

If the user desires, additional data can be added to the schedule buffer as well: a unique object ID, created in a similar way as the unique thread ID described above, a number indicating the class in which the event occurred, a number to point out

the method in the class, and the value of the program counter (PC) when the event occurred. When this debug information is included, the buffer contains "thread ID"-"code"-"class index"-"method-and-PC"-"object ID" quintuples.

When the buffer is full, the program's VM enters a special method that resets the index to 0 so the old data is overwritten. The master JVM that is running the analysis has set a breakpoint at the beginning of that method, though, and can therefore copy the contents of the buffer before they are overwritten. This way of transferring the buffer data is called "client push", because the client program initiates the transfer. The master JVM may also transfer the data any time in what is called a "master pull", except when the client JVM is manipulating the array, as that would create a race condition; therefore, the master JVM checks if the lock protecting the buffer is owned in the client JVM, and if that is the case, delays the access until the lock is released.

The schedule is formatted to be useful in a future tool that replays a program using the recorded schedule. But even without the replay tool, looking at a program's schedule can be insightful.

5.4.2 Deadlock Monitor

An additional benefit of creating the tool for recording schedules was the possibility of creating a deadlock monitor. As long as the schedule provides information about what thread is trying to acquire which lock, this information can be used to create a wait-for graph in the master JVM, search for cycles and detect deadlocks.

The unique object ID is added to objects in a way similar to the unique thread ID explained above; however, for a unique object ID, there were several additional complications:

- While methods can be added to `java.lang.Object`, it is impossible to add fields to the class, because an `Object` has a fixed representation inside the JVM.

- Similar problems were encountered with `String`, `Number`, and `Class`.

To circumvent this problem and still get the most accurate results possible, the instrumentor adds a new hidden method to the `java.lang.Object` class that returns the object's identity hashcode. The identity hashcode can be used as an approximation and it is unlikely that two distinct objects have the same hashcode, but uniqueness is not guaranteed (contrary to the common belief that equal identity hashcodes imply uniqueness; see Java bug report 6321873 [47]). All subclasses of `Object`, except for the problematic classes `String`, `Number`, and `Class`, then override the method to provide a real unique object ID controlled by bytecode that was added during the instrumentation. These real object IDs are always negative, while identity hashcodes are always positive. They are therefore easy to distinguish, and 0 can be used to indicate an object has not been assigned an ID yet.

The deadlock detector in the master JVM monitors the incoming schedule data and adjusts the wait-for graph whenever a thread attempts to enter a synchronized block, manages to obtain a lock and enter a synchronized block, or releases a lock and leaves a synchronized block. If the deadlock detector ever finds a cycle in the wait-for graph, the involved threads are displayed, together with debug information such as class, method, program counter and, if debug information is available, source file and line number. If the DrJava IDE is available, the source can be loaded with a single mouse click.

Using the object ID numbers of objects involved in a deadlock, it is even possible to find out where the object was created since the assignment of the unique ID number was accompanied by event codes 6 through 8.

The program is not aborted, and the threads not involved in the deadlock continue to run. The deadlock detector is fully equipped to handle even multiple deadlocks.

5.4.3 Random Delays and Yields

Since we have not yet implemented an algorithm to replay a program according to a specific schedule, we have experimented with an alternative approach: Inserting

random delays or yields before or after certain critical operations that can influence the behavior of the concurrent program.

Calls to `Thread.sleep` or `Thread.yield` can be inserted before any `monitorenter` opcode or call to `Object.wait`, `Object.notify`, `Object.notifyAll`, `Thread.join`, and `Thread.exit`. They can be inserted after any `monitorenter` opcode or call to `Object.wait`, `Thread.join`, `Thread.start`, and `Thread.run`.

It is important that both `Thread.start` and `Thread.run` are instrumented, since the former is still called by the parent thread, while the latter is the first method called by the child thread.

The length and probability that a delay or yield will be inserted can be changed, but so far it is not known what probabilities and durations are most likely to uncover concurrency issues. With certain test programs, we can demonstrate that random delays and yields have the potential to expose concurrency problems, but we cannot yet make an educated guess what values would work on real applications.

Listing 5.4 shows a program with a concurrency flaw that the random delay strategy detected. It uses two threads, and one thread assumes that the other thread has finished its work, but this dependency is not expressed in code, for example using a `Thread.join`.

The two threads share a variable `int sharedInt` that starts out at 0. Access to it is protected by the field `lock`, so there are no race conditions. The worker thread spends some time doing something (simulated by the `lengthyProcessing` method), then increments `sharedInt`, and finishes.

The main thread starts the worker thread, also spends some time doing something, and then decrements `sharedInt`. The amount of work is set up so that under normal conditions, the worker thread will have finished and the value of `sharedInt` will be 1. The main thread asserts that after decrementing `sharedInt`, its value is back at 0.

```
public class SyncProblem2 extends TestCase {                          1
  Character lock = new Character(1);                                  2
  volatile int sharedInt = 0;                                         3
  public void testUnexpressedDependency() {                           4
    Thread worker = new Thread(new Runnable() {                       5
      public void run() { synchronized(lock) {                        6
        sharedInt = sharedInt + 1;                                    7
    } } });                                                           8
    worker.start();                                                   9
    lengthyProcessing(Main thread, 10000);                            10
    synchronized(lock) {                                              11
      sharedInt = sharedInt - 1;                                      12
      assertEquals(The shared integer should be back to 0,            13
                   0, sharedInt);                                     14
    }                                                                 15
  }                                                                   16
  private void lengthyProcessing(String threadName, int               17
     iterations) {
    for(int i=0; i<iterations; ++i) { }                               18
  }                                                                   19
}                                                                     20
```

Listing 5.4: “Too Late” Faulty Program

When this test is run after the random delay instrumentation has been performed, delays have been inserted, in particular at the beginning of the worker thread's `run` method. This postpones incrementing `sharedInt` long enough so that the main thread can decrement it first, resulting in a value of -1 and a failed assertion as seen in listing 5.5. The assertion did not fail without the random delay instrumentation.

This is just a simple example, but it demonstrates that problems in the category “concurrent task had not finished when it was expected” can be detected with random delays and the right settings.

What if the main thread expects that thread 2 has not finished performing some task, but it actually has? This is another common problem, and it is particularly important if the code involves calls to `Object.wait` and `Object.notify` or `Object.notifyAll`. One thread may be waiting for another thread's notification, but that

```
[junit] Testcase: testUnexpressedDependency(SyncProblem2):
   FAILED
[junit] The shared integer should be back to 0 expected:<0> but
   was:<-1>
[junit] junit.framework.AssertionFailedError: The shared integer
   should be back to 0
        expected:<0> but was:<-1>
[junit]     at SyncProblem2.testUnexpressedDependency(
   SyncProblem2.java:33)
BUILD FAILED
/Users/mgricken/Documents/Research/Concutest/ClassLoader/build.
   xml:708: Test SyncProblem2 failed
```

Listing 5.5: Fault Detected by Random Wait

notification had already occurred before the first thread was ready to receive it. Listing 5.6 contains an example.

In this listing, the calls to `Thread.sleep` simulate performing some computation. Under normal circumstances, the worker thread will call `signal.notify` long after the main thread has reached `signal.wait`, so the main thread is woken up.

If, however, the main thread for some reason takes longer than usual to get to `signal.wait`, then the notification may be lost. The correct way of doing this is to include a flag that is protected by the lock of the same object being used for signaling: The flag is initially `false`, gets set to `true` before the call to `Object.notify`, and `Object.wait` is only called if the flag is still `false`.

Note that if the `Object.notify` is reached first and the notification is lost, then this unit test does not fail but hangs indefinitely. For this reason, any unit test should have a timeout set, and if the test has not finished executing after the specified time has run out, the test should be considered a failure. An examination of the thread stacks would then show that one of the threads had made a call to `Object.wait`.

There are two yet unverified suggestions how this situation could be improved further:

```
public class SyncProblem3 extends TestCase {                 1
  public void testNotifyTooEarly() {                         2
    Thread worker = new Thread(new Runnable() {              3
      public void run() {                                    4
        try { Thread.sleep(2000); }                          5
        catch(InterruptedException e) { }                    6
        synchronized(signal) { signal.notify(); }            7
        try { Thread.sleep(3000); }                          8
        catch(InterruptedException e) { }                    9
      }                                                      10
    });                                                      11
    worker.start();                                          12
    try {                                                    13
      synchronized(signal) {                                 14
        signal.wait();                                       15
      }                                                      16
    }                                                        17
    catch(InterruptedException e) { / ignore */ }            18
  }                                                          19
}                                                            20
```

Listing 5.6: “Too Early” Faulty Program

- The test could be run with a different version of `Object.wait` that always includes a timeout. If the time is exceeded, the method throws an exception, forcing the test to fail. The problem again is the choice of the timeout length. Some tests could potentially run for a very long time and then succeed, and that is the expected behavior.

- In some cases, it may be possible to detect that a thread cannot be woken up again, either because there are no other user threads left alive, or because there is a deadlock. The first case is easy to check just before the call to `Object.notify`; the second case would require the deadlock detector. Preliminary tests indicate that the execution of an `Object.wait` call without timeout should be broken down into a series of `Object.wait` calls with timeouts and interspersed checks of the number of living threads.

```
public synchronized int getLength() { ...  }                        1
public synchronized void remove(int offs, int len) { ...  }         2
// ...                                                               3
remove(getLength()-1, 1);                                            4
```

Listing 5.7: Possible Atomicity Hazard

If the program kept track of the number of live, waiting, and dead threads, then it could easily prevent a thread from making a call to `Object.wait` that the framework knows is doomed, because no other thread is alive.

The random sleeps and yields are expected to also help with compound actions that really should perform atomically, like the example from DrJava's source code, summarized in listing 5.7.

If the two methods that should execute atomically are synchronized, then there is a reasonably high probability that a delay will be inserted in between them, uncovering an atomicity violation that can be detected in a unit test.

Aside from these conceptual proofs that random yields and random sleeps can help detect concurrency problems, and aside from providing the means for changing the probabilities and durations of the individual delays, we have not studied the efficacy of these techniques on large programs like DrJava or other applications with unit testing suites.

We also believe that a purely random trial can be improved upon. There exist certain relationships among the list of places where delays and yields can be inserted, and adding a delay or yield in one location could destructively interfere with another delay or yield added somewhere else.

Here are some of the relationships that we believe exist:

- `Object.wait` vs. `Object.notify` vs. `Object.nofityAll`
- `Thread.run` vs. `Thread.join`
- `Thread.run` vs. `Thread.start`

- `Thread.exit` vs. `Thread.start`

These instrumentation points form a graph of relationships from which certain non-interfering subsets of insertion points can be constructed. The program is then run several times with each subset, which should minimize destructive interference and increase the probability that the random delays or yields change the schedule in a detectable way.

A closer inspection of the effects of one delay or yield on another has not been done yet, and no complete graph has been constructed.

5.5 Results

Bytecode rewriting has proved to be tremendously flexible and has helped create the low-overhead invariant checker from chapter 4. It has spawned a number of tools that read and analyze the schedule being executed, and it is even being used in attempts to exert control over the schedule. Ultimately, the recorded schedule will serve as input for a scheduler implemented using bytecode rewriting that will replay existing schedules as well as schedules derived from them.

Work on a replay algorithm has already begun, and a Promela/SPIN [48] model has been created, but we have not yet ported the Promela program back to Java and into the confines of the JVM.

Currently, all the tools discussed in this section use a two-JVM model to separate the code being analyzed from the slow portions of the code that analyze, decide, and rewrite. The buffer of primitives definitely has several advantages over the initial implementation using "fat" object instances that were immediately transferred:

- As the graph in figure 5.1 shows, buffered transfers perform a lot better than immediate transfers, although the benefit obtained by an increase of the buffer size diminishes as its size increases. This implies that a major cost of a transfer is the initiation of the transfer itself and not the amount of data transferred.

- The buffer with primitive values encoding the threads and objects involved in an event is much more effective than the immediate transfer of the thread object and lock object at the time the event occurs.
- By encoding the events as primitive data, the real objects are not referenced anymore, and they are not prevented from being garbage-collected.

After the success of the invariant checker, we will take another look at one-JVM debugging tools, which would eliminate the inter-JVM transfers entirely. Creating a tool that can accurately reproduce a previously recorded input schedule will be a crucial and urgently awaited step.

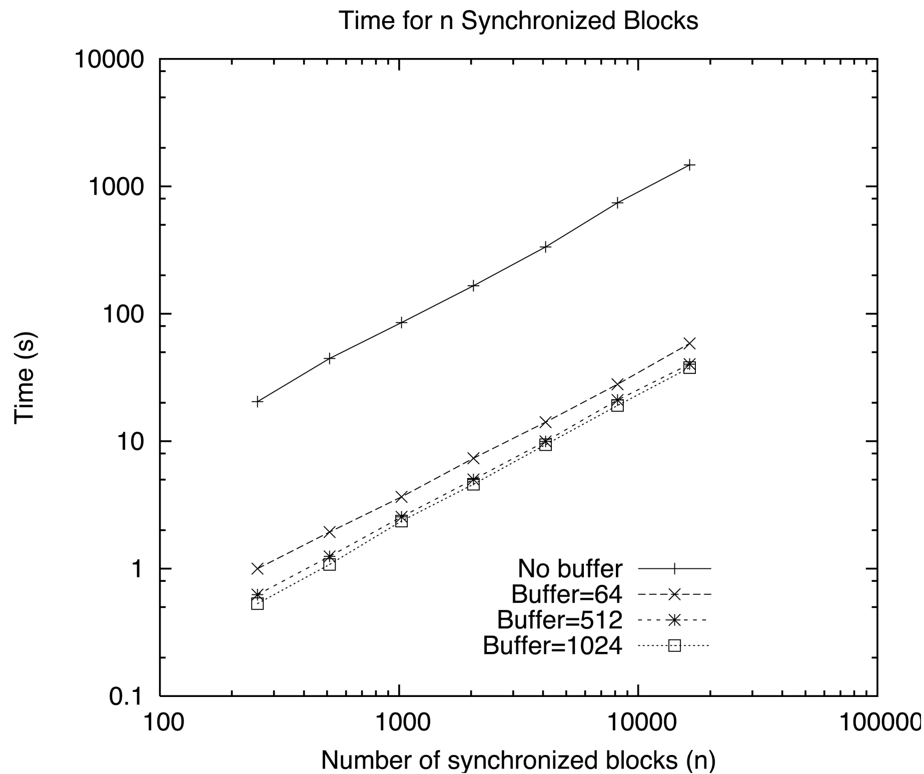

Figure 5.1 : Time for n Synchronized Blocks

The graph shows the time it takes to execute n tight `synchronized(o) { }` blocks.

Chapter 6

Conclusion

This thesis has provided an enhanced version of JUnit that is able to handle uncaught exceptions and failed assertions in all threads, not just the main thread. Additionally, an annotation language has been created that allows the developers of a library to express the concurrency invariants that programmers using the library need to maintain. Most checks occur at runtime, but a large number of checks can also be performed statically, just after compile time. The software that accompanies this thesis therefore provides a solid foundation for testing the correctness of a program in a single schedule, the schedule that was nondeterministically chosen by the platform.

The thesis also explored tools for recording a program's schedule, detecting deadlocks, and inserting random delays and yields to modify the schedule chosen to execute a program.

All these tools provide many of the necessities of a future testing framework that can generate and replay schedules. The annotation language may also be extended to check data invariants or postconditions, and additional static analysis may be added to automatically generate some of the invariants, for example that code executed in a program's `main` method is run by the main thread. An extended listing of invariants gathered from the Concutest Community Project [39] is expected to be of tremendous use as well.

The tools described in this thesis work on all three major platforms: Windows, Linux, and Mac OS X.

All source code is open source and available at: `http://www.concutest.org/` [38].

Bibliography

[1] Ron Jefferies and et al, *Extreme Programming Installed* (Addison-Wesley, Boston, MA, USA, 2001).

[2] Ron Jefferies, XProgramming.com, `http://www.xprogramming.com`.

[3] The Apache Ant Project, Apache Ant, `http://ant.apache.org/`.

[4] Rice JavaPLT, DrJava Web Site, `http://drjava.org`.

[5] Thomas J. LeBlanc and John M. Mellor-Crummey, *Debugging Parallel Programs with Instant Replay* (in IEEE Trans. Comput., **36**, IEEE Computer Society, Washington, DC, USA, 1987), No. 4, pp. 471–482.

[6] Jong-Deok Choi and Harini Srinivasan, *Deterministic Replay of Java Multithreaded Applications* (in SPDT '98: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools, ACM Press, New York, NY, USA, 1998), pp. 48–59.

[7] A. Georges, M. Christiaens, M. Ronsse, and K. D. Bosschere, *JaRec: a Portable Record/Replay Environment for Multi-Threaded Java Applications* (in Software—Practice & Experience, **34**, John Wiley & Sons, Inc., New York, NY, USA, 2004), No. 6, pp. 523–547.

[8] Andy Schneider, JUnit Best Practices, `http://www.javaworld.com/javaworld/jw-12-2000/jw-1221-junit.html`, 2000.

[9] Robert W. Nettleton, JUnit Best Practices, `http://www.ftponline.com/javapro/2003_11/online/rnettleton_11_26_03/`, 2003.

[10] Eric Allen, Diagnosing Java Code: Recorders test for proper method invocation, `http://www.ibm.com/developerworks/java/library/j-diag0612.html`.

[11] N. Alex Rupp, Multithreaded Tests with JUnit, `http://today.java.net/pub/a/today/2003/08/06/multithreadedTests.html`, 2003.

[12] Derek L. Bruening, *Systematic Testing for Multithreaded Programs* (Master's thesis, MIT, Cambridge, MA, USA, 1999).

[13] Derek Bruening and John Chapin, *Systematic Testing for Multithreaded Programs* (MIT/LCS, Cambridge, MA, USA, 2000), No. LCS-TM-607.

[14] Stefan Savage et al, *Eraser: A Dynamic Data Race Detector for Multithreaded Programs* (in ACM Trans. Comput. Syst., **15**, ACM Press, New York, NY, USA, 1997), No. 4, pp. 391–411.

[15] JUnit Project, JUnit Website, `http://www.junit.org`.

[16] TestNG Project, TestNG Website, `http://testng.org`.

[17] David Evans, *Static Detection of Dynamic Memory Errors* (in Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM Press, New York, NY, USA, 1996), pp. 44–53.

[18] William R. Bush, Jonathan D. Pincus, and David J. Sielaff, *A Static Analyzer for Finding Dynamic Programming Errors* (in Software—Practice & Experience, **30**, John Wiley & Sons, Inc., New York, NY, USA, 2000), No. 7, pp. 775–802.

[19] Sun Microsystems, Inc., JSR 305: Annotations for Software Defect Detection, `http://www.jcp.org/en/jsr/detail?id=305`.

[20] Cormac Flanagan and Martín Abadi, *Types for Safe Locking* (in Proceedings of the Eighth European Symposium on Programming, Springer, Berlin, 1999), pp. 91–108.

[21] Cormac Flanagan and Stephen N. Freund, *Type-Based Race Detection for Java* (in ACM SIGPLAN Notices, **35**, ACM Press, New York, NY, USA, 2000), Vol. 35, pp. 219–232.

[22] Christian Skalka and Scott Smith, *Static Enforcement of Security with Types* (in ACM SIGPLAN Notices, ACM Press, New York, NY, USA, 2000), pp. 34–45.

[23] Robert DeLine and Manuel Fähndrich, *Enforcing high-level protocols in low-level software* (in PLDI '01: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, ACM Press, New York, NY, USA, 2001), pp. 59–69.

[24] Maurice Herlihy, *Wait-free synchronization* (in ACM Transactions on Programming Languages and Systems, **13**, ACM Press, New York, NY, USA, 1991), No. 1, pp. 124–149.

[25] Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir, *Dynamic-sized lock-free data structures* (in PODC '02: Proceedings of the 21st Annual Symposium on Principles of Distributed Computing, ACM Press, New York, NY, USA, 2002), pp. 131–131.

[26] Håkan Sundell, *Efficient and Practical Non-Blocking Data Structures* (Ph.D. thesis, Chalmers University of Technology and Goteburg University, Goteburg, Sweden, 2004).

[27] Keir Fraser and Tim Harris, *Concurrent programming without locks* (in ACM Trans. Comput. Syst., **25**, ACM Press, New York, NY, USA, 2007), No. 2, p. 5.

[28] University of Cambridge Systems Research Group, Practical lock-free data structures, `http://www.cl.cam.ac.uk/research/srg/netos/lock-free/`.

[29] Atlassian Software Systems Pty Ltd., Clover Code Coverage for Java, `http://www.cenqua.com/clover/`.

[30] JUnit Project, GroboUtils, http://groboutils.sourceforge.net/.

[31] Kohsuke Kawaguchi, Parallel JUnit, https://parallel-junit.dev.java.net/.

[32] Sun Microsystems, Inc., JSR 133: Java Memory Model and Thread Specification Revision, http://jcp.org/en/jsr/detail?id=133.

[33] Jeremy Manson, *The Java Memory Model* (Ph.D. thesis, University of Maryland, College Park, MD, USA, 2004).

[34] Jeremy Manson, William Pugh and Sarita V. Adve, *The Java Memory Model* (in Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005), ACM Press, New York, NY, USA, 2005), pp. 378–391.

[35] Sun Microsystems, Inc., Java Language Specification, http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html.

[36] Sun Microsystems, Inc., Javadoc for Thread.UncaughtExceptionHandler, http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Thread.UncaughtExceptionHandler.html.

[37] Koders, Inc., Koders.com, http://www.koders.com.

[38] Mathias Ricken, Concutest Project Website, http://www.concutest.org.

[39] Mathias Ricken, Concutest Community Project, http://community.concutest.org.

[40] Aspect Research, Associates, Contract4J, http://www.contract4j.org/contract4j.

[41] The Eclipse Foundation, AspectJ, http://www.eclipse.org/aspectj/.

[42] Benjamin C. Pierce, *Types and Programming Languages* (MIT Press, Cambridge, MA, USA, 2002).

[43] B. Atshushi Igarashi and Philip Wadler, *Featherweight Java: A Minimal Core Calculus for Java and GJ* (in Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99), **34**, ACM Press, New York, NY, USA, 1999), No. 10, pp. 132–146.

[44] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, New York, NY, USA, 1994).

[45] Apache Jakarta Project, BCEL Byte Code Engineering Library Website, `http://jakarta.apache.org/bcel/`.

[46] Object Web, ASM Java Bytecode Manipulation Framework, `http://asm.objectweb.org/`.

[47] Sun Microsystems, Inc., Java Bug 6321873, `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6321873`.

[48] G. J. Holzmann, SPIN Website, `http://spinroot.com/`.

[49] Sun Microsystems, Inc., JSR 308: Annotations on Java Types, `http://www.jcp.org/en/jsr/detail?id=308`.

[50] Mathias Ricken, LAPT-javac, `http://www.cs.rice.edu/~mgricken/research/laptjavac/index.shtml`.

[51] Sun Microsystems, Inc., Java Virtual Machine Specification, `http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html`.

[52] Sun Microsystems, Inc., Java Virtual Machine Specification, Changes for JDK 1.5, `http://java.sun.com/docs/books/jvms/second_edition/jvms-clarify.html`.

Appendix A

Pre-Defined Predicate Annotations

The library associated with the invariant checker provides a rich assortment of pre-written annotations. The tables in this appendix provide a brief overview. All of the predicate annotations can be found in the package `edu.rice.cs.cunit.threadCheck.predicates`.

Table A.1 : Predicate Annotation Library: `AnySynchronized*`

- `@AnySynchronizedThis`:
  Method only allowed to run if some thread, regardless which one, owns the `this` lock.

- `@AnySynchronizedField(Class fieldClass, String fieldName)`:
  Method only allowed to run if some thread, regardless which one, owns the lock specified by class and field name.

- `@AnySynchronizedField.Any(AnySynchronizedField[] value)`:
  Method only allowed to run if some thread, regardless which one, owns at least one lock of the fields specified.

- `@AnySynchronizedField.All(AnySynchronizedField[] value)`:
  Method only allowed to run if the locks of all fields specified are owned by some thread, regardless which one.

- `@AnySynchronizedField.ByName(`
  `String fieldClassName, String fieldName)`:
  Method only allowed to run if some thread, regardless which one, owns the lock specified by class name and field name.

- `@AnySynchronizedField.ByName.Any(`
  `AnySynchronizedField.ByName[] value)`:
  Method only allowed to run if some thread, regardless which one, owns at least one lock of the fields specified.

- `@AnySynchronizedField.ByName.All(`
  `AnySynchronizedField.ByName[] value)`:
  Method only allowed to run if the locks of all fields specified are owned by some thread, regardless which one.

- `@AnySynchronizedArgument(int value)`:
  Method only allowed to run if some thread, regardless which one, owns the lock of method argument specified by the zero-based index.

- `@AnySynchronizedArgument.Any(AnySynchronizedArgument[] value)`:
  Method only allowed to run if some thread, regardless which one, owns at least one lock of the arguments specified.

- `@AnySynchronizedArgument.All(AnySynchronizedArgument[] value)`:
  Method only allowed to run if the locks of all arguments specified are owned by some thread, regardless which one.

Table A.2 : Predicate Annotation Library: `NoneSynchronized*`

- `@NoneSynchronizedThis`:
  Method only allowed to run if no thread, regardless which one, owns the `this` lock.
- `@NoneSynchronizedField(Class fieldClass, String fieldName)`:
  Method only allowed to run if no thread, regardless which one, owns the lock specified by class and field name.
- `@NoneSynchronizedField.None(NoneSynchronizedField[] value)`:
  Method only allowed to run if no thread, regardless which one, owns any of the locks of the fields specified.
- `@NoneSynchronizedField.ByName(`
  `String fieldClassName, String fieldName)`:
  Method only allowed to run if no thread, regardless which one, owns the lock specified by class name and field name.
- `@NoneSynchronizedField.ByName.None(`
  `NoneSynchronizedField.ByName[] value)`:
  Method only allowed to run if no thread, regardless which one, owns any of the locks of the fields specified.
- `@NoneSynchronizedArgument(int value)`:
  Method only allowed to run if no thread, regardless which one, owns the lock of method argument specified by the zero-based index.
- `@NoneSynchronizedArgument.None(NoneSynchronizedArgument[] value)`:
  Method only allowed to run if no thread, regardless which one, owns any of the locks of the arguments specified.

Table A.3 : Predicate Annotation Library: `OnlySynchronized*`

- `@OnlySynchronizedThis`:
  Method only allowed to run if the current thread owns the `this` lock.
- `@OnlySynchronizedField(Class fieldClass, String fieldName)`:
  Method only allowed to run if the current thread owns the lock specified by class and field name.
- `@OnlySynchronizedField.Any(OnlySynchronizedField[] value)`:
  Method only allowed to run if the current thread owns at least one lock of the fields specified.
- `@OnlySynchronizedField.All(OnlySynchronizedField[] value)`:
  Method only allowed to run if the locks of all fields specified are owned by the current thread
- `@OnlySynchronizedField.ByName(`
  `Class fieldClassName, String fieldName)`:
  Method only allowed to run if the current thread owns the lock specified by class name and field name.
- `@OnlySynchronizedField.ByName.Any(`
  `OnlySynchronizedField.ByName[] value)`:
  Method only allowed to run if the current thread owns at least one lock of the fields specified.
- `@OnlySynchronizedField.ByName.All(`
  `OnlySynchronizedField.ByName[] value)`:
  Method only allowed to run if the locks of all fields specified are owned by the current thread.

- `@OnlySynchronizedArgument(int value)`:
  Method only allowed to run if the current thread owns the lock of method argument specified by the zero-based index.

- `@OnlySynchronizedArgument.Any(OnlySynchronizedArgument[] value)`:
  Method only allowed to run if the current thread owns at least one lock of the arguments specified.

- `@OnlySynchronizedArgument.All(OnlySynchronizedArgument[] value)`:
  Method only allowed to run if the locks of all arguments specified are owned by the current thread.

Table A.4 : Predicate Annotation Library: `NotSynchronized*`

- `@NotSynchronizedThis`:
  Method only allowed to run if the current thread does not own the `this` lock.

- `@NotSynchronizedField(Class fieldClass, String fieldName)`:
  Method only allowed to run if the current thread does not own the lock specified by class and field name.

- `@NotSynchronizedField.None(NotSynchronizedField[] value)`:
  Method only allowed to run if the current thread does not own any of the locks of the fields specified.

- `@NotSynchronizedField.ByName(`
  `Class fieldClassName, String fieldName)`:
  Method only allowed to run if the current thread does not own the lock specified by class name and field name.

- `@NotSynchronizedField.ByName.None(`
  `NotSynchronizedField.ByName[] value)`:
  Method only allowed to run if the current thread does not own any of the locks of the fields specified.

- `@NotSynchronizedArgument(int value)`:
  Method only allowed to run if the current thread does not own the lock of method argument specified by the zero-based index.

- `@NotSynchronizedArgument.None(NotSynchronizedArgument[] value)`:
  Method only allowed to run if the current thread does not own any of the locks of the arguments specified.

Table A.5 : Predicate Annotation Library: `*EventThread`

- `@OnlyEventThread`:
  Method only allowed to run if the current thread is the AWT event thread.

- `@OnlyEventThread.AfterRealized`:
  Method only allowed to run if the current thread is the AWT event thread, or by any thread, as long as `this` refers to an AWT or Swing component that has not been realized.

- `@NotEventThread`:
  Method only allowed to run if the current thread is not the AWT event thread.

Table A.6 : Predicate Annotation Library: `OnlyThreadWithName*`

- `@OnlyThreadWithName(String value, boolean regex)`:
  Method only allowed to run if the current thread has the specified name. If `regex` is `true` (default: `false`), then the name is treated as a regular expression.

- `@OnlyThreadWithName.Any(OnlyThreadWithName[] value)`:
  Method only allowed to run if the current thread has one of the specified names.

- `@OnlyThreadWithName.InField(Class fieldClass, String fieldName)`:
  Method only allowed to run if the current thread has the name stored in the field specified by class and field name.

- `@OnlyThreadWithName.InField.Any(`
  `OnlyThreadWithName.InField[] value)`:
  Method only allowed to run if the current thread has the name stored in one of the specified fields.

- `@OnlyThreadWithName.InField.ByName(`
  `String fieldClassName, String fieldName)`:
  Method only allowed to run if the current thread has the name stored in the field specified by class name and field name.

- `@OnlyThreadWithName.InField.ByName.Any(`
  `OnlyThreadWithName.InField.ByName[] value)`:
  Method only allowed to run if the current thread has the name stored in one of the specified fields.

Table A.7 : Predicate Annotation Library: `NotThreadWithName*`

- `@NotThreadWithName(String value, boolean regex)`:
  Method only allowed to run if the current thread does not have the specified name. If `regex` is `true` (default: `false`), then the name is treated as a regular expression.

- `@NotThreadWithName.None(NotThreadWithName[] value)`:
  Method only allowed to run if the current thread does not have any one of the specified names.

- `@NotThreadWithName.InField(Class fieldClass, String fieldName)`:
  Method only allowed to run if the current thread does not have the name stored in the field specified by class and field name.

- `@NotThreadWithName.InField.None(`
  `NotThreadWithName.InField[] value)`:
  Method only allowed to run if the current thread does not have any of the names stored in the specified fields.

- `@NotThreadWithName.InField.ByName(`
  `String fieldClassName, String fieldName)`:
  Method only allowed to run if the current thread does not have the name stored in the field specified by class name and field name.

- `@NotThreadWithName.InField.ByName.None(`
  `NotThreadWithName.InField.ByName[] value)`:
  Method only allowed to run if the current thread does not have any of the name stored in one of the specified fields.

Table A.8 : Predicate Annotation Library: `OnlyThreadWithGroupName*`

- `@OnlyEventThreadWithGroupName(String value, boolean regex)`:
  the specified group name. If `regex` is `true` (default: `false`), then the group name is treated as a regular expression.

- `@OnlyEventThreadWithGroupName.Any(`
  `OnlyEventThreadWithGroupName[] value)`:
  Method only allowed to run if the current thread has one of the specified group names.

- `@OnlyEventThreadWithGroupName.InField(`
  `Class fieldClass, String fieldName)`:
  Method only allowed to run if the current thread has the group name stored in the field specified by class and field name.

- `@OnlyEventThreadWithGroupName.InField.Any(`
  `OnlyEventThreadWithGroupName.InField[] value)`:
  Method only allowed to run if the current thread has the group name stored in one of the specified fields.

- `@OnlyEventThreadWithGroupName.InField.ByName(`
  `String fieldClassName, String fieldName)`:
  Method only allowed to run if the current thread has the group name stored in the field specified by class name and field name.

- `@OnlyEventThreadWithGroupName.InField.ByName.Any(`
  `OnlyEventThreadWithGroupName.InField.ByName[] value)`:
  Method only allowed to run if the current thread has the group name stored in one of the specified fields.

Table A.9 : Predicate Annotation Library: `NotThreadWithGroupName*`

- `@NotEventThreadWithGroupName(String value, boolean regex)`:
  not have the specified group name. If `regex` is `true` (default: `false`), then the group name is treated as a regular expression.
- `@NotEventThreadWithGroupName.None(`
  `NotEventThreadWithGroupName[] value)`:
  Method only allowed to run if the current thread does not have any one of the specified group names.
- `@NotEventThreadWithGroupName.InField(`
  `Class fieldClass, String fieldName)`:
  Method only allowed to run if the current thread does not have the group name stored in the field specified by class and field name.
- `@NotEventThreadWithGroupName.InField.None(`
  `NotEventThreadWithGroupName.InField[] value)`:
  Method only allowed to run if the current thread does not have any of the group names stored in the specified fields.
- `@NotEventThreadWithGroupName.InField.ByName(`
  `String fieldClassName, String fieldName)`:
  Method only allowed to run if the current thread does not have the group name stored in the field specified by class name and field name.
- `@NotEventThreadWithGroupName.InField.ByName.None(`
  `NotEventThreadWithGroupName.InField.ByName[] value)`:
  Method only allowed to run if the current thread does not have any of the group name stored in one of the specified fields.

Table A.10 : Predicate Annotation Library: Miscellaneous

- `@OnlyNullArgument(int value)`:
  Method only allowed to run if the argument specified by the zero-based index is `null`.

- `@OnlyNullArgument.Any(OnlyNullArgument[] value)`:
  Method only allowed to run if at least one of the argument specified is `null`.

- `@OnlyNullArgument.All(OnlyNullArgument[] value)`:
  Method only allowed to run if all of the argument specified are `null`.

- `@NotNullArgument(int value)`:
  Method only allowed to run if the argument specified by the zero-based index is not `null`.

- `@NotNullArgument.None(NotNullArgument[] value)`:
  Method only allowed to run if none of the argument specified are `null`.

- `@SameArgument(int[] value)`:
  Method only allowed to run if all the arguments specified by the zero-based indices are equal to each other.

- `@SameArgument.Any(SameArgument[] value)`:
  Method only allowed to run if at least one of the arrays of arguments specified is equal.

- `@SameArgument.All(SameArgument[] value)`:
  Method only allowed to run if all of the arrays of arguments specified are equal.

- `@SameArgument.None(SameArgument[] value)`:
  Method only allowed to run if none of the arrays of arguments specified are equal.

- `@DistinctArgument(int[] value)`:
  Method only allowed to run if none the arguments specified by the zero-based indices are not equal to each other.

- `@DistinctArgument.Any(DistinctArgument[] value)`:
  Method only allowed to run if at least one of the arrays of arguments specified does not contain equal values.

- `@DistinctArgument.All(DistinctArgument[] value)`:
  Method only allowed to run if all of the arrays of arguments specified does not contain equal values.

- `@DistinctArgument.None(DistinctArgument[] value)`:
  Method only allowed to run if none of the arrays of arguments specified does not contain equal values.

- `@NumberBoundedArgument(Mode mode, int index,double bound, double upperBound)`:
  Method only allowed to run if the argument specified by the zero-based index is a `Number` and in the specified bounds. `mode` can be one of `LESS`, `LESS_EQ`, `GREATER`, `GREATER_EQ`, `IN_EQ`, `IN`, `OUT_EQ`, and `OUT`. Except for the last four, ony the `bound` parameter is used. For the last four, `bound` specifies the lower bound and `upperBound` the upper bound. `IN_EQ` means "inside the bounds, inclusive", `IN` means "strictly inside the bounds, excluding the boundaries". `OUT_EQ` and `OUT` mean the "outside the bounds, inclusive" and "outside the bounds, excluding the boundaries", respectively.

- `@NumberBoundedArgument.Any(NumberBoundedArgument[] value)`:
  Method only allowed to run if at least one of the arguments is within its given bounds.

- `@NumberBoundedArgument.All(NumberBoundedArgument[] value)`:
  Method only allowed to run if all of the arguments is within its given bounds.

- `@NumberBoundedArgument.None(NumberBoundedArgument[] value)`:
  Method only allowed to run if none of the arguments is within its given bounds.

Appendix B

XML Annotation Syntax

B.1 Class-Based XML Format

The class-based XML format was developed first and uses a representation that is very natural for the instrumentation strategies that rewrite class files. It contains a header, a middle section representing packages and classes, and a footer. The header in figure B.1 designates the file as XML file and opens the `<concutest>` and `<threadcheck>` nodes. The footer in figure B.2 closes those two nodes. There may only be one `<threadcheck>` node per XML file. Class-based data can be mixed with invariant-based data in the same XML file, though.

```
<?xml version="1.0" encoding="UTF-8"?>    1
<concutest>                               2
  <threadcheck>                           3
```

Listing B.1: Class-Based XML Header

```
  </threadcheck>    1
</concutest>        2
```

Listing B.2: Class-Based XML Footer

In the middle section, packages, classes and methods are represented by nested nodes. For example, to specify the `java.lang.Integer` class, the following nested nodes in figure B.3. The `<class>` node is necessary to designate `Integer` as a class and make sure it is not interpreted as package name; it cannot be misinterpreted as an identifier since `class` is a reserved word in Java. Inside the `<class>` node, the developer can list invariants (see B.3 below) and methods, which in turn can list invariants. To specify the `String Integer.toString()` method, the nodes shown in figure B.4 are used.

```
...
  <java>
    <lang>
      <Integer>
        <class>
          <!-- invariants and methods -->
        </class>
      </Integer>
    </lang>
  </java>
...
```

Listing B.3: Class-Based XML Body: Class

```
    ...
      <Integer>
        <class>
          <!-- invariants and methods -->
          <method sig="toString()Ljava/lang/String;">
            <!-- invariants -->
          </method>
        </class>
      </Integer>
    ...
```

Listing B.4: Class-Based XML Body: Methods

B.2 Invariant-Based XML Format

The invariant-based XML format approaches the specification of invariants for classes and methods from a different angle: It specifies the particular invariant first, and then lists all methods which must adhere to that invariant. Just like the class-based format, invariant-based XML files contain a header, a middle section representing invariants and their methods, and a footer. The header in figure B.5 designates the file as XML file and opens the `<concutest>` node. The footer in figure B.6 closes that node.

```
<?xml version="1.0" encoding="UTF-8"?>
<concutest>
```

Listing B.5: Invariant-Based XML Header

```
</concutest>
```

Listing B.6: Invariant-Based XML Footer

In the middle section, invariants and methods are listed, nested inside `<threaddesc:def>` nodes. For example, to specify an invariant for the `String Integer.toString()` method, the nodes shown in figure B.7 are used. There can be an arbitrary number of `<threaddesc:def>` nodes in an XML file. Additionally, they can be mixed with class-based data in the same file.

```
...
  <threadcheck:def>
    <invariant>
      <!-- invariant -->
    </invariant>
    <method name="java.lang.Integer"
            sig="toString()Ljava/lang/String;"/>
    <!-- more methods -->
  </threadcheck:def>
...
```

Listing B.7: Invariant-Based XML Body: Method

B.3 XML Invariant Notation

In the appropriate places in XML files, XML nodes representing invariants may be inserted: For class-based XML files, they may appear either inside `<class>` or `<method>` nodes; for invariant-based XML files, they may appear only inside `<invariant>` nodes.

The following nodes specify primitive invariants:

- `<eventThread type="only"/>`: Only the event thread may execute the method.
- `<eventThread type="only_after_realized"/>`: Only the event thread may execute the method after it has been realized; before that, any thread may execute the method.
- `<eventThread type="no"/>`: The event thread may not execute the method.
- `<name type="not" value="`*threadname*`"/>`: The method may not be executed by threads named *threadname*.

- `<name type="only" value="`*threadname*`"/>`: The method may be executed by threads named *threadname*. Since primitive "only" annotations are treated as alternatives, the method may be executed if at least one of the "only" invariants is met.

- `<group type="not" value="`*threadgroupname*`"/>`: The method may not be executed by threads named *threadgroupname*.

- `<group type="only" value="`*threadgroupname*`"/>`: The method may be executed by threads named *threadgroupname*.

- `<id type="not" value="`*threadid*`"/>`: The method may not be executed by the thread with *threadid* as thread identification number.

- `<id type="only" value="`*threadid*`"/>`: The method may not be executed by the thread with *threadid* as thread identification number.

B.3.1 XML `@PredicateLink` Notation

The syntax shown in listing B.8 is required for `@PredicateLink` annotations:

```
...
  <predicate type="classname" arguments="true|false">
    <!-- <arg> and <values> nodes -->
  </predicate>
...
```

Listing B.8: `@PredicateLink` Annotation in XML

A `<predicate>` node may contain at most one `<values>` node and an arbitrary number of `<arg>` nodes. They both represent data from inside an annotation: `<arg>` nodes contain only the data that was actually specified in the annotation at the usage site. `<values>` nodes, on the other hand, contain more `<arg>` nodes that include default values as well. Take the `@OnlyThreadWithName` annotation, for example, which has a default value for the `regex` member. If it is used as `@OnlyThreadWithName("foo")`, then nested inside the `<values>` node, the XML file will specify both

the thread name, `value=="foo"` and the default value `regex==false`; however, there will only be one `<arg>` node for the `value` member since the value of `regex` was not specified.

The `arguments` attribute determines whether the method arguments should be passed to the predicate method.

B.3.2 XML `@Combine` Notation

The syntax shown in listing B.9 is required for `@CombineLink` annotations:

```
...                                                       1
  <predicate type="classname" arguments="true|false">     2
    <!-- <arg> and <values> nodes -->                     3
  </predicate>                                            4
...                                                       5
```

Listing B.9: `@Combine` Annotation in XML

The meaning of `<values>` and `<arg>` nodes nested inside a `<combine>` node is exactly the same as for `<predicate>` nodes. Additionally, however, the `values` node has an attribute `mode` that can be set to `"AND"`, `"OR"`, `"XOR"`, `"NOT"` and `"IMPLIES"`.

The `arguments` attribute determines whether the method arguments should be passed to the predicate method.

B.3.3 XML Argument Notation

The format of an `<arg>`, except if it is describing an array, is shown in figure B.10.

```
...                                                       1
  <arg name="membername" type="membertype"                2
       value="membervalue/>                               3
...                                                       4
```

Listing B.10: Non-Array `<arg>` XML Node

membername corresponds to the name of the member in the annotation. *membervalue* contains the data in the member specified by *membername*. *membertype*, finally, is a one-letter code that describes the type of data. Possible values for *membertype* are shown in table B.1.

Code	Description	Code	Descrption
B	byte	S	short
C	char	Z	boolean
D	double	s	java.lang.String
F	float	e	enumeration
I	int	c	class object
J	long	[	array
@	annotation		

Table B.1 : XML @PredicateLink Member Type Codes

If the member in the annotation is an array, then *membertype* contains an open square bracket [as code. For arrays, *membervalue* contains the array size, and exactly that many <element> nodes are nested inside the <arg> node, as shown in listing B.11.

```
...                                                                1
  <arg name="membername" type="membertype"                         2
       value="membervalue desc="arraytype">                        3
    <element type="elementtype" value="elementvalue"/>             4
  </arg>                                                           5
...                                                                6
```

Listing B.11: Array `<arg>` XML Node

Here, *arraytype* contains the full type of the array. An array of `String`s would, for example, have `desc="[Ljava/lang/String;"`. *elementtype* again contains a oneletter code of the kind of data stored in this element, and *elementvalue* contains the value.

In case *elementtype* is `[` (square bracket, i.e. array) or `@` (at sign, i.e. annotation), then the `<element>` tag may have nested `<arg>`, `<value>` or `<element>` nodes nested inside.

Appendix C

Suggestions for Improving Java Annotations

While working with Java annotations, we have discovered several shortcomings in the way the annotations are implemented in the Java compiler, the Java runtime, and the Java Language Specification [35]:

- Local variables may be annotated, but the annotations are ignored.
- It is illegal to use the same annotation more than once per target.
- There is no subtyping for annotations.
- Generics are not allowed for annotations.

These shortcomings are described in greater detail in the sections below.

It should be noted that there is a Java Specification Request, JSR 308 [49], that attempts to improve Java annotations. It still is in an early stage.

C.1 Annotations of Data

Currently, the annotations are limited to classes, methods and constructors and the invariants are only checked at the beginning of a method or constructor. It would be useful to extend annotations so they can also describe data: Fields, parameters, and local variables.

If these parts of a Java program were annotated, then concurrency invariants could be checked every time a piece of data is accessed. On the extreme side, this would allow a programmer to re-create a full lock-set algorithm, such as the one explained in [14], to ensure that a proper locking discipline is maintained. Most likely, though,

a much leaner, more efficient set of invariants could be created to achieve the same effect.

According to the grammar found in the Java Language Specification, 3rd Edition [35] in chapter 18.1, annotations can be attached to other annotations, types (classes and interfaces), packages, constructors, methods, fields, parameters, and local variables. Unfortunately, the utilities and APIs part of Java 5.0 and 6.0 do not support annotations on local variables: They are accepted as valid Java code, but they are not accessible at either the source, class or runtime retention level.

When studying the possibilities of annotating data for checking concurrency invariants, we created a modified version of the javac compiler, LAPT-javac [50], that at least supports local variable annotations at the class level, which is sufficient for the way the invariant checker operates when using bytecode rewriting. apt, the Annotation Processing Tool, which operates at source level, can probably be modified just as easily. Supporting local variable annotations at runtime, necessary for the reflection-based invariant checker, is considerably more difficult.

With the abilities Java 5.0 or higher provides, concurrency invariant checking for fields and parameters could be done, although our software currently does not support this. Some simple data checks can still be performed using a hack: The data is passed to an empty dummy method, which has been annotated to perform the desired check on the argument. Listing C.1 shows an example of calling such a dummy method to check data. The method checks whether the object passed to it equals the string `"foo"`. The necessary annotation is shown in listing C.2, the predicate in listing C.3, and the invariant checker's output in listing C.4.

```java
public class DummyDataCheck {
  @CheckDummyData
  public static void dummy(Object o) { }
  public static void main(String[] args) {
    String s = "foo"; // local variable
    dummy(s); // perform check, succeeds
    s = "bar";
    dummy(s); // fails
  }
}
```

Listing C.1: Using a Dummy Method to Check Data

```java
@PredicateLink(value = CheckDummyDataPredicate.class,
               arguments = true)
public @interface CheckDummyData { }
```

Listing C.2: Annotation for Listing C.1

C.2 Repeated Annotations

In Java 5.0 and 6.0 it is illegal to attach the same annotation more than once to a target. Allowing repeated annotations would make specifying several similar invariants much easier. Listing C.5 shows the most intuitive way to specify that a method may not be executed by threads named `"foo"` or `"bar"`, but the Java compiler will reject the repeated annotation.

The invariant checker library provides two remedies to this problem: Listing C.6 shows how the desired invariant from listing C.5 can be achieved using regular ex-

```java
public class CheckDummyDataPredicate {
  public static boolean check(Object this0,
                              Object[] args) {
    if (args.length!=1) return false;
    return args[0].equals("foo");
  }
}
```

Listing C.3: Predicate for Listing C.2

```
Thread Predicate Violation: (2 checks, 1 violation)                1
        Current thread 'main', id 1, group 'main'                  2
        Violated predicate @CheckDummyData                         3
        Method arguments                                           4
                'bar' : java.lang.String                           5
        at DummyDataCheck.dummy (DummyDataCheck.java:3)            6
        at DummyDataCheck.main (DummyDataCheck.java:8)             7
```

Listing C.4: Invariant Checker Output for Listing C.1

```
@NotThreadWithName("foo")                                          1
@NotThreadWithName("bar")                                          2
void someMethod() { /* ... */ }                                    3
```

Listing C.5: Repeated Annotations

pressions, and C.7 does the same using an array of annotations. Specifying multiple invariants that use the same annotation is therefore possible, but not as convenient as it could be.

After studying the Java Language Specification [35] and the Java Virtual Machine Specification [51, 52], we do not see a compelling reason against allowing repeated annotations. To allow more than one annotation per type, only the `getAnnotation` method in `java.lang.reflect.AccessibleObject` would have to be changed from its current form

```
public <T extends Annotation> T
  getAnnotation(Class<T> annotationClass)
```

to one returning an array:

```
public <T extends Annotation> T[]
  getAnnotation(Class<T> annotationClass)
```

To maintain backward compatibility, the method returning an array could be given a different name, `getAnnotations`, and the semantics of the original `getAnnotation`

```
@NotThreadWithName(value="(foo)|(bar)", regex=true)       1
void someMethod() { /* ... */ }                            2
```

Listing C.6: Regular Expressions as Alternative

```
@NotThreadWithName.None({                                  1
  @NotThreadWithName("foo"),                               2
  @NotThreadWithName("bar")})                              3
void someMethod() { /* ... */ }                            4
```

Listing C.7: Annotation Array as Alternative

changed to return the first annotation if more than one exists. Allowing repeated annotations could have made `@Combine`-style predicate annotations much less verbose.

C.3 Subtyping for Annotations

In Java 5.0 and 6.0, an annotation cannot extend another annotation, even though annotations are handled in a very similar way as interfaces. It could often be useful to extend one annotation and add additional elements, as listing C.8 shows.

The `extends` clause is not allowed in annotation declarations, even though allowing it could often increase code reuse. The most unfortunate result of the lack of subtyping for annotations is that two annotations do not have a common base class and therefore cannot be treated abstractly. All annotations implement the interface

```
@interface Named {                                         1
  String value();                                          2
  boolean regex() default false;                           3
}                                                          4
@interface NotThreadWithName extends Named { }             5
@interface NotThreadWithGroupName extends Named { }        6
```

Listing C.8: Extending Annotations

```
@Combine(Mode.AND)
@interface And extends ThreadCheckerAnnotation {
  ThreadCheckerAnnotation[] value();
}
@Combine(Mode.OR)
@interface Or extends ThreadCheckerAnnotation {
  ThreadCheckerAnnotation[] value();
}
@Combine(Mode.NOT)
@interface Not extends ThreadCheckerAnnotation {
  ThreadCheckerAnnotation value();
}
```

Listing C.9: Subtyping for Annotations

```
// common superclass
@interface ThreadCheckerAnnotation { }
// simple ThreadChecker annotations
@interface Named extends ThreadCheckerAnnotation {
  String value();
  boolean regex() default false;
}
@interface NotThreadWithName extends Named { }
@interface NotThreadWithGroupName extends Named { }
@interface NotThreadWithID extends ThreadCheckerAnnotation {
  int value();
}
```

Listing C.10: Annotation Definitions for Listing C.9

`java.lang.annotation.Annotation`, but that interface itself is not an annotation and therefore cannot be used in an annotation.

If subtyping were allowed for annotations, we could have written very simple annotations that perform the Boolean operations “and”, “or”, “xor”, “not” and “implies” to completely replace the current `@Combine`-style annotations. Listing C.9 shows how simple “and”, “or” and “not” could have been implemented if annotations had an `extends` clause and supported subtyping. Listings C.10 and C.11 contain a few annotation declarations and a usage example. Since the Boolean operators are invariant checker annotations themselves, they could have been nested arbitrarily as well.

```
@And({
  @Or({
    @NotThreadWithName("foo"),
    @NotThreadWithName("bar")}),
  @Not(@NotThreadWithGroupName("main")),
  @NotThreadWithID(5)})
void someMethod() { /* ... */ }
```

Listing C.11: Usage Example for Listing C.9

C.4 Generics for Annotations

Just like subtyping, generics are not allowed for annotations in Java 5.0 and 6.0. To a certain degree, this is understandable, because annotations are limited to a small subset of data types (primitive data, strings, enumerations, class objects, other annotations, and arrays of these types). It would nonetheless be convenient if generics could at least be used for those types. While inferior to the subtyping solution presented above, generic annotations also would have allowed us to write a concise solution for Boolean operations, as shown in listings C.12 and **??**.

```
@Combine(Mode.AND)
@interface And<T> {
  T[] value();
}
@Combine(Mode.OR)
@interface Or<T> {
  T[] value();
}
@Combine(Mode.NOT)
@interface Not<T> {
  T[] value();
}
```

Listing C.12: Generics for Annotations