Diss. ETH No. 16020

Combining Static and Dynamic Analysis to Find Multi-threading Faults Beyond Data Races

A dissertation submitted to the
Swiss Federal Institute of Technology Zurich
ETH Zürich

for the degree of
Doctor of Technical Sciences

presented by
Cyrille Artho
Dipl. Informatik-Ing. ETH
born June 17, 1976
citizen of St. Gallenkappel SG,
Switzerland

accepted on the recommendation of
Prof. Dr. Armin Biere, examiner
Prof. Thomas Gross, co-examiner
Dr. Klaus Havelund, co-examiner
Prof. Dr. Doron Peled, co-examiner

April 2005

To the memory of Christoph Lehmann

Circumstances, and a certain bias of mind,
have led me to take interest in such riddles,
and it may well be doubted whether human ingenuity
can construct an enigma of the kind
which human ingenuity may not, by proper application, resolve.

— *Edgar Allen Poe (1809 – 1849)*

Acknowledgements

This thesis would not have been possible without the support of many people. First and foremost, I would like to thank my advisor, Armin Biere, for his support, his collaboration on all the papers written, and for letting me work at NASA Ames during two summers. Furthermore, much of this work, especially experiments using model checking, would not have been possible without the input and work of Viktor Schuppan.

The two summer projects at Ames were an invaluable source of inspiration for me and determined the direction of my thesis. I am especially indebted to Klaus Havelund who supervised my work there, but also to Allen Goldberg and Robert Filman for sharing their ideas with me. I would also like to thank Saddek Bensalem, whom I also met at Ames, for inviting me to Verimag in 2003 and developing ideas for more future projects than could be started within the time of a single thesis.

My thanks also go to Thomas Gross and Doron Peled who agreed to be my co-advisors. I am thankful for their suggestions and feedback.

Students who worked for the JNuke project also contributed significantly to its success. Pascal Eugster, Marcel Baur, Peter Farkas, and Boris Zweimüller all contributed to vital parts of JNuke. I would also like to thank Horacio Gagliano and Raphael Ackermann for work on related projects that did not find a way into this dissertation.

Many thanks go to Christoph von Praun for kindly providing his benchmark applications and for quickly answering questions about the nature of the atomicity violations in them, and to Robert Stärk for sharing his profound knowledge of the intricacies of bytecode correctness and his input throughout the work involving bytecode simplification.

Finally I would like to thank all my co-workers at the Computer Systems Institute, who created a friendly and stimulating environment at work, and especially the secretaries, Ruth Hidalgo, Eva Ruiz, and Hanni Sommer, for all their work behind the scenes, which is often not directly visible.

Table of Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Overview . . . 2
  - 1.2.1 Concurrent Programming . . . 2
  - 1.2.2 Software Analysis Techniques . . . 4
- 1.3 Thesis Statement . . . 5
- 1.4 Outlook . . . 5

2 Background **7**
- 2.1 Terminology . . . 7
- 2.2 Analysis Techniques . . . 8
  - 2.2.1 Static Analysis . . . 8
  - 2.2.2 Dynamic Analysis . . . 9
  - 2.2.3 Comparison Between the Different Technologies . . . 11
  - 2.2.4 Possible Application Strategies . . . 13
- 2.3 Concurrent Programming in Java . . . 15
  - 2.3.1 Program data . . . 15
  - 2.3.2 Multi-threading . . . 16
  - 2.3.3 Thread synchronization . . . 17
  - 2.3.4 Lock synchronization structure . . . 18
- 2.4 Concurrency Errors in Software . . . 19
  - 2.4.1 Low-level Data Races . . . 21
  - 2.4.2 High-level Data Races . . . 23
  - 2.4.3 Atomic Sequences of Operations . . . 24
- 2.5 Summary . . . 25

3 High-level Data Races **27**
- 3.1 Motivation . . . 27
- 3.2 Informal Definition of High-level Data Races . . . 30
  - 3.2.1 Basic Definition . . . 30
  - 3.2.2 Refinement of Basic Definition . . . 31
- 3.3 Formal Definition of High-Level Data Races . . . 32
  - 3.3.1 Views . . . 33
  - 3.3.2 Views in Different Threads . . . 33
  - 3.3.3 Examples . . . 34

3.3.4 Soundness and Completeness 35
3.4 Summary 36

4 Block-local Atomicity 37
4.1 Our Data-flow-based Algorithm 37
4.2 Formalization of Our Algorithm 39
4.3 Extension to Nested Locks and Recursion 40
4.4 Precision and Limitations of Our Algorithm 41
4.4.1 Soundness and Completeness 41
4.4.2 Comparison to Previous Atomicity-based Approaches 41
4.4.3 Limitations of Atomicity-based Approaches 42
4.4.4 Serializability 43
4.5 Summary 43

5 Run-time Verification 45
5.1 Definitions 46
5.1.1 Preliminaries 46
5.1.2 Trace generation 47
5.1.3 Trace monitoring 49
5.2 Benefits and Limitations of Each Approach 50
5.2.1 Code instrumentation 50
5.2.2 Wrappers 50
5.2.3 Custom execution environment 50
5.2.4 On-the-fly vs. off-line trace monitoring 51
5.2.5 Hybrid approaches 51
5.3 Property Verification 52
5.3.1 Generic properties 52
5.3.2 Application-specific properties 53
5.3.3 Steering 53
5.3.4 Relation to testing 53
5.4 Existing Work 54
5.5 Practical Experience 54
5.5.1 Flexibility 56
5.5.2 Scalability 56
5.6 Capabilities and Limits 58
5.7 Summary 60

6 Combined Static and Dynamic Analysis 61
6.1 Background and Motivation 61
6.2 Static Analysis in JNuke 63
6.2.1 Graph-free abstract interpretation 63
6.2.2 Separation of control flow and bytecode semantics 64
6.2.3 Optimized state space management 65
6.3 Run-time verification in JNuke 66
6.4 Generic Analysis Algorithms 66

6.4.1 Context data . . . 68
6.4.2 Interfacing run-time verification . . . 71
6.4.3 Interfacing static analysis . . . 72
6.5 Summary . . . 74

7 Bytecode Inlining and Abstraction 75
7.1 Problems with Bytecode . . . 75
7.2 Java Compilation with Bytecode Subroutines . . . 76
7.2.1 Java Bytecode . . . 76
7.2.2 Exception Handlers and Finally Blocks . . . 77
7.2.3 Finally Blocks and Subroutines . . . 78
7.2.4 Nested Subroutines . . . 79
7.3 Inlining Java Subroutines . . . 80
7.3.1 Sufficient and Necessary Well-formedness Conditions . . . 82
7.3.2 Control Transfer Targets . . . 83
7.3.3 Exception Handler Splitting . . . 87
7.3.4 Exception Handler Copying . . . 88
7.3.5 Violation of Well-formedness Conditions in JDK 1.4 . . . 88
7.3.6 Costs of Inlining . . . 89
7.4 Abstract, Register-based Bytecode . . . 90
7.5 Related Work . . . 92
7.6 Summary . . . 93

8 Implementation 95
8.1 Overview of JNuke . . . 95
8.2 Observer Architecture . . . 96
8.3 VM-based Implementation: JNuke . . . 97
8.3.1 JNuke VM . . . 99
8.3.2 Run-time verification API . . . 100
8.4 Instrumentation-based Implementation: JPaX . . . 101
8.4.1 Java Bytecode Instrumentation . . . 101
8.4.2 Event Stream Format . . . 102
8.4.3 Observer Architecture . . . 103
8.5 Module Overview . . . 103
8.5.1 Description . . . 105
8.5.2 Module Dependencies . . . 105
8.6 JNuke's OO in C . . . 105
8.6.1 Memory Management . . . 107
8.6.2 Comparison to Java . . . 107
8.6.3 Type `JNukeObj` . . . 108
8.6.4 Statically Typed Method Calls . . . 111
8.6.5 Polymorphism . . . 111
8.6.6 Inheritance . . . 111
8.6.7 Constructor and Destructor . . . 111

8.6.8 Containers . . . 114
8.7 Unit tests . . . 116
8.7.1 Structure . . . 116
8.7.2 Registering Test Cases . . . 118
8.7.3 `JNukeTestEnv` . . . 118
8.7.4 Log Files . . . 118
8.7.5 Code Coverage . . . 119
8.8 Summary . . . 119

9 Experiments 125
9.1 Applications . . . 125
9.2 JNuke VM . . . 126
9.3 JNuke Model Checker . . . 128
9.4 Eraser . . . 131
9.5 High-level Data Races . . . 131
9.5.1 Java Path Explorer . . . 131
9.5.2 JNuke . . . 133
9.5.3 View Consistency as a Fault Pattern . . . 135
9.6 Block-local Atomicity . . . 135
9.6.1 Comparison to Other Atomicity-based Approaches . . . 135
9.6.2 Performance and Results of the Generic Analysis . . . 136
9.6.3 Block-local Atomicity as a Fault Pattern . . . 139
9.7 Summary . . . 140

10 Related Work 141
10.1 Data Races . . . 141
10.2 Atomicity of Operations . . . 141
10.3 Database Concurrency . . . 143
10.4 Hardware Concurrency . . . 144
10.5 Stale Values . . . 145
10.6 Escape Analysis . . . 145
10.7 Serializability . . . 145
10.8 Summary . . . 147

11 Future Work 149
11.1 Dynamic Analysis in JNuke . . . 149
11.2 Static Analysis in JNuke . . . 150
11.3 High-level Data Races . . . 151
11.4 Block-local Atomicity . . . 151
11.5 Concurrent Programming in the Future . . . 151
11.6 Summary . . . 152

12 Conclusions 153

Bibliography 155

List of Figures

2.1 Applicability and precision of each technology. 11
2.2 Expressiveness and degree of automation of each technology. 12
2.3 Computational and human power required for each technology. 13
2.4 Application scenarios for different analysis technologies. 14
2.5 A `synchronized` block. 17
2.6 Nested locking operations. 19
2.7 Generalization to non-nested locking operations. 19
2.8 A low-level data race example. 22
2.9 A high-level data race resulting from three atomic operations. 24
2.10 A non-atomic increment operation. 25
2.11 An atomicity violation that is not a high-level data race. 26

3.1 The Remote Agent Executive 28
3.2 The task/daemon synchronization inconsistency. 29
3.3 Points with x and y coordinates. 30
3.4 One updating thread and three reading threads. 31

4.1 Intuition behind our algorithm. 38
4.2 The importance of data flow analysis. 42
4.3 A correct non-atomic, non-serializable program. 43

5.1 Comparison of tool architectures. 56
5.2 Aliasing may easily hide faults during testing. 59

6.1 A new tool flow using combined static and dynamic analysis. 62
6.2 Separation of control flow and analysis algorithm. 64
6.3 Run-time verification in JNuke. 66
6.4 Classical approaches duplicate the analysis algorithm. 67
6.5 Running generic analysis algorithms. 68
6.6 Excerpt of the block-local atomicity algorithm (simplified). 70
6.7 Interfacing run-time verification with generic analysis. 72
6.8 Interfacing static analysis with a generic analysis algorithm. 72

7.1 A simple finally block, its bytecode and its control flow graph. 79
7.2 Breaking out of a subroutine to an enclosing subroutine. 80
7.3 Control flow graph of nested subroutines. 81
7.4 Instruction sequences violating well-formedness conditions. 84
7.5 Inlining a subroutine. 85
7.6 Inlining a nested subroutine: Step 1. 85
7.7 Inlining a nested subroutine: Step 2. 87
7.8 Sizes of subroutines in all JRE packages of Java 1.4.1. 89
7.9 Size increase after inlining all subroutines in each method. 90

8.1 Overview of JNuke. 96
8.2 The generic observer pattern. 96
8.3 Event observers in JNuke. 97
8.4 The observer-based architecture for run-time verification. 97
8.5 Detailed view of run-time verification architecture. 98
8.6 Overview of the key components of the JNuke VM. 99
8.7 Instrumentation-based approach to run-time verification. 101
8.8 Interleaving of light-weight event entries. 102
8.9 Detailed view of instrumentation-based event observers. 104
8.10 Module dependencies. 106
8.11 Dependencies of higher-level modules. 106
8.12 JNuke's OO model. 108
8.13 The C `struct JNukeObj`. 108
8.14 Retrieving an object's instance data. 109
8.15 The type information of each object. 110
8.16 Nested inheritance. 112
8.17 Constructor of the type `JNukePair`. 113
8.18 An example for using a read-only iterator. 115
8.19 A simple test case. 117
8.20 A construct ensuring maximal coverage. 117
8.21 The struct `JNukeTestEnv`. 118
8.22 Project size in lines of code. 120
8.23 Uncovered code in percent. 121
8.24 Uncovered lines of code. 122
8.25 Number of files containing uncovered lines. 123

9.1 A false positive resulting from redundant locks. 139

10.1 A non-atomic operation that does not violate view consistency. 142
10.2 A code idiom where block-local atomicity is not applicable. 146

List of Tables

3.1 Examples with two threads. 34
3.2 Examples with three threads. 35

5.1 Existing run-time verification tools. 55
5.2 Memory usage overhead for run-time verification in JNuke. 57
5.3 Run-time overhead of various tools. 57

6.1 Context differences for the static and dynamic version. 69

7.1 A subset of Java bytecode instructions. 77
7.2 Possible successors of Java bytecode instructions. 80
7.3 Number of calls per subroutine. 89
7.4 The benefits of register-based bytecode. 92

8.1 Run-time verification events in JNuke. 100
8.2 Short description of each module. 105
8.3 Replacement (wrapper) functions for memory management. 107

9.1 Benchmark programs. 126
9.2 Benchmarks to evaluate the performance of the VM. 127
9.3 Benchmark results for the JNuke model checker. 129
9.4 Comparison between JNuke and JPF. 130
9.5 Low-level data race analysis results using JNuke. 132
9.6 High-level data race analysis results using JPaX. 133
9.7 High-level data race analysis results using JNuke. 134
9.8 Block-local atomicity: comparison to other approaches. 136
9.9 Benchmark results for the block-local atomicity analysis (RV). 137
9.10 Results for block-local atomicity used in static analysis. 138

Abstract

Multi-threaded programming gives rise to errors that do not occur in sequential programs. Such errors are hard to find using traditional testing. In this context, verification of the locking and data access discipline of a program is very promising, as it finds many kinds of errors quickly, without requiring a user-defined specification.

Run-time verification utilizes such rules in order to detect possible failures, which do not have to actually occur in a given program execution. A common such failure is a data race, which results from inadequate synchronization between threads during access to shared data. Data races do not always result in a visible failure and are thus hard to detect. Traditional data races denote direct accesses to shared data. In addition to this, a new kind of high-level data race is introduced, where accesses to sets of data are not protected consistently. Such inconsistencies can lead to further failures that cannot be detected by other algorithms. Finally, data races leave other errors untouched which concern atomicity. Atomicity relates to sequences of actions that have to be executed atomically, with no other thread changing the global program state such that the outcome differs from serial execution. A data-flow-based approach is presented here, which detects stale values, where local copies of data are outdated.

The latter property can be analyzed efficiently using static analysis. In order to allow for comparison between static and dynamic analysis, a new kind of generic analysis has been implemented in the JNuke framework presented here. This generic analysis can utilize the same algorithm in both a static and dynamic setting. By abstracting differences between the two scenarios into a corresponding environment, common structures such as analysis logics and context can be used twofold. The architecture and other implementation aspects of JNuke are also described in this work.

Kurzfassung

Programme mit mehreren Threads erlauben Fehler, die nicht in sequentiellen Programmen vorkommen. Solche Fehler sind mit traditionellem Testen schwer zu finden. Darum ist die Überprüfung der Locking- und Datenzugriffsdisziplin eines Programms sehr vielversprechend, da es viele Arten von Fehlern schnell findet, ohne eine benutzerdefinierte Spezifikation zu benötigen.

Run-time Verification benutzt solche Regeln, um mögliche Fehler zu finden, ohne dass diese tatsächlich in einer gegebenen Programmausführung auftreten müssen. Ein häufiger solcher Fehler ist ein Date Race, das aus unzureichender Synchronisation zwischen mehreren Threads mit gemeinsamen Daten resultiert. Data Races resultieren nicht immer in einem sichtbaren Fehler und sind darum schwer zu entdecken. Traditionelle Data Races beziehen sich auf direkte Zugriffe zu gemeinsamen Daten. Darüberhinaus wird eine neue Art von Data Race eingeführt, wo Mengen von Daten nicht konsistent geschützt sind. Solche Inkonsistenzen können zu weiteren Fehlern führen, welche nicht durch andere Algorithmen entdeckt werden können. Schlussendlich lassen Data Races andere Fehler ausser acht, welche die Atomicity betreffen. Atomicity bezieht sich auf Sequenzen von Aktionen, die atomar ausgeführt werden müssen, ohne dass ein anderer Thread den globalen Zustand auf eine Art ändert, so dass das Ergebnis einer Operation sich von serieller Ausführung unterscheidet. Ein data-flow-basierter Ansatz wird hier präsentiert, welcher Stale Values entdeckt, wo lokale Kopien von Daten veraltet sind.

Letztere Eigenschaft kann effizient statisch analysiert werden. Um einen Vergleich zwischen statischer und dynamischer Analyse zu erlauben, wurde im JNuke-Framework, das hier präsentiert wird, eine neue Art von generischer Analyse implementiert. Diese generische Analyse kann denselben Algorithmus in einem statischen oder dynamischen Umfeld benutzen. Indem man die Unterscheide zwischen den beiden Szenarien in ein entsprechendes Environment abstrahiert, können gemeinsame Strukturen wie die Analyse-Logik und der Kontekt doppelt genutzt werden. Die Architektur und andere Implementationsaspekte von JNuke werden auch in dieser Arbeit beschrieben.

1 Introduction

Multi-threaded programming is very difficult, and can result in errors that cannot be found through testing as is the case with sequential programs. Due to its scalability, the use of general-purpose rules to verify the locking and data access discipline of a program is more promising than systematic exploration of the entire program space. It is difficult to find such rules, but once found, they can be applied to any program using only a single program trace and detect errors that cannot usually be found through testing.

JNuke contains a virtual machine for Java bytecode that can execute Java programs and monitor such general-purpose rules. Furthermore, it is capable of utilizing the same analysis algorithm statically, "at compile-time", and dynamically, at run-time. This allows for new combinations and comparisons of the two techniques. JNuke also offers model-checking capabilities, in order to explore the full program state space when run-time verification is not sufficient.

1.1 Motivation

Java [GJSB00] is a very popular programming language. Its built-in support for multi-threaded programming has been a major factor that made concurrent programming popular [Lea99]. Multi-threading creates a potential for new kinds of errors that cannot occur in traditional, sequential programs. Because the thread schedule determines the order in which thread actions occur, program behavior becomes non-deterministic. Testing becomes very ineffective because a test run only covers a single program schedule and therefore only a small part of the entire potential behavior. Many faults therefore cannot be found through testing.

Two very common kinds of such faults are *deadlocks* and *data races.* In a deadlock, several threads mutually wait for another thread to perform a certain action. Because of circular dependencies among these actions, one thread cannot progress without the other thread releasing a resource, and vice versa. Thus both threads are stuck in a "deadly embrace" [Dij68]. A data race denotes a scenario where several threads perform an action without adequate synchronization between threads. If these actions occur concurrently, the result differs from serial execution of these actions. Unlike deadlocks, data races are hard to detect because they do not always result in a visible error (such as a halted program).

Therefore, simple testing has to be augmented with more refined checks. The idea of the Eraser algorithm [SBN$^+$97] is to monitor the *lock set,* the set of locks held, during each memory access of any thread. If the intersection of all these lock sets ever becomes empty, this means that no common lock protects that data in memory, and a data race is possible. Eraser embodies a *fault pattern,* which is an undesired program behavior that can result in a run-time failure. The fact that the fault pattern is much easier to detect than the actual failure and that it is applicable to any program makes it a valuable contribution to software correctness. Furthermore, it requires no user-defined specification describing potential outcomes of program runs using corrupted data.

The Eraser algorithm proved to be very useful in practice; however, it only covers data races at the level of individual data accesses. Such a data race will be denoted *low-level data race* here. *High-level data races,* on the other hand, concern accesses to sets of fields where a given set is not always accessed consistently [AHB03]. This can lead to further errors that are not detected by other algorithms. Like low-level data races, high-level data races can be found effectively at run-time, requiring only a single execution trace.

Finally, data races leave other errors untouched which are concerned with *atomicity.* Atomicity denotes the fact that a certain block of code should be executed atomically, with no other thread changing the global program state in a way such that the outcome differs from a serial execution. Several approaches to atomicity exist; the approach presented here, *block-local atomicity* [ABH04], is data-flow based and detects errors where a local copy of a value is outdated, or *stale* [BL02]. Because that property is thread-local, static analysis can check it very effectively.

Experiments shown in this thesis indicate that method-local properties are better suited to static analysis while global properties can be checked more efficiently at run-time. However, so far, no tool has been available to compare the two approaches directly. JNuke is the first tool that can utilize the same algorithm for both static and dynamic analysis [AB05a] and therefore allows comparing algorithms which are implemented using this generic framework. Beyond that, JNuke is also able to perform run-time verification and software model checking for the full Java bytecode instruction set. It can therefore verify any Java program, as long as all library calls used are implemented.

1.2 Overview

1.2.1 Concurrent Programming

Concurrent programming is a cornerstone of modern software development, allowing systems to perform multiple tasks at the same time [Tan92]. In recent years, the use of multi-threaded software has become increasingly widespread. Especially for large servers, multi-threaded programs have advantages over multi-process programs: Threads are computationally less expensive to create than processes, and share the same address space.

Threads are often used in a client-server environment, where each thread serves a request. Typical applications are servers [AB01] or software frameworks for ubiquitous computing where interaction occurs in a distributed and highly concurrent manner [BM04]. Java [GJSB00] is a programming language that is widely used for such applications.

Java is an object-oriented programming language that has enjoyed much success in the past few years. Source code is not compiled to machine code, but to a different form, the *bytecode.* This bytecode runs in a dedicated environment, the *virtual machine* (VM). In order to guarantee the integrity of the system, each class file containing bytecode is checked prior to execution [GJSB00, LY99, Sun04b].

The Java language and its base libraries support multi-threaded or concurrent programming. Threads can be created, run, and suspended. For thread communication and mutual exclusion, locks are used, using the `synchronized` keyword, offering monitor-based programming. Locks can also be used for shared conditionals using `wait` and `notify` [Lea99].

The Java language allows each object to have any number of *fields,* which are attributes of each object. These may be static, i.e., shared among all instances of a certain class, or dynamic, i.e., each instance has its own fields. Static fields are always globally visible: They can be accessed by any thread throughout program execution. Dynamic fields are only accessible through their instance reference. If an instance reference is accessible to several threads, its fields are *shared* and can be accessed concurrently. In contrast to that, *local variables* are thread-local and only visible within one method.

Some programming languages limit possible concurrency errors by offering only limited multi-threading constructs, where thread interaction is structured by high-level mechanisms such as rendez-vous [Bar97], communication channels in conjunction with non-deterministic guards [Hoa83, Hol91], or semaphores [Dij68]. Most programming languages today, though, have the same multi-threading mechanisms that Java offers and feature explicit thread creation and locking mechanisms that allow developers to control program behavior precisely. In most operating systems and programming languages available today, POSIX threads [But97] offer such functionality. These constructs, while very versatile, also make it easy to introduce faults in the program. Such faults are the subject of interest in this work. This thesis investigates multi-threaded programs only, where threads share a common address space; it does not investigate software running as separate processes, even though other mechanisms are available to share memory [TRY$^+$87], or distributed systems which can run on several computers [HV99].

Multi-threaded programming requires a developer to protect a program against uncontrolled interference between threads. To achieve this, shared memory can be protected by locks in order to prevent uncontrolled concurrent access. However, incorrect lock usage using too many locks may lead to deadlocks. For example, if two threads each wait on a lock held by the other thread, both threads cannot continue their execution. Therefore locks should be used sparingly.

On the other hand, if a value is accessed with insufficient lock protection, data races may occur: two threads may access the same value concurrently, and the results of the operations are no longer deterministic [SBN$^+$97]. The non-determinism results from the fact that the thread schedule is chosen by the VM and cannot be controlled by the application [LY99]. The result is that application behavior may vary depending on factors such as system load, and therefore a program that is deterministic when executed serially becomes non-deterministic in concurrent execution. This can make it extremely hard to find faults in such programs, since a fault may rarely manifest itself as a failure.

Even if no data race occurs, shared data may be used inconsistently. For example, a data structure representing a coordinate pair may have to be treated atomically. Even if individual accesses are protected by a common lock, certain operations, such as setting both coordinates to 0, must be executed without releasing that lock during the operation. Inconsistent accesses to such sets of shared data will be denoted as high-level data races [AHB03].

Data races do not cover all kinds of concurrency errors because they do not capture data flow from shared fields to local variables or stack values. A program may continue to use these local copies outside a synchronized region. Stale values denote copies of shared data where the copy is no longer synchronized. This thesis utilizes a data-flow-based technique to find stale-value errors, which are not found by low-level and high-level data race algorithms. This property, block-local atomicity, determines the necessary scope of synchronized blocks in a program that is needed to ensure the absence of stale values [ABH04].

1.2.2 Software Analysis Techniques

Much effort has gone into automated fault-finding in software, single-threaded and multi-threaded. The approaches can be separated into *static analysis,* which checks a program at compile-time and tries to approximate its run-time behavior, and *dynamic analysis,* which tries to catch and analyze anomalies during execution of a concrete program, with its full semantics and its entire range of operations.

Static analysis approximates the set of possible program states. It includes abstract interpretation [CC77], where a fix point over abstract states, which represent sets of concrete states, is calculated. Static analysis scales well for many properties, as properties may be modular and thus only require summary information of dependent methods or modules. "Classical" static analysis constructs a graph representation of the program and calculates the fix point of properties using that graph [CC77].

An alternative static approach is *theorem proving,* which uses a series of deductions in order to determine whether a given property holds for the entire program. One tool that has successfully implemented an annotation-based, automated approach to check such properties is ESC/Java [DLNS98].

Dynamic tools have the advantage of having more precise information available in the execution trace. *Run-time verification* analyzes application execution. The Eraser algorithm [SBN$^+$97], which has been implemented in the Visual Threads tool [Har00] to analyze C and C++ programs, is such an algorithm that examines a program execution trace for locking patterns and variable accesses in order to predict potential data races. The Java PathExplorer tool (JPaX) [HR01] performs deadlock analysis and data race analysis (both low-level and high-level) on Java programs. JNuke, the tool presented here, implements these algorithms, which are complemented by block-local atomicity analysis [ABH04]. It furthermore can use a given algorithm both in static analysis and run-time verification [AB05a].

More heavyweight dynamic approaches include model checking, which explores all possible schedules in a program. Recently, model checkers have been developed that apply directly to programs (instead of just models thereof). Such *software model checkers* include JNuke [ASB$^+$04], Java PathFinder (JPF) developed by NASA [VHB$^+$03], and similar systems [BPR01, CDH$^+$00, HS99]. Such systems, however, suffer from the state space explosion problem: The size of the state space is exponential in the size of the system state. Therefore techniques have to be employed to reduce the number of states that have to be explored, by analyzing independent states only, and by removing parts of the program that are not relevant to the properties checked.

1.3 Thesis Statement

> This thesis introduces two new fault patterns, high-level data races and block-local atomicity. These general-purpose rules can be applied to any multi-threaded program in order to find errors that are extremely hard to find using classical testing, while allowing for specificationless, effective checking. Both properties find errors that are not, or less effectively, found by other means.
> Moreover, this thesis shows that static and dynamic analysis can be combined by using a software architecture that takes advantage of their common structures. Such a generic environment allows for integration of static analysis and run-time verification.

This thesis introduces two new fault patterns, high-level data races and block-local atomicity. When run-time verification is used during a test run to check these patterns, finding faults in multi-threaded software is much more effective than with classical testing [BDG$^+$04]. Static analysis can even find certain errors without running tests, in a partially finished program. Nonetheless, certain multi-threading faults remain which cannot be captured by such fault patterns. These can be found by exploring the entire program behavior using software model checking. This thesis shows that both an effective state-space exploration heuristics and an efficient implementation are crucial for the scalability of this technology.

1.4 Outlook

Chapter 2 introduces the terminology and technologies referred to in this thesis. The new concepts, high-level data races and block-local atomicity, are presented in Chapters 3 and 4. Chapter 5 gives a detailed overview of run-time verification, the key technology used in this work, while Chapter 6 shows a novel concept that allows combining run-time verification with abstract interpretation. Technical aspects of this thesis, bytecode simplification through inlining and abstraction, and architectural information about the implementation, are presented in Chapters 7 and 8. Experimental results are described in Chapter 9. Related work is discussed in Chapter 10 while Chapter 11 shows possible areas for future work. Chapter 12 concludes.

2 Background

2.1 Terminology

In software, the term *fault* denotes an incorrect implementation, introduced by a human *error* during development. A fault can eventually lead to a *failure,* i.e., incorrect behavior during program execution [IEE83]. Software testing is an established way of detecting failures in software [Pel01]. In order to execute software tests, a developer tries to create situations *(tests)* that discover a fault in the software by choosing a representative set of inputs *(test cases).* Finding a fault requires a test case leading to a failure. Usually, test cases are written to model known or anticipated failures, which explains why tests only uncover some faults.

Certain sets of faults can be partitioned into equivalence classes, because identical underlying programming mechanisms are used. The incorrect or incomplete usage of program constructs results in a possible failure. Classes of equivalent faults therefore fulfill certain predicates or *fault patterns* which can serve to classify faults. In this thesis, fault patterns are primarily used to *detect* faults of a certain kind.

When an analysis tool is applied to a program, it will issue a number of *reports.* These reports each indicate a possible fault in the software. A report that represents a genuine fault is a *true positive,* one which cannot be confirmed as such is a *false positive* or *spurious warning.* A fault that does not result in any report is undetected and thus a *false negative.* Note that this definition corresponds to classical mathematical sciences and medicine, but is sometimes reversed in computer science literature.

A proof procedure for a logic is *sound* if it proves only valid formulae. Possible faults in the argument therefore always result in the absence of a proof. Hence a sound tool does not miss any faults. A correct property may still not be provable in such a system. Such a case corresponds to false positives reported.

In logic, a calculus is *complete* iff for any statement P, there exists a proof for either P or $\neg P$. A system is *consistent* if there are no contradictions and a proof for both P and $\neg P$ never exists. For an open class of undecidable problems, it is impossible to develop a formal system that is both consistent and complete [Göd31].

Proof theory uses a different definition of completeness. A formal calculus is complete if all tautologies can be proven. A complete program prover can therefore always prove a correct program as such, but may miss faults. A prover that is both sound and complete cannot exist because this would solve the Halting Problem, which has been proven to be impossible by the Church-Turing thesis [Chu36, Tur37]. In practice, sound tools are often preferred because once the number of reports is zero, there are no faults left in the software w.r.t. the properties investigated. Fine-tuning such a tool for a specific domain can even reduce false positives to zero [BCC+02].

A *suppression list* is a list of program parts (methods, classes, or packages) that are to be excluded from analysis. Such parts often have statically been proved safe, either by previous manual or automatic analysis. Sometimes they are simply excluded because the code in question is not of interest.

The term *design pattern* denotes composition of objects in software [GHJV95]. In this thesis, a different notion of composition is also used. It includes lock patterns and sometimes only applies to a small part of the program. The term *code idiom* applies to that such a context.

An *application programming interface* (API) provides access to the externally available functions or methods of a library. The use of an API is also denoted an API call or library call.

Other terms are defined in the remainder of this chapter, and presented in the context in which they have emerged or are commonly used.

2.2 Analysis Techniques

Analysis techniques can be categorized into *static* and *dynamic analysis.* Static analysis investigates properties "at compile time", without executing the actual program. It explores a simplified version of the concrete program, examining the entire behavior. Dynamic analysis, on the other hand, actually executes the program (or *system under test,* SUT) and verifies properties on an execution trace of that SUT.

2.2.1 Static Analysis

Abstract Interpretation

The two technologies that are traditionally associated with static analysis are *abstract interpretation* and *theorem proving.* Abstract interpretation constructs a mathematical representation of the program properties and calculates a fix point of properties using that graph [CC77]. This (abstract) representation of the program often encompasses a (strictly) wider range of behaviors than the original. Such a conservative static analysis is sound: Its wider spectrum of behaviors admits any erroneous behavior and thus detects any possible violation of a given property. The downside are false positives, correct programs that cannot be proved as such. Careful tuning to a specific domain may eliminate false positives [BCC+02]. General-purpose static analyzers based on abstract interpretation may discard soundness in addition to completeness for simplicity and speed [AB01]. Abstract intepretation is used in an unconventional way in this thesis, operating on the program itself rather than a mathematical model of it. This *graph-free analysis* [Moh02] is presented in Chapter 6.

Theorem Proving

Theorem proving is a mathematical program verification and entails the proof of given properties by a series of proof transformation steps, each of which corresponds to an implication or equivalence. This rigorous mathematical approach is sound and complete, but typically involves human interaction since the general problem of proving program correctness is undecidable. Tools to support mathematicians in finding and executing proof steps exist [Abr96, ABB$^+$00, Sch00] but typically require a strong background in mathematics and considerable skill with such proofs [Pel01]. This rigorous approach usually entails adapting the entire development process to this methodology, because proofs are usually performed in stages, many of which have to be repeated when the specification or system architecture changes [Abr96, CTW99].

Theorem proving is sometimes automated and used as a part of a tool chain, the most well-known example being the *Simplify* theorem prover, which is the core of the Extended Static Checking tool for Java (ESC/Java) [DLNS98]. By giving up on soundness and completness, automation in theorem proving becomes possible. However, the following discussion will consider the typical case of theorem proving, where exact results are sought.

2.2.2 Dynamic Analysis

Dynamic analysis actually executes the system under test, covering the entire behavior, or at least the part which is relevant for the properties of interest, precisely. Compared to static analysis, dynamic tools have the advantage of having precise information available in the execution trace. However, coverage of the complete system behavior is often untractable.

Run-time Verification

Classical testing executes the SUT given manually or semi-automatically created test cases and observes its output [Mye79]. This has the drawback that execution of a faulty SUT must produce an incorrect output (or observable behavior) in order to allow a fault to be detected. Since the output of multi-threaded systems may depend on the thread schedule [Lea99], the probability that a particular schedule reveals faults in the SUT is very low [BDG$^+$04], because a test requires a particular "bad" schedule to reveal a failure.

Run-time verification tries to observe properties of the system which are not directly tested. Instead, *stronger* properties than the failure itself are checked. A stronger property is usually independent of scheduling yet a strong indicator that the failure looked for can occur under a certain schedule [ABG$^+$03]. Therefore even a "good" schedule usually allows detection of faults in the system, even if no failure occurs. An occurrence of the failure looked for almost always violates the verification property, but the reverse is not true. A violated property may be benign and never lead to a failure.

Some fully automated dynamic analysis algorithms only require a single execution trace to deduce possible errors [AHB03, SBN$^+$97]. This fact is the foundation of run-time verification [RV04], ameliorating the major weakness of testing, which is the possible dependence of system execution on non-deterministic decisions of its environment, such as the thread scheduler. Run-time verification infers possible behaviors in other program executions and can thus analyze a larger part of the possible behaviors, scaling significantly better than software model checking. Run-time verification is the cornerstone technology used in this thesis and treated more extensively in Chapter 5.

Model Checking

Model checking [CGP99, VHB$^+$03] is often counted towards static analysis methods because it tries to explore the entire behavior of a SUT by investigating each reachable system state. This classification is certainly fitting when model checking is applied to a model of a system, which describes its behavior on a more abstract level. Model checking is commonly used to verify algorithms and protocols [Hol91]. However, more recently, model checking has been applied directly to software, sometimes even on concrete systems. Such model checkers include the Java PathFinder system [HP00, VHB$^+$03], JNuke [ASB$^+$04], and similar systems [BPR01, CDH$^+$00, God97, HS99, Sto00]. Due to this, the distinction between static and dynamic analysis is blurring. Even though model checkers explore system behavior exhaustively, it can still be hard to find certain multi-threading failures such as a data race, low-level as well as high-level. In order to find such a failure, model checking typically requires system exploration to cause a violation of some explicitly stated property.

Regardless of whether model checkers are applied to models or software, they suffer from the state space explosion problem: The size of the state space is exponential in the size of the system, which includes the number of threads and program points where thread switches can occur. This is the reason why most systems are too complex for model checking. System *abstraction* offers a way to reduce the state space by merging several concrete states into a single abstract state, thus simplifying behavior. In general, an abstract state allows for a wider behavior than the original set of concrete states, preserving any potential failure states. Abstraction of the system by removing unneccesary behavior is therefore crucial to reduce the state space [BPR01, CDH$^+$00]. For actual system exploration, a number of partial-order reduction techniques have been proposed which all have in common that they do not analyze multiple independent interleavings when it can be determined that their effect is equivalent [Bru99, Hol91].

Finally, model checkers are often classified according to their underlying technology. *Explicit-state model checkers* [Hol91] were available first; they store each state directly in memory and are very fast for systems that fit into available memory. *Symbolic model checkers* [BCM$^+$90, McM93] store the system state in a data structure called Binary Decision Diagrams (BDDs) [Bry86]. This data structure can share common subexpressions of a formula denoting a set or property. Finally, *bounded model checkers* [BCC$^+$03] only explore a system up to a certain limit, typically using SAT solvers instead of BDDs as their underlying data structure [BCC$^+$99]. Recently, it has been shown that Craig interpolation [Cra57] can bridge the gap between bounded model checking and unbounded model checking of finite systems [McM03].

2.2.3 Comparison Between the Different Technologies

The four technologies presented, abstract interpretation, theorem proving, run-time verification, and model checking, are not always used in isolation. For instance, a theorem prover may be used to provide a correct abstraction of a given predicate [BPR01] which is then verified using model checking. At the time of writing, the boundaries between the tools are still fairly clear, with one technology dominating the work flow of a tool chain and others playing a subsidiary role in it. Therefore tools can still be attributed to a particular technology, although this distinction is likely going to be blurring in the future.

Strengths and weaknesses of each technology determine their suitability for a particular project. This section makes an attempt at classifying these, and should be understood as a guide, not as a final judgement. In each category, there exist tools that work differently than their common counterparts. Due to inherent difficulties in classifying such broad classes of technologies, an attempt is made to characterize certain crucial trade-offs that each technology offers.

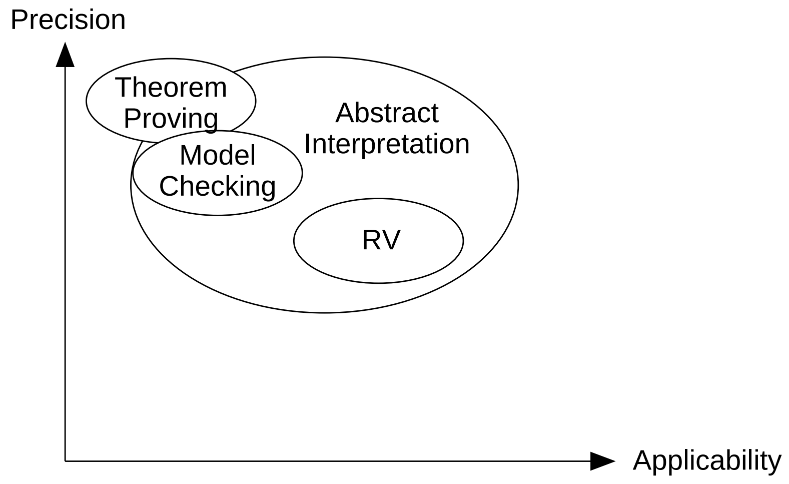


***Figure 2.1**: Applicability and precision of each technology.*

The first property investigated is the quality of the results that a tool can provide within its domain, and its applicability to a project. Applicability in this context refers to different kinds and sizes of programs, what models and properties can be verified, and how suitable a technology is within the context of a given development process. Figure 2.1 summarizes this trade-off.

Theorem proving is very labor-intensive. For obtaining satisfactory results, it requires an adaptation of the development process to that methodology [Abr96, CTW99]. However, it can give full confidence in all properties verified, and thus is very suitable for mission-critical software where it is worthwhile spending a significant amount of money on quality assurance. Model checking is often used in this area, where the goal is to prevent certain kinds of critical faults [BPR01]. Such projects are most successful within a specialized domain [HLP01], and while many properties can be checked decisively, the range of such properties is typically not as wide as with theorem proving, which is better suited to verification of unbounded systems [Pel01]. Run-time verification (RV) has the advantage that it is very easily applicable [BDG$^+$04]. The range of existing tools encompasses verification of hard-coded properties that require no user annotation [SBN$^+$97, ASB$^+$04] to verification of temporal properties [ABG$^+$03, Dru03]. RV does not always deliver precise results, which puts it into the same league as certain abstract interpretation-based tools that sacrifice both completeness and soundness for scalability and expediency [AB01]. Such simple static checkers can be used in early stages of a project, where an executable application may not even exist yet. On the other end of the spectrum of static tools, there are special-purpose tools geared towards a particular domain, delivering very precise results [BCC$^+$02]. Concerning large systems, only RV and certain static analyzers have so far successfully scaled up to larger systems [AB01, EM04, Har00].

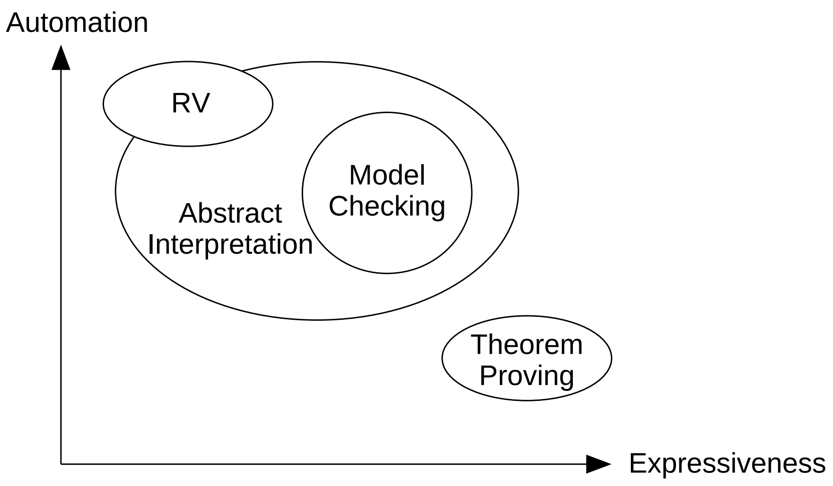


***Figure 2.2**: Expressiveness and degree of automation of each technology.*

The previous discussion already touched the second issue, which is shown in Figure 2.2: the degree of automation a tool can provide versus the range of properties that can typically be expressed. RV is limited to properties that are applicable to (and at least to some degree verifiable on) a single program trace [ABG$^+$03, NS03], but it is fully automated. Abstract interpretation can deliver the same degree of automation when verifying a fixed set of hard-coded properties [AB01]. Commonly, reports issued by a tool require further inspection due to possible false positives. Thus they can so far not be considered to be fully automated. Abstract interpretation can verify a larger set of properties if the tools is carefully tuned for its application domain [BCC$^+$02]. Model checkers typically are also built as general-purpose tools. Constructing a model of the software can be a labor-intensive process [HLP01] or also be fully automated [BPR01]. In the latter case, general-purpose properties were successfully re-used across a variety of device drivers, increasing automation. For theorem proving, automation is still is biggest weakness, since carrying out proofs typically requires human interaction.

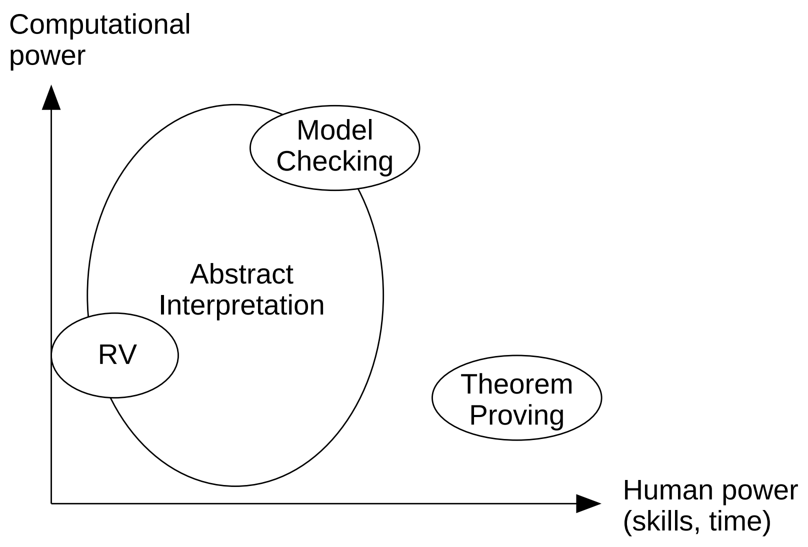


***Figure 2.3**: Computational and human power required for each technology.*

The final comparison is the trade-off between computational and human power required to use a tool effectively, outlined by Figure 2.3. Human power includes both the necessary skills and training to use a tool, and time required for extra program annotations or interaction with a tool. Run-time verification tools typically require very little training, but incur a certain overhead for program testing. This overhead is still smaller than for other tools, which has made RV very successful in practice [BDG$^+$04, Har00, NS03, HJ92]. Abstract interpretation typically requires some insight into the fact that a tool is imprecise [AB01] or the tool itself in case it is fune-tuned to a specific domain [BCC$^+$02]. Again, this technology covers a wide spectrum. Model checking typically is computationally very expensive [CGP99] and typically requires understanding of temporal logics such as LTL [Pnu77] in order to be used effectively. It is therefore still typically only used by highly trained engineers, despite attempts to simplify the complexity of temporal logics by providing a specification language as a front end [CDHR00]. Theorem proving, finally, does not only require such a mathematical background but also deep insight into mathematical proof strategies and capabilities of such a tool, typically requiring months of experience [Pau03].

2.2.4 Possible Application Strategies

Theorem proving can bring its full strength to bear in a project where it is used throughout the development process, by well-trained developers that carry out the mathematical work. Some mission-critical applications have been built successfully by starting with program proofs from which the final program was iteratively developed (by refinement) [Abr96, CTW99]. A similar development methodology can be used for model checking as well, where a model of an application is developed and then implemented once it is found to satisfy all required properties. This strategy requires that formal verification is part of a project from its very start, and is thus not applicable to existing systems.

The reverse direction is *deductive verification* [Pel01], where an attempt is made to prove properties in an existing implementation. The most promising aspect is that the actual implementation of a system (or at least a simplified, abstracted version of it) can be verified rather than a protocol or design [CP96]. Nonetheless, the complexity of such a system is so large that theorem proving usually requires too many resources [Pel01] and more mechanized approaches such as model checking [VHB$^+$03, HLP01] or abstract interpretation [BCC$^+$02, EM04] are more successful, even though they cannot cover all aspects of a potentially unbounded system.

Finally, simple, fast static analyzers, such as the use of strict compiler warnings or dedicated tools [AB01, AH04], can be a valuable tool throughout the development process, especially when development is still far from completion or no resources exist to use a more complex tool. Run-time verification tools typically also fall into that category [RV04] but are of course only applicable when a system has already reached the stage where it becomes testable.

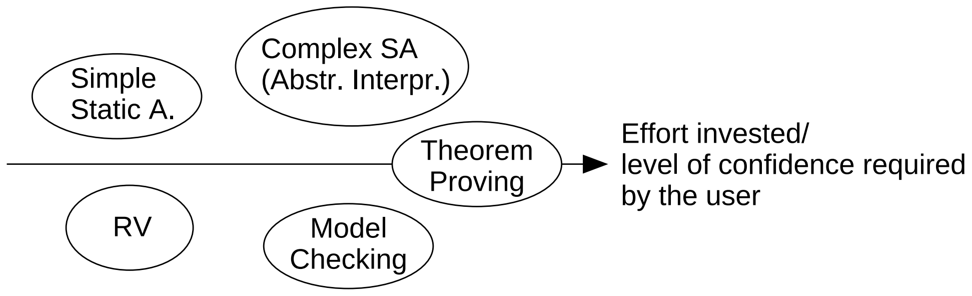

Simple Static A.
Complex SA (Abstr. Interpr.)
Theorem Proving
Effort invested/ level of confidence required by the user
RV
Model Checking

***Figure 2.4**: Application scenarios for different analysis technologies.*

Figure 2.4 summarizes this: Simple static analysis tools, based on abstract interpretation or ad-hoc methods, can be a highly useful first step for formal verification. Run-time verification is also easily applicable once a test suite exists. More complex static analyzers and model checkers are a good choice if a high degree of confidentiality is required in a domain where these technologies are well applicable, such as system controllers [BCC$^+$02, VHB$^+$03] or device drivers [BPR01]. Theorem provers require a very strong commitment to formal verification, but can handle problems where the other technologies fail. Finally, it should be emphasized that a combination of different tools and technologies is often more successful than pushing one individual technology to its limits [AB01, BDG$^+$04, EM04].

A possible tool chain combining these technologies could look as follows: a fast static analyzer is used as a first pass to reduce run-time checks. Remaining cases can be treated at run-time. This combination is often applied to type checking but can be extended to other properties. Since run-time checks cannot give full confidence in a system, more expensive methods such as model checking may be applied to cover the full behavior of the system.

The reason why model checking or theorem proving is usually not used at the beginning of a project, or for any kind of system, lies in the complexity of these methods. On larger systems, a major manual effort for specification, abstraction, and manual proving (in the case of theorem proving) is necessary. When verifying an implementation, it is therefore recommended to use more light-weight methods such as run-time verification first and then apply heavy-weight techniques if quality assurance policy requires this.

2.3 Concurrent Programming in Java

A brief introduction to concurrent programming in Java and other programming languages has been given in Section 1.2. This section describes concurrent programming mechanisms in Java in more detail. Even though this thesis describes concepts and implementations based on Java programs, the ideas are applicable to any programming language or environment that supports the same mechanisms, which are standardized by POSIX threads [But97] and available in many other programming languages such as C and C++. Java implementations commonly use a POSIX-compliant thread library underneath, even though this is not mandated by the standard [LY99].

2.3.1 Program data

Java is an object-oriented language where related data is organized in (dynamic) *instances* of objects which each have a common type or *class*. Such a type includes a set of *fields,* which are attributes of each object, and methods, which are functions that operate on instance data. Object instances are dynamically created at run-time. In addition to these instances, there exists a *static class instance* for each class [GJSB00]. This instance is a special instance which has a different set of fields and methods than the dynamic one. The static instance is a singleton instance [GHJV95] which is created after a class file is loaded into memory [LY99]. It is always globally accessible through its class name, unlike dynamic instances, which are only accessible through their instance reference [GJSB00].

Java memory is partitioned into a global *heap* (sometimes also denoted as main memory), which holds dynamically allocated data, and a set of *stacks,* with one stack for each thread holding method-local, thread-local data. A dynamic class instances is *live* (not eligible for garbage collection) as long as it is reachable from the stack of at least one active thread. If such an instance reference is potentially reachable by several threads, its fields are *shared* and can be accessed concurrently.

Within each method, Java also offers *local variables* which are used to store intermediate data that on the current stack frame while the method is executing. Such local variables are only visible within one method and created for the duration of each method call. Therefore they are also thread-local, because one such set of local variables will be created for each method call when several threads call the same method concurrently. Methods typically also use *stack variables* which are not available as a Java language construct. They are used to hold intermediate values for operands in Java bytecode [LY99]. Stack variables are also method-local and thread-local. In this thesis, the term *register* will denote both stack variables and local variables of the same method.

Even though the techniques in this thesis have been developed for application on object-oriented programs, the ideas transfer to non-object-oriented languages as well. The properties which will be defined below deal with atomicity of actions and atomic access to sets of values. A set of values does not necessarily have to be encapsulated in a single class and can represent arbitrary memory locations. Atomicity of actions treats thread-locality of data and actions using data; again, it is irrelevant how that data is organized within programming language constructs. Because of these reasons, the ideas presented in this thesis generalize trivially to non-object-oriented languages. This generalization is not described explicitly in the remainder of this text.

2.3.2 Multi-threading

Java includes multi-threading in its base classes and the language itself, unlike some other programming languages which use an external library to achieve this [But97]. The explicit availability of threads and locks as a language construct makes them easier to use and analyze, since all low-level constructs are standardized [GJSB00]. This overview only describes concurrency language features of Java which are relevant for this thesis.

Class `java.lang.Thread` allows creation of new threads at run-time. Several threads may run concurrently. At the beginning of execution of a Java program, thread `main` is started, representing the only active application thread.[1] Other threads are typically created by instantiating a class that either inherits from `java.lang.Thread` or implements interface `java.lang.Runnable`. In either case, it must implement a `run` method which specifies the code to be executed when such a thread instance is started. For practical purposes, the programmer can assume that the virtual machine runs on only *one* CPU, and each thread periodically receives a "time slice" by the scheduler. Note that the official Java specification poses no requirement for fair scheduling among threads of the same priority. This emphasizes once more that the programmer has to take *any* possible schedule into account.

Threads share the address space of the virtual machine. It is possible to keep certain special references thread-local [GJSB00, Lea99], but normally created references are always shared unless they can be guaranteed to be reachable by only a single thread. Threads may access the global heap directly, which is always the case for `volatile` fields [GJSB00] or in certain JVM implementations [Eug03].

However, in many JVM implementations, every thread also has a *working memory* in which it keeps its own working copy of variables that it must use or assign. A thread then operates on these working copies while the main memory contains the *master copy* of every variable. There are rules about when a thread is permitted or required to transfer the contents of its working copy of a variable into the master copy or vice versa. Most importantly, acquiring a lock forces a thread to re-read any working copies while releasing a lock writes any local copies back to main memory [LY99]. This has the consequence that any operations taking place without using locks may operate on stale data and never

[1] The Java specification allows for system threads running in the background, for example, as an idle thread [Eug03], or for garbage collection [PD00]. However, these threads cannot be controlled by the application and must not interfere with it.

become visible to other threads. This is the reason why correct locking is crucial for program correctness. Variables of size 32 bits or smaller which are declared `volatile` are exempt from this per-thread caching and always accessed atomically [LY99].[2]

2.3.3 Thread synchronization

Thread synchronization, often achieved through locking, ensures that access to shared data occurs in a well-defined manner, without any threads using stale data resulting from thread-local copies in working memory [GJSB00, LY99] or a lack of synchronization across several operations. Locking is used to prevent two threads from accessing the same object simultaneously. A *lock* is any Java object (but not a primitive value such as an integer) that is used as part of a `monitorenter` or `monitorexit` operation [LY99] which acquires and releases a lock, respectively. While any one thread holds the lock, another thread requesting it is *blocked* (suspended) until the first thread has released the lock. Locks in Java are *reentrant:* Acquiring the same lock twice has no effect other than increasing an internal counter; the lock is actually released when the counter is decreased to zero, i.e., when the number of lock release operations matches the number of previous lock acquisition operations.

```
synchronized (lock) { // acquires lock
  /* block of code */
  ...
  /* this sequence of operations is executed
   * while holding the lock */
} // releases lock
```

***Figure 2.5**: A `synchronized` block.*

There is only one way in the Java programing language to express lock acquisitions and releases: the `synchronized` statement. Using a synchronized block as shown in Figure 2.5, the current thread blocks until it is able to acquire `lock`. The lock is held until the entire block is finished (either when the last statement is executed or the block is aborted by other means, e.g. `break` or `return` statements, or exceptions).

A frequently used case of synchronization is synchronization on the current instance `this`, expressed by `synchronized(this)`. If such a block spans an entire method, keyword `synchronized` may instead by used to qualify a method. Such a method automatically acquires a lock on `this` before its body is executed. After method execution, the lock is released. (If a lock has been held before, acquiring it again simply increases the corresponding lock counter within the virtual machine, but has no other effect.)

A Java compiler transforms `synchronized` statements into a pair of bytecode instructions, `monitorenter` and `monitorexit`.[3] These instructions take one argument, the lock

[2]64-bit `volatile` variables are also exempt from being copied to thread-local working memory, but operations on them are not required to be atomic at the time of writing [LY99].

[3]Exception handlers may result in several copies of `monitorexit` instructions corresponding to one

to be acquired and released. A JVM has to implement any possible side-effects such as the flushing of thread-local copies as described above. Methods declared as `synchronized` are not implemented using these two bytecode instructions; instead, the lock on the current instance is acquired and released implicitly.

The remaining important synchronization primitives are available by the two methods `wait` and `notify`. If a thread holds a lock on instance `resource`, and has to wait for a certain condition to become true, common convention is to call `resource.wait()` inside a loop. This causes that thread to "sleep" (block), suspending it from execution. It remains suspended until another thread calls `resource.notify()`, which "wakes up" one thread (out of possibly many) waiting on `resource`. These methods are implemented as native code in the Java Run-time Environment, as they cannot be expressed by bytecode sequences [LY99].

Calling `notify` releases the lock, and causes the original (`wait`ing) thread to reacquire it before resuming execution. Normally, that thread has to verify again whether the condition it is waiting on now holds; hence `wait` is usually called inside a loop rather than an `if` statement. If it cannot be guaranteed that any thread that has just been notified can actually resume execution, i.e., the condition it is waiting on has become true, then `notifyAll` needs to be used instead. This will "wake up" all threads waiting on that resource (in random order). At least one of them has to be able to continue execution; otherwise all waiting threads may end up stopped. Potential problems (livelocks and deadlocks) arising with incorrect use of `wait` or `notify` are not part of this thesis but are documented in previous work [Art01].

2.3.4 Lock synchronization structure

The design of the Java locking primitive using keyword `synchronized` automatically guarantees that lock and unlock operations always occur pairwise, even though they may be nested [Lea99]. An unlock operation therefore must operate on the lock that corresponds to the last lock operation whose lock was not yet released. This thesis assumes such a symmetrical structure of lock operations for simplicity. However, the ideas presented here generalize to non-symmetrical locking operations. Such operations are not possible in the Java programming language but can theoretically be implemented in Java bytecode [LY99]. The generalization is described here and can be applied to both high-level data race and block-local atomicity checks.

A Java program that acquires lock a, then b, will release the second lock b first. The lock operations of such a program are shown in Figure 2.6. After releasing inner lock b, the lock set again equals $\{a\}$, the same as prior to the acquisition of lock b. Therefore accesses that occur after releasing b but before releasing a affect view or monitor block 1, the one used to represent actions under lock set $\{a\}$.

Figure 2.7 shows how a program that breaks this nesting and releases lock a before releasing b. Because the lock set after releasing the first lock, a, differs from any previously used lock set, a new view or monitor block is used to represent actions under lock set $\{b\}$. Note that such actions likely lead to low-level data races, which can be detected by the

`monitorenter` operation. At run-time, exactly one such `monitorexit` operation is always executed.

Lock operation	Lock set after operation	Corresponding view or monitor block
`monitorenter` a	$\{a\}$	1
`monitorenter` b	$\{a,b\}$	2
`monitorexit` b	$\{a\}$	1
`monitorexit` a	$\{\}$	–

Figure 2.6: *Nested locking operations.*

Lock operation	Lock set after operation	Corresponding view or monitor block
`monitorenter` a	$\{a\}$	1
`monitorenter` b	$\{a,b\}$	2
`monitorexit` a	$\{b\}$	3
`monitorexit` b	$\{\}$	–

Figure 2.7: *Generalization to non-nested locking operations.*

Eraser lock set algorithm. That algorithm only uses lock sets and thus is independent of the order of lock operations [SBN$^+$97]. The need for such non-nested locking is very rare in practice. In cases where it is desirable, it can be emulated in Java using symmetrical locking [Lea99].

2.4 Concurrency Errors in Software

Multi-threaded, or concurrent, programming has become increasingly popular in enterprise applications and information systems [AB01, Sun05]. Multi-threaded programming, however, provides a potential for introducing intermittent concurrency errors that cannot occur in sequential programs and are hard to find using traditional testing. The main source of this problem is that a multi-threaded program may execute differently from one run to another due to the apparent randomness in the way threads are scheduled. Since testing typically cannot explore all schedules, some bad schedules may never be discovered.

Such schedules carry the potential of new sets of program failures. A common problem that may occur under certain schedules is a deadlock [Lea99]:

> Among the most central and subtle liveness failures is deadlock. Without care, just about any design using synchronization on multiple cooperating objects can contain the possibility of a deadlock.

Two types of deadlocks are discussed in literature [Kna87, Sin89]: *resource deadlocks* and *communication deadlocks*. In a resource deadlocks, a process or thread must wait until it has acquired all the requested resources needed for a computation. A deadlock occurs if several threads request such a resource held by another thread. Efficient algorithms exist to detect such deadlocks at run-time [BH03]. In communication deadlocks, messages or shared conditionals are the resources for which threads wait. It is possible that such a message a thread is waiting for is never sent, and thus that thread can never continue. Tools exist to show such a lack of progress in a stalled thread [Har00]. This thesis does not cover deadlocks further; for more information, see [BH03].

Another kind of error that often occurs in multi-threaded programs is a *data race,* as defined below. Traditionally this term has denoted unprotected field accesses, which will be referred to as *low-level data races.* However, the absence of low-level data races still allows for other concurrency problems, such as *high-level data races* [AHB03] and *atomicity violations* [FF04, WS03]. Low-level as well as high-level data races can be characterized as occurring when two or more threads access a shared region simultaneously, the definition of region being dependent on what kind of data race is referred to.

Finally, atomicity violations denote actions that do not execute atomically. Atomic execution of an action, combined with actions from other threads, produce the same global result in concurrent and serial execution [Fla04]. The block-local atomicity algorithm [ABH04] presented here detects stale-value errors as defined by Burrows and Leino [BL02] and augments existing approaches concerning low-level and high-level data races, such that it can be employed in conjunction with these analyses. High-level data races and previously presented atomicity checks suffer from the fact that they show violations of common conventions, which do not necessarily imply the presence of a fault. Data flow between protected regions (`synchronized` blocks in Java) is ignored. The block-local atomicity approach analyzes data flow, making it more precise than previous atomicity-based approaches. It complements low-level and high-level data races by finding additional errors.

Data races and atomicity violations are extremely hard to detect with traditional testing techniques. Not only does a simultaneous access from two or more threads to a particular region have to occur, but this should additionally result in corrupted data, which violate some user-provided assertion. In traditional testing, data races therefore are highly schedule-dependent and usually harder to detect than deadlocks, which often cause some visible activity to halt.

The suggested detection algorithms analyze a *single execution trace* obtained by running an instrumented version of the program. In this context, single execution means the program is run once. While running, it emits a series of events which consitute the execution trace. The analysed properties are, for practical purposes, mostly independent of the thread interleavings during program execution. Hence the program only needs to be run once.

Both detection algorithms require no user-provided requirement specification, and hence are in line with the Eraser algorithm [SBN$^+$97]. This means that the algorithms are totally automated, requiring no user guidance at all beyond normal input. The algorithms look for *fault patterns* in the execution trace, and raise a warning in case such are detected. Such fault patterns denote violations of properties representing programming policies. Such policies are designed to ensure the absence of an error, such as a data race. A data race does not have to occur in the run of a faulty program, but its fault pattern can still be detected. This is why verification of fault patterns is more powerful than traditional testing techniques.

The high-level data race algorithm is neither sound, nor complete, and may yield false positives (false warnings) and false negatives (missed errors). However, the increased probability of detecting errors strongly out-balances this, in particular considering that it is fully automated. In addition, practice seems to support that the rates of false positives as well as false negatives are low. The algorithm has been implemented in JNuke [ASB$^+$04] after a prototype has been implemented the Java PathExplorer (JPaX) tool [ABG$^+$03, HR01].

The block-local atomicity is sound (not missing errors) but incomplete, producing warnings that may denote benign usages of stale values. The algorithm is very modular and thus suitable for static analysis; it has been implemented as a generic algorithm that can be executed both in static and dynamic analysis [AB05a]. Compared to other atomicity-based approaches, it is simpler yet more precise because it captures data flow and thus models the semantics of the analyzed programs more precisely. It is related to Burrow's algorithm [BL02] but models the full semantics of Java bytecode, including arithmetic expressions. The checking algorithm is implemented as a dedicated algorithm in JNuke [ASB$^+$04], both for static and dynamic analysis. Experiments show that the static variant is about two orders of magnitude faster than Burrows' prototype.

Although the algorithms and tools presented here analyze Java programs, the principles and theory presented here are universal and also apply in full to concurrent programs written in languages like C and C++ [NBF98].

2.4.1 Low-level Data Races

The traditional definition of a data race is as follows [SBN$^+$97]:

> A data race can occur when two concurrent threads access a shared variable and when at least one access is a write, and the threads use no explicit mechanism to prevent the accesses from being simultaneous.

Consider for example two threads, that both access a shared object containing a counter variable *x*, and assume that both threads call an *increase()* method on the object, which increases *x* by 1. The *increase()* method is compiled into a sequence of bytecode instructions (load *x* to the operand stack, add 1, write back the result). The Java Virtual Machine (JVM) executes this sequence non-atomically. Suppose the two threads call *increase()* at nearly the same time and that each of the threads execute the *load* instruction first, which loads the value of *x* to the thread-local operand stack. Then they will both add 1 to the original value, which results in a combined increment of 1 instead of 2. According to the Java memory model, it is even possible that the effect of a write operation will never be observed by other threads [GJSB00]. Therefore it is universally agreed that low-level data races must be avoided.

This traditional notion of data race shall be referred to as a *low-level data race*, since it focuses on a single variable. The standard way to avoid low-level data races on a variable is to *protect* the variable with a lock: all accessing threads must acquire this lock before accessing the variable, and release it again after. In Java, methods can be defined as `synchronized` which causes a call to such a method to lock the current object instance. Return from the method will release the lock. Java also provides an explicit statement form `synchronized(`*obj*`){`*stmt*`}`, for taking a lock on the object *obj*, and executing statement *stmt* protected under that lock. If the above mentioned *increase()* method is declared `synchronized`, the low-level data race cannot occur.

```
public void reset {
  synchronized (a) {
    synchronized (b) {
      shared.value = 0;
    }
  }
}

public void inc {           public void dec {
  synchronized (a) {          synchronized (b) {
    shared.value++;             shared.value--;
  }                           }
}                           }
```

***Figure 2.8**: A low-level data race example.*

Data races often occur when locking is used incorrectly, as illustrated in Figure 2.8. It demonstrates how a field of a shared instance (`shared.value`) is accessed by several methods. Two locks, *a* and *b*, protect access. The first method, `reset`, uses both locks. While a given thread is holding both locks *a* and *b*, it is therefore not possible that the `synchronized` block of `reset`, `inc`, or `dec`, can be executed by another thread at the same time. However, this lock protection does not apply to mutual exclusion in methods `inc` and `dec`. Assume thread t_a calls `inc`, using lock *a*. This does not prevent another thread t_b from calling `dec` at the same time, since lock *b* is not taken by t_a. As described above, the increment and decrement operations are performed as a sequence of instructions which load the value on the stack, manipulate it, and write it back. Like in the scenario given above, there is no mutual exclusion between the two threads. Therefore the total effect of two threads executing `inc` and `dec` concurrently may be a net change of -1, 0, or $+1$.

Assume now a correct run, with a given schedule that executes `reset` before `inc`, which terminates before `dec` is called. The schedule chosen does not execute accesses to the shared value concurrently, and therefore the net result of all operations is 0, which is correct. The potential data race under another schedule, which produces an incorrect result, is missed.

Several algorithms and tools have been developed for analyzing multi-threaded programs for low-level data races. The Eraser algorithm [SBN$^+$97], which has been first implemented in the Visual Threads tool [Har00] to analyze C and C++ programs, is an example of a dynamic algorithm that examines a program execution trace for locking patterns and variable accesses in order to predict potential data races. The Eraser algorithm maintains a *lock set* for each *variable*, which is the set of locks that have been owned by all threads accessing the variable in the past. Each new access causes a refinement of the lock set to the intersection of the lock set with the set of locks currently owned by the accessing thread. The set is initialized to the set of locks owned by the first accessing thread. If the set ever becomes empty, a data race is possible. The Eraser algorithm is also implemented for Java by JNuke [ASB+04], JPaX [HR01, Hav00], and JProbe [Sit05], and in valgrind for C programs [NS03].

In the example of Figure 2.8, Eraser detects the possible data race as follows: During the first access in `reset`, the lock set for `shared.value` is initialized to $\{a,b\}$. A subsequent access to `inc`, with lock set $\{a\}$, refines the lock set protecting the shared value to $\{a\}$. Intersecting this lock set with lock set $\{b\}$ of the final access in `dec` results in the empty set. Therefore the Eraser algorithm correctly concludes that in this example, a data race can occur under certain schedules, even though the given schedule did not reveal an error.

2.4.2 High-level Data Races

A program may contain a potential for concurrency errors, even when it is free of low-level data races and deadlocks. Low-level data races concern unprotected accesses to shared fields. The notion of high-level data races refers to sequences in a program where each access to shared data is protected by a lock, but the program still behaves incorrectly because operations that should be carried out atomically can be interleaved with conflicting operations [AHB03].

Figure 2.9 shows an example for a high-level data race. It uses a shared coordinate instance, `coord`, with two components, x and y. While the `swap` operation on the two components x and y is atomic, the `reset` operation is not. Because the lock is released after setting x to 0, other threads may observe state $\langle 0, y\rangle$, an intermediate state, which is inconsistent. If `swap` is invoked by another thread before `reset` finishes, this results in final state $\langle y, 0\rangle$. This is inconsistent with the semantics of `swap` and `reset`. The view consistency algorithm finds such errors [AHB03]. More formal and generic definitions of view inconsistency are presented in Chapter 3.

Detecting this error using normal testing is very hard since it requires not only to execute the just described interleaving (or a similar one), but it also requires the formulation of a correctness property that can be tested for, and which is violated in the above scenario. However, regarding this as a view inconsistency problem makes it possible to find the error without actually executing this particular interleaving, and without a requirement specification.

Both aspects are very important. This is confirmed by the success of the Eraser algorithm [SBN+97]. Data races, both low-level or high-level ones, occur only rarely in a concrete run, because they only appear under a certain schedule. This makes them very hard to observe using traditional testing. Therefore one tries to observe a policy, such

```
public void swap() {               public void reset() {
  int oldX, oldY;                    synchronized (lock) {
  synchronized (lock) {                coord.x = 0;
    oldX = coord.x;                  }
    oldY = coord.y;                  // inconsistent state (0,y)
    coord.x = oldY; // swap X        synchronized (lock) {
    coord.y = oldX; // swap Y          coord.y = 0;
  }                                  }
}                                  }
```

***Figure 2.9**: A high-level data race resulting from three atomic operations.*

as the locking behavior of a program, and infer potential errors from that. The locking behavior of each thread usually does not change across different schedules (it is only dependent on the input, which can be automated), and therefore it is a reliable base for fault-finding algorithms.

Requiring no annotations is not only important when it comes to the usability of a tool. Tools like ESC/Java suffered from having a high initial overhead due to the required annotations in a program [DLNS98]. Algorithms that do not require annotations are also more interesting from a research point of view: It is attempted to extract as much information as possible from the program itself rather than having the user specify (possibly incorrect or redundant) information. Of course such approaches are sometimes inherently more limited than ones requiring annotations. The goal is to find algorithms that still capture the underlying problems with sufficient precision.

The algorithm presented in this thesis achieves this to a high degree. In the Eraser algorithm [SBN$^+$97] for detecting low-level data races, the set of locks protecting a single variable, referred to as the *lock set*, is considered. In this thesis this idea is turned upside down. The *variable set* associated to a *lock* is now of interest. This notion makes it possible to detect what shall be referred to as *high-level data races*. The inspiration for this problem was originally due to a small example provided by Doug Lea [Lea00], which will be presented in Chapter 3.

2.4.3 Atomic Sequences of Operations

The absence of low-level and high-level data races still allows for other concurrency errors. Figure 2.10 shows such an error: The increment operation is split into a read access, the actual increment, and a write access. Consider two threads, where one thread has just obtained and incremented the shared field. Before the updated value is written back to the shared field, another thread may call method `inc` and read the *old* value. After that, both threads will write back their result, resulting in a total increment of only one rather than two.

The problem is that the entire method `inc` is not *atomic*, so its outcome may be unexpected. Approaches based on reduction [FF04, WS03] detect such atomicity violations. The algorithm presented here uses a different approach but also detects the error in the

```
public void inc() {
  int tmp;
  synchronized (lock) {
    tmp = shared.field;
  } // lock release
  tmp++;
  synchronized (lock) {
    shared.field = tmp;
  }
}
```

***Figure 2.10**: A non-atomic increment operation.*

example. Moreover, it is conceptually simpler than previous atomicity-based approaches and at the same time more precise.

The difference between atomicity violations and high-level data races can be illustrated by the example in Figure 2.11. It closely resembles Figure 2.9, but the `swap` method has been split up into two `synchronized` blocks while the `reset` method is now executing in a single block. View consistency will not report an error for the faulty `swap` method, because its two views both include x and y. However, the new example contains another error: The shared coordinate may be updated when the lock is released, and the local copies may contain outdated, *stale* values [BL02]. Consider the following scenario in Figure 2.11: Thread t_1 calls `swap`, which executes up to the end of the first `synchronized` block. Then another thread t_2 calls `reset`. The thread scheduler lets t_2 run, which resets both coordinates to 0. Then t_1 finishes its `swap` method, using the old values for x and y in the process. Therefore the call to `reset` is nullified by the use of stale values in `swap`! The block-local algorithm presented here correctly deduces from the data flow in the `swap` method that the entire method must be atomic (its `synchronized` block must be contiguous). Conversely, the block-local atomicity algorithm could not have detected the high-level data race in the old reset method (which contained two `synchronized` blocks). The reason for this is the absence of data flow in the two assignments to x and y, since the two values assigned are constants and independent of each other.

2.5 Summary

Software analysis techniques can be categorized into abstract interpretation, theorem proving, run-time verification, and model checking. Abstract interpretation constructs a simplified model of the program on which a fix point of given properties is calculated. Theorem proving applies a sequence of mathematical proof transformation steps to a program or a model thereof. Run-time verification supervises additional properties during test execution, which allow for detection of faults that cannot be found by testing alone. Model checking explores the entire state space of a program.

```
public void swap() {            public void reset() {
  int oldX, oldY;
  synchronized (lock) {           synchronized (lock) {
    oldX = coord.x;                 coord.x = 0;
    oldY = coord.y;                 coord.y = 0;
  }                               }
  synchronized (lock) {
    coord.x = oldY; // swap X   }
    coord.y = oldX; // swap Y
  }
}
```

***Figure 2.11**: An atomicity violation that is not a high-level data race.*

Java is a multi-threaded programming language offering language constructs for construction of threads and inter-thread synchronization. The focus of this thesis is protection of data accesses by using `synchronized` blocks, which only permit the thread holding a given lock to execute that section of code.

The lack of synchronization can lead to various concurrency failures, which include data races and atomicity violations. Data races can be categorized into low-level data races and high-level data races. Low-level data races denote unprotected accesses to data while high-level data races describe accesses to sets of data where synchronization of all accesses is not consistent. Atomicity violations describe actions which are not synchronized during their entire life time. The key difference to high-level data races is that these are concerned with data accesses while atomicity violations apply to actions and data flow between accesses.

3 High-level Data Races

Data races are a common problem in multi-threaded programming [Lea99]. Experience shows that the classical notion of data race is not powerful enough to capture certain types of inconsistencies occurring in practice. This chapter investigates data races on a higher abstraction layer. This enables detection of inconsistent uses of shared variables, even if no classical race condition occurs. For example, a data structure representing a coordinate pair may have to be treated atomically. By lifting the meaning of a data race to a higher level, such problems can now be covered. This chapter defines the concepts view and view consistency to give a notation for this novel kind of property. It describes what kinds of errors can be detected with this new definition, and where its limitations are. It also gives a formal guideline for using data structures in a multi-threaded environment.

This chapter is organized as follows. Section 3.1 motivates the problem with a real error found in a space craft controller. Section 3.2 introduces the problem of high-level data races. A formal definition of high-level data races is given in Section 3.3. Section 3.4 summarizes this chapter. Related work is discussed in Chapter 10, which includes other concurrency properties.

3.1 Motivation

A problem that was detected in NASA's *Remote Agent* [PGK$^+$97], a space craft controller, shall serve as a realistic example of a high-level data race situation. The problem was originally detected using model checking [HLP01]. The fault was very subtle, and was originally regarded hard to find without actually exploring all execution traces as done by a model checker. Because only very particular thread interleavings result in a data race and hence corrupted data, a single execution trace does usually not exhibit this failure. As it turns out, it is an example of a high-level data race, and can therefore be detected with the low-complexity algorithm presented here.

The Remote Agent is an artificial-intelligence-based software system for generating and executing plans on board a space craft. A plan specifies a set of tasks to be executed within certain time constraints. The plan execution is performed by the *Executive*. A sub-component of the Executive, the *task manager*, is responsible for managing the execution of tasks, once the tasks have been activated. The data structures used by the task manager are illustrated in Figure 3.1.

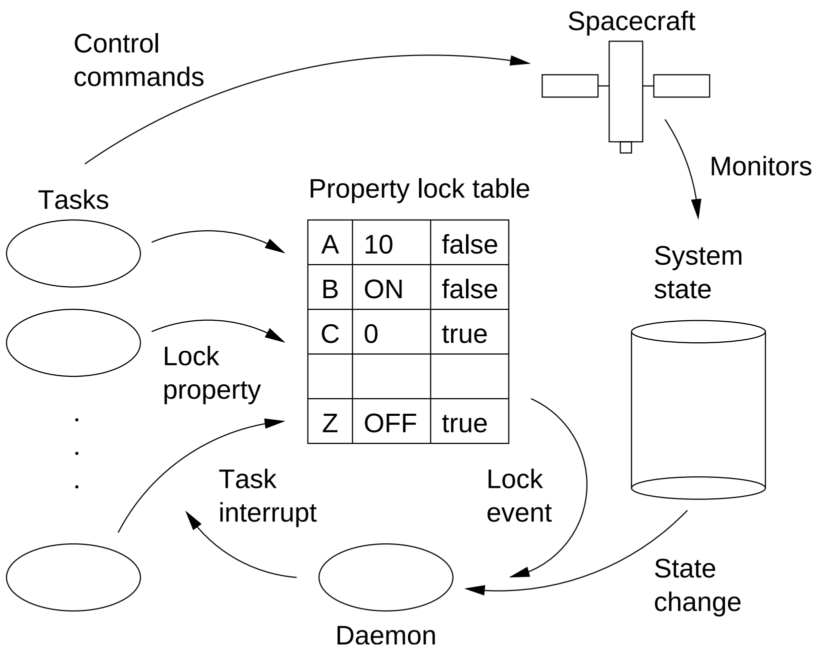


***Figure 3.1**: The Remote Agent Executive*

The state of the spacecraft (at any particular point) can be considered as consisting of a set of properties, each being an assignment of a value to a variable corresponding to a component on board the space craft. The values of variables are continuously read by sensors and recorded in a *system state*. A task running on board the space craft may require that specific properties hold during its execution, and it notifies the task manager about such requirements before start. That is, upon the start of the task, it first tries to lock those properties it requires in a *lock table*, telling the task manager that its execution is only safe if they hold throughout the execution. For example, a task may require *B* to be *ON* (assuming that *B* is some system variable). Now other threads cannot request *B* to be *OFF* as long as the property is locked in the lock table. Next, the task tries to achieve this property (changing the state of the space craft, and thereby the system state through sensor readings), and when it is achieved, the task sets a flag *achieved*, associated with that variable, to *true* in the lock table. This flag is true whenever the system state is expected to contain the property. These flags are initially set to false at system initialization, are set to true in the just described scenario, and are set back to false once the task no longer requires the property to hold.

A *Daemon* constantly monitors that the system state is consistent with the lock table in the sense that properties that are required to hold, as stated in the lock table, are actually also contained in the system state. That is, it checks that: *if a task has locked a value to some variable, and the corresponding flag* achieved *is true, then it must be a true property of the space craft, and hence true in the system state*. Violations of this property may occur by unexpected events on board the space craft, which cause the system state to be updated. The daemon wakes up whenever events occur, such as when the lock table or the system state are modified. In case an inconsistency is detected, the involved tasks are interrupted.

Task

```
synchronized(table) {
  table[N].value = V;
}

/* achieve property */

synchronized(table) {
  table[N].achieved = true;
}
```

Daemon

```
synchronized(table) {
  if (table[N].achieved &&
    system_state[N] !=
      table[N].value) {

    issueWarning();
  }
}
```

Figure 3.2: *The synchronization inconsistency between a task and the daemon.*

The relevant code from the task and the daemon is illustrated in Figure 3.2, using Java syntax. (The Remote Agent was coded in LISP.) The task contains two separate accesses to the lock table, one where it updates the value and one where it updates flag *achieved*. The daemon on the other hand accesses all these fields in one atomic block. This can be described as an inconsistency in lock views, as described below, and actually presents an error potential.

The error scenario is as follows: suppose the task has just achieved the property, and is about to execute the second synchronized block, setting flag *achieved* to true. Suppose now however, that suddenly, due to unpredicted events, the property is destroyed on board the space craft, and hence in the system state. Assume that the daemon wakes up, and performs all checks. Since flag *achieved* is false, the daemon reasons incorrectly that the property is not supposed to hold in the system state, and hence it does not detect any inconsistency with the lock table (although conceptually there is one). Only then the task continues, and sets flag *achieved* to true. The result is that the violation has been missed by the daemon. The nature of this error can be described as follows:

> The daemon accesses the value and flag *achieved* in one atomic block, while the task accesses them in two different blocks. These two different ways of accessing tuple $\{value, achieved\}$, atomically and as a compound operation, is an inconsistency called *view inconsistency*.

The seriousness of the error scenario depends on the frequency with which the daemon gets activated. If events updating the lock table or the system state occur often, the daemon will just detect the problem later, and hopefully soon enough. However, if such events are far apart, the task may execute for a while without its required property holding. In the above example the view inconsistency is in itself not an error, but a symptom that if pointed out may direct the programmer's attention to the real problem, that property achievement and setting the flag *achieved* are not done in one atomic block. Note that repairing this situation is non-trivial since achieving properties may take several clock cycles, and it is therefore not desirable to hold the lock on the table during this process.

3.2 Informal Definition of High-level Data Races

Consistent lock protection for a shared field ensures that no concurrent modification is possible. However, this only refers to low-level access to the fields, not their entire use or their use in conjunction with other fields. The remainder of this chapter assumes detection of low-level data races is covered by the Eraser algorithm [SBN$^+$97], which can be applied in conjunction with the analysis described in this chapter. The definition of high-level data races will be shown by examples before it will be formalized. First a very basic example is presented, followed by more pathological ones, which at first sight may appear as high-level data races, but which shall not be classified as such.

3.2.1 Basic Definition

Figure 3.3 shows a class implementing a two-dimensional coordinate pair with two fields x, y, which are guarded by a single lock.

```
class Coord {
  double x, y;
  public Coord(double px, double py) { x = px; y = py; }
  synchronized double getX() { return x; }
  synchronized double getY() { return y; }
  synchronized Coord getXY() { return new Coord (x, y); }
  synchronized void setX(double px) { x = px; }
  synchronized void setY(double py) { y = py; }
  synchronized void setXY(Coord c) { x = c.x; y = c.y; }
}
```

***Figure 3.3**: The `Coord` class encapsulating points with x and y coordinates.*

If only `getXY`, `setXY`, and the constructor are used by any thread, the pair is treated atomically by all accessing threads. However, the versatility offered by the other accessor (`get`/`set`) methods is dangerous: if one thread only uses `getXY` and `setXY` and relies on complete atomicity of these operations, other threads using the other accessor methods (`getX`, `setX`, `getY`, `setY`) may falsify this assumption.

Imagine for example a case where one thread, t_r, reads both coordinates while another thread, t_w, sets them to zero. Assume that t_r reads the variables with `getXY`, but that t_w's write operation occurs in two phases, `setX` and `setY`. The thread t_r may then read an *intermediate result* which contains the value of x already set to zero by t_w but still the original y value (not zeroed yet by t_w). This is clearly an undesired and often unexpected behavior. In this chapter, the term *high-level data race* will describe this kind of scenario:

> A high-level data race is the occurrence of two concurrent threads accessing a set V of shared variables. That set should be accessed atomically, but at least one of the threads does not access the variables in V atomically.

In the coordinate pair example above, the set V is $\{x, y\}$, and thread t_w violates the atomicity requirement. Of course, a main question is how one determines whether a set of variables are to be treated atomically, assuming that the user does not specify that explicitly. For now it is assumed that an oracle determines this. In Section 3.3 an approximation to this oracle will be suggested, which does not require any kind of specification to be provided by the user. Of course it is an undecidable problem in practice, and furthermore requires a specification of the expected behavior of the program. For instance, in the above coordinate pair example, atomicity might not be required at all if the reader only wants to sample an x value and a y value without them being related.

3.2.2 Refinement of Basic Definition

Although the definition above may be useful, it yields false positives (false warnings). Using the coordinate example, any use of the methods `getX`, `setX`, `getY`, and `setY` will cause a high-level data race. However, there exist scenarios where some of these access methods are allowed without the occurence of high-level data races. Hence the notion of high-level data race needs to be refined. This is analog to the refinement in Eraser [SBN+97] of the notion of low-level data races in order to reduce the number of false positives.

The refinement of the definition will be motivated with the example in Figure 3.4, which shows four threads working in parallel on a shared coordinate pair c. Thread t_1 writes to c (and is similar to t_w from Subsection 3.2.1) while the other threads t_2, t_3 and t_4 read from c (t_3 is similar to t_r from Subsection 3.2.1). The threads use local variables $\texttt{x}_i$ and $\texttt{y}_i$ of type `double` and $\texttt{d}_i$ of type `Coord`, where i identifies the thread.

Thread t_1 Thread t_2

```
d1 = new Coord(1, 2);   x2 = c.getX();
c.setXY(d1);            use(x2);
```

Thread t_3 Thread t_4

```
                       x4 = c.getX();
                       use(x4);
    x3 = c.getX();     d4 = c.getXY();
    y3 = c.getY();     x4 = d4.getX();
    use(x3, y3);       y4 = d4.getY();
                       use(x4, y4);
```

***Figure 3.4**: One thread updating a pair of fields and three other threads reading fields individually.*

Initially, only threads t_1 and t_3 are considered. That situation is already described in Section 3.2.1. Inconsistencies might arise with thread t_3, which reads x in one operation and y in another operation, releasing the lock in between. Hence, thread t_1 may write to x and y in between, and t_3 may therefore obtain inconsistent values corresponding to two *different* global states.

Now consider the two threads t_1 and t_2. It is not trivial to see whether an access conflict occurs or not. However, this situation is safe. As long as t_2 does not use y as well, it does not violate the first thread's assumption that the coordinates are treated atomically. Even though t_1 accesses the entire pair $\{x, y\}$ atomically and t_2 does not, the access to x alone in t_2 can be seen as a partial read access. That is, the read access to x may be interpreted as reading $\{x, y\}$ and discarding y. So both threads t_1 and t_2 behave in a consistent manner. Each thread is allowed to use only a part of the coordinates, as long as that use is consistent.

The difficulty in analyzing such inconsistencies lies in the wish to still allow such partial accesses to sets of fields, like the access to x of thread t_2. The situation between t_1 and t_4 serves as another, more complicated, example of a situation which at first sight appears to provide a conflict, but which shall be regarded as safe. Regard thread t_4 as consisting of two operations: the first consisting of the first two statements, including `use(x4)`, and the second operation consisting of the remaining four statements. The second operation is completely self-contained, and accesses, in addition to y, everything the first operation accesses (namely x). Consequently, the first operation in t_4 likely represents an operation that does not need y. Therefore, the two operations are unrelated and can be interleaved with the atomic update statement in t_1 without interfering with the operations of t_4 on x and y. On a more formal basis, t_4 is safe because the set of variables accessed in the first operation of t_4 is a subset of the set of variables accessed in its second operation; the variable sets form a *chain*. When they do not form a chain, they *diverge*. On the basis of this example, the definition of high-level data races can be refined as follows:

> A high-level data race can occur when two concurrent threads access a set V of shared variables, which should be accessed atomically, but at least one of the threads accesses V partially several times such that those partial accesses diverge.

This definition is adopted for the remainder of this chapter. It can, however, still lead to false positives and false negatives as is described in Section 3.3.4.

The algorithm presented in the remainder of this chapter does not distinguish between read and write accesses. This abstraction is sufficiently precise because view consistency is independent of the whether the access is a read or a write: A non-atomic read access may result in inconsistent values among the tuple read, because other threads may update the tuple between reads. A write access that is carried out non-atomically allows other threads to read partial updates between write operations. Note that it is assumed that at least one write access occurs; constant values can be ignored in this analysis.

3.3 Formal Definition of High-Level Data Races

This section defines *view consistency.* It lifts the common notion of a data race on a single shared variable to a higher level, covering sets of shared variables and their uses. This definition assumes that the specification of what fields have to be treated atomically is not provided by the user. It is instead extracted by program analysis. This analysis turns the problem of high-level data races into a testable property, using view consistency. The definition of this analysis is precise but allows for false positives and false negatives. This is discussed at the end of this section.

3.3.1 Views

A lock *guards* a shared field if it is held during an access to that field. The same lock may guard several shared fields. Views express what fields are guarded by a lock. Let I be the set of object instances generated by a particular run of a Java program. Then F is the set of all fields of all instances in I.

A *view* $v \in \mathbb{P}(F)$ is a subset of F. Let l be a lock, t a thread, and $B(t,l)$ the set of all `synchronized` blocks using lock l executed by thread t. For $b \in B(t,l)$, a view *generated by t* with respect to l, is defined as the set of fields accessed in b by t. The *set of generated views* $V(t) \subseteq \mathbb{P}(F)$ of a thread t is the set of all views v generated by t. In the previous example in Figure 3.4, thread t_1 using both coordinates atomically generates view $v_1 = \{x,y\}$ under lock $l = \texttt{c}$. Thread t_2 only accesses x alone under l, having view $v_2 = \{x\}$. Thread t_3 generates two views: $V(t_3) = \{\{x\},\{y\}\}$. Thread t_4 also generates two views: $V(t_4) = \{\{x\},\{x,y\}\}$.

3.3.2 Views in Different Threads

A view v_m generated by a thread t is a *maximal view*, iff it is maximal with respect to set inclusion in $V(t)$:

$$\forall v \in V(t)\ [v_m \subseteq v \rightarrow v_m = v]$$

Let $M(t)$ denote the set of all maximal views of thread t. Only two views which have fields in common can be responsible for a conflict. This observation is the motivation for the following definition. Given a set of views $V(t)$ generated by t and a view v' generated by another thread, the *overlapping views* of t with v' are all non-empty intersections of views in $V(t)$ with v':

$$\text{overlap}(t,v') \equiv \{v' \cap v \mid (v \in V(t)) \wedge (v \cap v' \neq \emptyset)\}$$

A set of views $V(t)$ is *compatible* with the maximal view v_m of another thread iff all overlapping views of t with v_m form a chain:

$$\text{compatible}(t,v_m) \quad \text{iff} \quad \forall v_1,v_2 \in \text{overlap}(t,v_m)\ [v_1 \subseteq v_2 \vee v_2 \subseteq v_1]$$

View consistency is defined as mutual compatibility between all threads: A thread is only allowed to use views that are compatible with the maximal views of all other threads.

$$\forall t_1 \neq t_2,\ v_m \in M(t_1)\ [\text{compatible}(t_2,v_m)]$$

In the example in Figure 3.4, the views were

$$\begin{array}{rcl} V(t_1) & = & M(t_1) \; = \; \{\{x,y\}\} \\ V(t_2) & = & M(t_2) \; = \; \{\{x\}\} \\ V(t_3) & = & M(t_3) \; = \; \{\{x\},\{y\}\} \\ V(t_4) & = & \{\{x\},\{x,y\}\} \\ M(t_4) & = & \{\{x,y\}\} \end{array}$$

There is a conflict between $t_1$and t_3 as stated, since $\{x,y\} \in M(t_1)$ intersects with the elements in $V(t_3)$ to $\{x\}$ and $\{y\}$, which do not form a chain. A similar conflict exists between t_3 and t_4.

The above definition of *view consistency* uses three concepts: the notion of *maximal views*, the notion of *overlaps*, and finally the *compatible* notion, also referred to as the *chain* property. The chain property is the core concept. Maximal views do not really contribute to the solution other than to make it more efficient to calculate and reduce the number of warnings if a violation is found. The notion of overlaps is used to filter out irrelevant variables.

3.3.3 Examples

#		Thread t_a	Thread t_b	Incompatible views
1	$V(t)$	$\{x\},\{y\}$	$\{x\},\{y\}$	none
	$M(t)$	$\{x\},\{y\}$	$\{x\},\{y\}$	
2	$V(t)$	$\{x,y\}$	$\{x\},\{y\}$	$\{x\} = \{x,y\} \cap \{x\} \in M(t_a) \cap V(t_b)$
	$M(t)$	$\{x,y\}$	$\{x\},\{y\}$	$\{y\} = \{x,y\} \cap \{y\} \in M(t_a) \cap V(t_b)$
3	$V(t)$	$\{x,y\},\{x\},\{y\}$	$\{x\},\{y\}$	$\{x\} = \{x,y\} \cap \{x\} \in M(t_a) \cap V(t_b)$
	$M(t)$	$\{x,y\}$	$\{x\},\{y\}$	$\{y\} = \{x,y\} \cap \{y\} \in M(t_a) \cap V(t_b)$
4	$V(t)$	$\{x,y,z\}$	$\{x,y\},\{x\}$	none
	$M(t)$	$\{x,y,z\}$	$\{x,y\}$	

***Table 3.1**: Examples with* two *threads illustrating the principle of view consistency. The set of views generated by each thread, $V(t)$, is given in the first line of each example, the set of maximal views, $M(t)$, in the second line.*

A few examples help to illustrate the concept. Table 3.1 contains examples involving two threads. Note that the outermost brackets for the set of sets are omitted for better readability. Example 1 is the trivial case where no thread treats the two fields $\{x\}$ and $\{y\}$ atomically. Therefore there is no inconsistency. However, if thread t_a treats $\{x,y\}$ as a pair, and thread t_b does not, there is a conflict as shown in example 2. This even holds if the first thread itself uses partial accesses on $\{x\}$ or $\{y\}$, since this does not change its maximal view. Example 3 shows that case. Finally, example 4 illustrates the case where thread t_a uses a three-dimensional coordinate set atomically and thread t_b reads or updates different subsets of it. Since the subsets are compatible as defined in Section 3.3.2, there is no inconsistency.

#		Thread t_c	Thread t_d	Thread t_e	Incompatible views
5	$V(t)$ $M(t)$	$\{x,y\}$ $\{x,y\}$	$\{x\}$ $\{x\}$	$\{x\},\{y\}$ $\{x\},\{y\}$	$\{x\}=\{x,y\}\cap\{x\}$ $\{y\}=\{x,y\}\cap\{y\}$
6	$V(t)$ $M(t)$	$\{x,y\}$ $\{x,y\}$	$\{x\}$ $\{x\}$	$\{y\}$ $\{y\}$	none
7	$V(t)$ $M(t)$	$\{x,y\},\{x\},\{y\}$ $\{x,y\}$	$\{y,z\},\{y\},\{z\}$ $\{y,z\}$	$\{z,x\},\{z\},\{x\}$ $\{z,x\}$	none
8	$V(t)$ $M(t)$	$\{x,y\},\{x\},\{y,z\}$ $\{x,y\},\{y,z\}$	$\{y,z\},\{y\},\{z\}$ $\{y,z\}$	$\{z,x\},\{z\},\{x\}$ $\{z,x\}$	$\{y\}=\{y,z\}\cap\{y\}$ $\{z\}=\{y,z\}\cap\{z\}$

***Table 3.2**: Examples with* three *threads illustrating the principle of view consistency.*

Table 3.2 shows four cases with three threads. The first entry, example 5, corresponds to the first three threads in Figure 3.4. There, thread t_e violates the assumption of t_c about the atomicity of $\{x,y\}$. Example 6 shows a "fixed" version, where t_e does not access $\{x\}$. More complex circular dependencies can occur with three threads. Such a case is shown in example 7. Out of three fields, each thread only uses two, but these two fields are not always used atomically. Because the accesses of any thread only overlap in one field with each other thread, there is still no inconsistency. This example only requires a minor change, shown in example 8, to make it faulty: Assume the third view of t_c were $\{y,z\}$ instead of $\{y\}$. This would contribute another maximal view $\{y,z\}$, which conflicts with the views $\{y\}$ and $\{z\}$ of t_d.

3.3.4 Soundness and Completeness

By collecting views, this approach tries to infer what the developer intended when writing the multi-threaded code. The sets of shared fields which must be accessed atomically are not directly available in the program code. Therefore views are used to detect the most likely candidates, or maximal views. View consistency is used to detect violations of accesses to these sets. The underlying assumption behind the algorithm is that an atomic access to a set of shared fields is an indication of atomicity. Under this assumption, a view consistency violation indicates a high-level data race.

However, sets of fields may be used atomically even if there is no requirement for atomic access. Therefore, an inconsistency may not automatically imply a fault in the software. An inconsistency that does not correspond to a fault is referred to as a *false positive* (spurious warning). Similarly, lack of a reported inconsistency does not automatically imply lack of a fault. Such a missing inconsistency report for an existing fault is referred to as a *false negative* (missed fault).

False positives are possible if a thread uses a coarser locking than actually required by operation semantics. This may be used to make the code shorter or faster, since locking and unlocking can be expensive. Releasing the lock between two independent operations requires splitting one `synchronized` block into two blocks.

False negatives are possible if all views are consistent, but locking is still insufficient. Assume a set of fields that must be accessed atomically, but is only accessed one element at a time by every thread. Then no view of any thread includes all variables as one set, and the view consistency approach cannot find the problem. Another source of false negatives is the fact that a particular (random) run through the program may not reveal the inconsistent views, if the corresponding code sections are not executed even once.

The fact that false positives are possible means that the solution is not sound. Similarly, the possibility of false negatives means that the solution neither is complete. This may seem surprising, but actually also characterizes the Eraser low-level data race detection algorithm [SBN$^+$97] implemented in the commercial Visual Threads tool [Har00], as well as the deadlock detection algorithm also implemented in the same tool. The same holds for the similar algorithms implemented in JPaX. For Eraser, it is very hard to determine automatically whether a warning is a false positive or a false negative [Bur00]. Furthermore, it is an unsolved problem to prove soundness and completeness properties about the Eraser algorithm. In real software programs, there are always situations where having program threads use inconsistent values is acceptable. For example, a monitoring thread may just "sample" a value at a given time; it is not crucial that this value is obtained with proper synchronization, because it does not have to be up-to-date.

The reason for the usefulness of such algorithms is that they still have a much higher chance of detecting an error than if one relies on actually executing the particular interleaving that leads to an error, without requiring much computational resource. These algorithms are essentially based on turning the property to be verified (in this case: no high-level data races) into a more testable property (view consistency). This aspect is discussed in more detail in [BH03] in relation to deadlock detection.

3.4 Summary

Data races denote concurrent access to shared variables with insufficient lock protection, leading to a corrupted program state. Classical, or low-level, data races concern accesses to single fields. The notion of high-level data races deals with accesses to sets of related fields which should be accessed atomically.

View consistency is a novel concept considering the association of variable sets to locks. This permits detecting high-level data races that can lead to an inconsistent program state, similar to classical low-level data races. Experiments, which are described in Chapter 9, have shown that developers follow the guideline of view consistency to a surprisingly large extent. Thus view consistency captures an important idea in multi-threading design.

4

Using Block-local Atomicity to Detect Stale-value Errors

Data races do not cover all kinds of concurrency errors. This chapter presents a data-flow-based technique to find stale-value errors, which are not found by low-level and high-level data race algorithms. Stale values denote copies of shared data where the copy is no longer synchronized. The algorithm to detect such values works as a consistency check that does not require any assumptions or annotations of the program. It has been implemented as a static analysis in JNuke. The analysis is sound and requires only a single execution trace if implemented as a run-time checking algorithm. Being based on an analysis of Java bytecode, it encompasses the full program semantics, including arbitrarily complex expressions. As will be shown, related techniques are more complex and more prone to over-reporting.

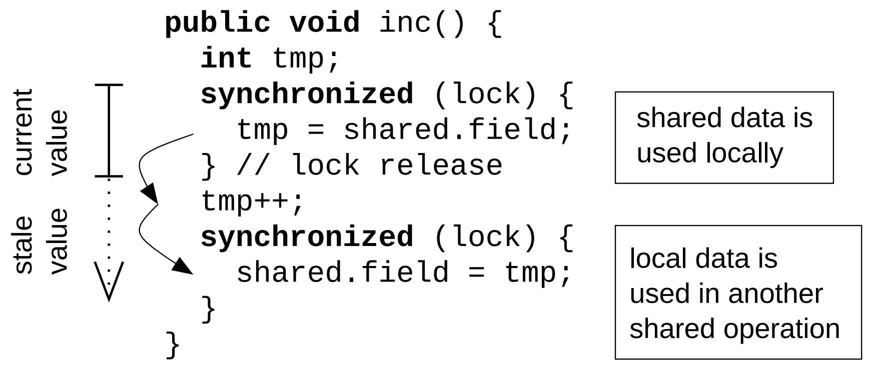
Section 4.1 gives the intuition behind our algorithm. Section 4.2 formalizes the property to be checked, and Section 4.3 extends the algorithm to nested locks and recursion. The precision of this new algorithm is discussed in Section 4.4. Section 4.5 summarizes this chapter. Related work is discussed in Chapter 10, which includes other concurrency properties.

4.1 Our Data-flow-based Algorithm

The intuition behind this algorithm is as follows: Actions using shared data, which are protected by a lock, must always operate on *current* values. Shared data is stored on the heap in shared fields, which are globally accessible. Correct synchronization ensures that each access to such shared fields is exclusive. Hence shared fields that are protected by locks always have current values.

Shared values are accessed by different threads and may be copied when performing operations such as an addition. Storing shared values in local variables is common practice for computing complex expressions. However, these local variables retain their original value even when a critical (`synchronized`) region is exited; they are not updated when the global shared field changes. If this happens, the local variable will contain a *stale* value [BL02] which is inconsistent with the global program state.

Figure 4.1 shows how such an error is discovered by our new algorithm. A shared field is assigned to a local variable `tmp`, which is again used later, outside the `synchronized`



```
public void inc() {
  int tmp;
  synchronized (lock) {
    tmp = shared.field;
  } // lock release
  tmp++;
  synchronized (lock) {
    shared.field = tmp;
  }
}
```

current value
stale value
shared data is used locally
local data is used in another shared operation

***Figure 4.1**: Intuition behind our algorithm.*

block. The value of the shared field thus "escapes" the `synchronized` block, as indicated by the first arrow. While the lock is not held, other threads may update the shared field. As soon as the original thread continues execution (in computations, method calls, or assignments), effects of its actions may depend on a stale value. The second arrow indicates the data flow of the stale value.

Note that we use an uncommon notion of escape analysis. Usually escape analysis is concerned with *references* escaping from a certain scope or region [Bla99, BH99, CGS$^+$99, WR99]. In our algorithm, escaping *values* are considered, not just references, and the scope of interest are `synchronized` blocks.

The lack of a single synchronization scope for the entire sequence of operations is responsible for having stale values in this example. Hence, if the entire method had been `synchronized`, it would have consisted of a single block, which would have executed atomically. Our algorithm uses existing `synchronized` blocks to verify whether shared data escapes them. It therefore requires synchronization to be present for accesses to shared fields. The assumption that each field access itself is properly guarded against concurrent access can be verified using Eraser [SBN$^+$97].

Like Eraser and the high-level data race algorithm [AHB03], our new algorithm only requires one execution trace if implemented as a run-time verification algorithm. Furthermore, the property is entirely thread-local. A static implementation of the algorithm is therefore straightforward. If aliases of locks are known, method-local static analysis can verify the desired property for each method while requiring only summary information about other methods. Static analysis has the advantage of being able to symbolically examine the entire program space.

A dynamic analysis on the other hand has precise information about aliases of locks. However, a particular execution typically cannot cover the entire behavior of a program. Even though the probability of actually observing erroneous states in a multi-threaded program is small, dynamic analysis algorithms are often capable of detecting a potential error even if the actual error does not occur [AHB03, SBN$^+$97]. The reason is that the property which is checked against (such as locking discipline) is stronger than the desired property (e.g. the absence of a data race). The algorithm presented here also falls into that category.

4.2 Formalization of Our Algorithm

This section gives a precise formalization of our algorithm. The algorithm is explained without going into details about nested locks and method calls. These two issues are covered in the next section.

In Java, each method invocation frame contains an array of variables known as its *local variables* and a fixed-size stack holding its *stack variables.* These two kinds of variables are always thread-local [GJSB00]. Both kinds of variables will be referred to as registers r. A shared field f will denote a field of a dynamic object instance which is accessed in a shared context, using lock protection.

A *monitor block* encompasses a range of instructions: Its beginning is the lock acquisition (`monitorenter`) of a new lock. Its end is marked by the corresponding lock release (`monitorexit`). It is assumed that lock acquisitions and releases are nested as required by the Java semantics [GJSB00]. Non-nested locking mechanisms can be treated as well, as described in Section 2.3.4. Each monitor block has a unique ID b distinguishing individual lock acquisitions. Reentrant lock acquisitions and releases have no effect on mutual exclusion and are ignored.

A register is *shared* when it contains the value of a shared field f and *unshared* otherwise. When shared, the monitor block in which the shared value originated is also recorded. The *state* $\mathrm{s}(r) = \langle sh, b\rangle$ of a register comprises its shared status $sh \in \{0, 1\}$ and its monitor block ID b. The *current monitor block b_{curr}* is the block corresponding to the latest non-reentrant lock acquisition.

At the beginning of execution, all registers are unshared. There are two possibilities to obtain a shared value: First, a `getfield` instruction within a monitor block will produce a shared value. Second, a method invocation may return a shared value. Shared values have state $\langle 1, b_{curr}\rangle$. We will use two auxiliary functions returning the first and second part of a state s, respectively: $\mathrm{shared}(s)$ and $\mathrm{monitorblock}(s)$.

Each assignment of a value will carry over the state of the assigned value. Operations on several values will result in a shared value if any of the operands was shared. A register r is *used* by an instruction *i*, $r \in \mathrm{used}(i)$, if it is read by it. If r is a stack element, the corresponding stack argument is consumed when it is read, according to the Java bytecode semantics [GJSB00]. If r is a local variable, reading it does not have any further effect. Note that this definition of usage includes expressions and arithmetic operations. In expression `tmp2 = tmp1 + tmp0`, the result `tmp2` is shared if any of the operands `tmp1` or `tmp0` is shared.

A *stale value* is a value of a shared register that originated from a different monitor block than where it is used. This can be formalized as follows: A program uses no stale values iff, for each program state and each register r used by current instruction i, the following holds: monitor block of that register, $s(r)$, must be equal to the current monitor block:

$$\forall i, r \cdot (r \in \mathrm{used}(i) \wedge \mathrm{shared}(s(r)) \rightarrow (\mathrm{monitorblock}(s(r)) = b_{curr})$$

If a single operation uses several shared values with different monitor block IDs b, then at least one of them must be a stale value. This property is then violated, and the result of that operation is again a shared value.[1] We will refer to this property as *block-local atomicity.* If it holds for the entire program, then actions based on shared data will always operate on current data.

4.3 Extension to Nested Locks and Recursion

The assumption behind dealing with nested locks is that any locks taken beyond the first one are necessary to ensure mutual exclusion in nested `synchronized` blocks. This is a natural assumption arising from the program semantics: nested locks are commonly used to access shared fields of different objects, which use different locks for protection. Let l_{outer} and l_{inner} denote an outer and an inner lock, respectively. Assume a thread acquires l_{inner} when already holding l_{outer}. It then accesses a shared field f holding both locks. After releasing l_{inner}, the shared field is no longer protected by that nested lock and may thus be updated by other threads. Any usage of stale values outside the nested lock l_{inner} violates block-local atomicity.

Low-level data race detection like Eraser misses this error, because each field access operation is properly protected. Block-local atomicity detects that the shared value becomes stale outside the inner monitor block. The following treatment of nested locks covers such errors: The algorithm declares a separate monitor block for each nested lock. If any operation outside the inner block uses a shared value such as f, this will be detected by the consistency check defined in the previous section.

Using the shared data from f outside the inner block would only be safe if l_{inner} was superfluous: If l_{inner} was always used only in conjunction with l_{outer}, then l_{inner} would not contribute to protection against concurrent access. Instead the extra lock would constitute an overhead that should be eliminated, and the warning issued by our algorithm can help to identify this problem.

Because a new monitor block is used with each lock acquisition, the total number of locks held when acquiring a new lock l_{inner} is not relevant. Thus the idea generalizes to a set of outer locks L_{outer} instead of a single outer lock l_{outer}.

When dealing with *method calls,* only the effect of data flow and `synchronized` blocks has to be considered. In run-time analysis, this is implemented trivially as method calls do not have to be treated specially.[2] In static analysis, method calls are essentially inlined, using only summary information of callees. If no new synchronization is used by the callee, the method call has no special effect and behaves like a local operation. Otherwise, if a new (non-reentrant) lock is used by the callee, the return value will be shared with a new unique monitor block ID. Hence the return value of a call to a `synchronized` method is shared, unless the caller itself used the same lock during the call, which would make the inner lock merely reentrant.

Because of this treatment of nested locks, handling inter-method data flow is quite natural and very efficient. The analysis does not have to consider calling contexts other

[1] In our implementation we marked the result of any such operation as unshared. The operation already generates a warning. Resetting the state of that register prevents generating more than one warning for any stale value.

[2] A call to a synchronized method is treated like a block using `synchronized(this)`.

than the lock set held. A context-insensitive variant of the algorithm is easily created: One can simply assume that any locks used in called methods are distinct. The algorithm will still be sound but may emit more false warnings. The same assumption can be used if the effect of a called method is unknown, e.g. when a method is native.

Finally, in a static implementation of the algorithm, the temporary lock release in a `wait()` operation has to be modeled explicitly [BL02]. For run-time verification in JNuke [ASB+04], the lock release event is implicitly generated by its run-time verification API [ASB+04].

4.4 Precision and Limitations of Our Algorithm

If a program is free of data races, our algorithm finds all stale values but may issue false warnings. Atomicity-based approaches, including this one, are sometimes too strict because certain code idioms allow that the globally visible effect of a non-atomic operation corresponds to an atomic execution. *Serializability* is a more precise property, but even non-serializable programs can be correct.

4.4.1 Soundness and Completeness

Our algorithm assumes that no low-level data races are present. This kind of error can be detected by algorithms like Eraser [SBN+97]. If a program is free of (low-level) data races then our static analysis algorithm is sound; no faults are missed. In a static approximation of this analysis, however, the alias information about locks is not always known. If one assumes each lock acquisition utilizes a different lock, the algorithm remains sound but becomes more prone to overreporting. Furthermore, soundness is also preserved if it is assumed that any unknown method called returns a shared value belonging to a monitor block of its own. If the algorithm is implemented dynamically, then soundness depends on the quality of a test suite and can usually not be guaranteed.

False positives may be reported if too many distinct monitor blocks are created by the analysis. A possible reason is the creation of more locks than actually necessary to ensure mutual exclusion. However, assuming that synchronization primitives are only used when necessary, then the algorithm will not report false positives, in the following sense: each reported usage of a shared value in a different monitor block actually corresponds to the use of a stale value.

4.4.2 Comparison to Previous Atomicity-based Approaches

Block-local atomicity is more precise than method-level atomicity as used by previous approaches [FF04, FQ03, vPG03, WS03]. These approaches check for the atomicity of operations and assume that each method must execute atomically. This is too strict. Non-atomic execution of a certain code block may be a (welcome) optimization allowing for increased parallelism. Our algorithm detects whether such an atomicity violation is benign or results in stale values. Furthermore, it does not require assumptions or annotations about the desired scope of atomicity.

Our algorithm uses data flow to decide which regions must necessarily be atomic. At the same time, the analysis determines the size of atomic regions. Therefore block-local atomicity reports any errors found by earlier atomicity-based approaches but does not report spurious warnings where no data flow exists between two separated atomic regions.

```
void sensorDaemon() {
  while (true) {
    synchronized (lock) {
      value = shared.field; // acquire latest copy
      value = func (value);
      shared.field = value; // write back result
    }
    sleep(1000); // wait
  }
}
```

Figure 4.2: *The importance of data flow analysis for* `synchronized` *blocks.*

Figure 4.2 shows an example that illustrates why our algorithm is more precise. Assume the program consists of several threads. The one shown in the figure updates a shared value once a second. For instance, it could read the value from a sensor and average it with the previously written value. It then releases the lock, so other threads can access and use this value. A reduction-based algorithm will (correctly) conclude that this method is not atomic, because the lock is released during each loop iteration. However, as there is no data flow between one loop iteration and the next one, the program is safe. In fact, an atomic implementation which does not release the lock inside the loop would result in a livelock.

Block-local atomicity analyzes the program correctly and does not emit a warning. Other approaches which assume that each method must be atomic will report a false positive [FF04, FQ03, vPG03, WS03]. While it is common in practice that the scope of synchronization coincides with method boundaries, this is not always the case. Our algorithm offers better precision without extra complexity.

4.4.3 Limitations of Atomicity-based Approaches

The strict semantics of atomic operations and block-local atomicity are not always required for a program to be correct. This creates a potential for warnings about benign usages of stale values. An example is a logging class using lax synchronization: It writes a local copy of shared data to its log. For such purposes, the most current value may not be needed, so block-local atomicity is too strict.

Finally, conflicts may be prevented using higher-level synchronization. For instance, accesses can be separated through thread `start` or `join` operations [Har00]. This is the most typical scenario resulting in false positives. Note that other atomicity-based approaches will always report a spurious error in such cases as well. The segmentation algorithm can eliminate such false positivies [Har00].

4.4.4 Serializability

Even without higher-level synchronization, block-local atomicity is sometimes too strong as a criterion for program correctness. Serializability is a weaker but still sufficient criterion for concurrent programs [Fla04]. Nevertheless, there are cases involving container structures where a program is correct, but neither atomic nor serializable. Consider Figure 4.3, where a program reads from a buffer, performs a calculation, and writes the result back. Assume `buffer.next()` always returns a valid value, blocking if necessary. After a value has been returned, its slot is freed, so each value is used only once. Method `buffer.add()` is used to record results. The order in which they are recorded does not matter in this example.

The reason why the program is correct is because the calculation does *not* depend on a stale *shared* value; "ownership" of the value is transferred to the current thread when it is consumed by calling `buffer.next()`. Thus the value becomes thread-confined and is no longer shared. This pattern is not captured by our data flow analysis but is well-documented as the "hand-over protocol" [Lea99]. It could be addressed with an extension to the approach presented here, which checks for thread-local confinement of data.

```
public void work() {
  int value, fdata;
  while (true) {
    synchronized(lock) {
      value = buffer.next();
    }

    fdata = f(value); // long computation

    synchronized(lock) { // Data flow from previous block!
      buffer.add(fdata); // However, the program is correct:
    }                    // The buffer protocol ensures that the
  }                      // returned data remains thread-local.
}
```

***Figure 4.3**: A correct non-atomic, non-serializable program.*

4.5 Summary

We have presented a data-flow-based algorithm to detect concurrency errors that cannot be detected by low-level [SBN$^+$97] or high-level [AHB03] data races. Previous atomicity-based approaches were entirely based on the atomicity of operation sequences, but ignored data flow between `synchronized` blocks [FF04, vPG03, WS03]. This results in cases where correct non-atomic methods are reported as faulty. The algorithm presented in this chapter detects stale values [BL02]. This conceptually simpler and more precise property captures data flow between `synchronized` blocks. The property can be checked in a thread-modular, method-local way. It can be implemented as a static analysis or as a run-time checking algorithm.

5 Approaches to and Limitations of Run-time Verification

A program is typically tested by manually creating a *test suite,* which in turn is a set of *test cases.* An individual test case is a description of a single test input to the program, together with a description of the properties that the corresponding output is expected to have [Mye79]. Tools exist to make creation and evaluation of such a test suite fairly efficient [LF03].

Test cases are typically written as additional program code for the system under test. This has two major limitations: Code redundancy and limited testing capabilities. The nature of input generation and output evaluation code leads to a lot of shallow code that often instantiates variations of the same data structure. Such code typically contains redundancies that makes it hard to maintain unit tests [vDMvdBK01]. Furthermore, only observable output, such as return values, global data structures, or file contents can be verified; observing the internal state is not directly possible with testing.

Run-time verification (RV) takes care of *observation* of the system under test. Properties that have to hold for specific inputs or the entire program can be specified in a more generic way than with testing. Furthermore, the entire internal state of the program is available for checking properties, which allows more efficient modelling of such. Finally, higher-level constructs such as contracts [Mey97, vdBJ01], data races [AHB03, SBN$^+$97], or linear temporal logic (LTL) properties [Pnu77] are typically offered by run-time verification packages, allowing to express program properties more succinctly, improving their maintainability.

Note that in run-time verification of multi-threaded software, only a single run produced by a single schedule is usually taken for analysis. This contrasts with software model checking [VHB$^+$03], where all necessary schedules are enumerated to analyze the entire possible behavior of an application.

Typically run-time verification is still used in conjunction with manual testing. The test suite is applied to the program, which executes and generates an execution trace. The observer module checks the trace against a given set of properties. Hence, it takes the execution trace and the set of properties as input. This chapter describes how the execution trace can be generated and what kind of properties can be verified with existing tools.

This chapter is organized as follows: Section 5.1 defines commonly used terms that are not yet all consistently used in the community. Inherent advantages and limitations of each approach are given in Section 5.2. Different types of properties to be verified are shown in Section 5.3. Section 5.4 describes existing tools and what approaches they use. Practical experience with some of these tools is described in Section 5.5. Section 5.6 shows what inherent limitations exist in RV. Section 5.7 concludes.

5.1 Definitions

5.1.1 Preliminaries

The *system under test* (SUT) is the software to be analyzed. It may encompass an entire application or running process, or a set of smaller units, typically public classes, that can be tested individually using unit testing. The test suite is part of the SUT and comprises a set of individual tests; the *test harness* runs the test suite with given input data, and thus attempts to simulate the behavior of the SUT after deployment [LF03]. Normally the test suite is fully automated, with deterministic input. The *environment* includes the *execution environment* (for Java, a virtual machine), system libraries and a scheduler that is used to determine which thread to run in a multi-threaded environment. By default, the scheduler is assumed to be non-deterministic and cannot be influenced by the SUT or its test harness.

The *system state* encompasses everything that is part of the SUT, i.e., directly influenced by its execution, but not its test harness. This view is directly applicable in practice when the test harness is an external application, such as a shell or expect [Lib91] script. In unit testing, the test harness is usually linked to the SUT and therefore executes as one application [LF03]. However, since the test harness is a separate library, independent of the test suite, state changes inside it can be ignored and seen as invisible state transitions. Therefore the system state includes public and private data of the SUT and also state information that is not directly available to the SUT, such as the program counter, current thread ID, or lock set. Indirectly manipulated data is managed by the execution environment, which may allow partial access to this data to the SUT. *External components* of the system state are a subset of the entire state, including all data that is visible by the environment, such as shared memory [TRY$^+$87], or public data in the case of unit testing.

The system under test starts in an initial state s_0 and then executes, producing a transition through a series of system states $\langle s_0, \ldots, s_n \rangle$. A system *trace* is defined as a sequence of events emitted by the SUT $\langle e_0, \ldots, e_m \rangle$. An event e_i corresponds to a change in the external component of a system state, e.g. side-effects of a system call such as I/O. Note that a single event e_i may correspond to several states, since state transitions may be invisible from outside the system. However, each system state transition from one state s_j to another state s_k can emit at most one event.[1] Program analysis usually ignores most events and is only concerned with a small subset of them, e.g. all modifications to a certain file.

[1]This is conceptually the case but may not be literally true for an implementation. For instance, JNuke generates two events for execution of a `monitorenter` bytecode: An event for bytecode execution and an event for lock acquisition [AB05a]. However, these events are simply two views of the same system event, and therefore can be conceptually seen as one.

The trace generated by the default execution environment is usually not sufficient for a meaningful program analysis. Therefore a *verification environment* complements the execution environment. The task of the verification environment is to interface with system behavior, allowing the observation of a trace where previously internal events are *externalized.* Such events can be part of the execution environment (such as thread context switches) or the SUT (such as field accesses).

5.1.2 Trace generation

Traces can be generated in several ways: by *instrumentation,* which changes the code of the SUT; by *wrappers,* which add event generation to external (system) libraries; or by a *customized execution environment,* which replaces a conventional one by a new one which generates events.

Code instrumentation

Code instrumentation achieves event generation by injecting additional code into the SUT. The additional code is responsible for event generation but must not interfere with the SUT. Instrumentation is thus defined as a set of changes C to the SUT, based on a rule set R, such that C does not alter the internal functionality of the SUT. Instead, it generates events that are transparent to the SUT but visible to the verification environment. This means that C must not produce any side-effects in the SUT and must not generate events that do not correspond to a state change in the SUT.

Rule set R contains pairs of predicates P and actions A. Whenever a predicate evaluates to true, its corresponding action defines the change $c \in C$ to apply to the SUT. Note that P in this chapter is only concerned with rules that can be evaluated statically (at instrumentation time). If event generation is dependent on run-time data, then the dynamic part of it is part of A, reflected in the inserted code in C [GH03].

Instrumentation can happen at source code or byte (machine) code level. Source code instrumentation (obviously) takes place prior to compilation while bytecode instrumentation can be performed as a post-compilation stage or at load time. The instrumentation process $\text{instr}(R, SUT)$ takes R and the SUT as input, producing C. Each rule can lead to any number of changes, i.e., $|R| \geq 1, \forall r \in R \cdot |\text{instr}(\{\text{r}\}, SUT)| \geq 0$. In general, it is desirable to keep R small to ensure its correctness, in particular, non-interference of C with the normal functionality of the SUT.

The simplest way of instrumentation is to add manual changes to the program source code, which can be expressed as $|R| = |C| = 1$. In that sense, adding debugging output to the program to generate a trace of events that would otherwise be invisible (without a debugger) constitutes an ad-hoc form of run-time verification. However, even in ad-hoc approaches, simple scripts (such as search/replace patterns) are often used, corresponding to the common case $|R| \geq 1, |C| \geq |R|$.

Source code instrumentation without the help of tools quickly becomes impractical. Therefore a tool-based approach is usually taken. Two directions are prevalent for source code instrumentation: Aspect-oriented programming (AOP), implemented in toolkits such as AspectJ [KHH+01], and compiler back ends, for instance, array bounds checking for GCC [JK97]. In AOP, *P* corresponds to the *concerns* used, and *A* corresponds to the *advice.*

Compiler back ends typically operate on an abstract syntax tree of the program, which is then analyzed by an instrumentation module. Such an analysis can therefore use special source-level constructs, but the result of the analysis is not necessarily modified source code. *C* could instead contain intermediate code which is then included in the generated executable SUT at a subsequent machine code generation stage.

Because the syntax of modern programming languages is fairly complex, many instrumentation approaches work on compiled code, which can be bytecode or machine code. The process of code instrumentation is the same. The only difference to source code instrumentation is that the actual code changed is not directly human-readable. Thus rule set *R* typically contains more complex expressions (requiring some assembly-like language or data structures representing the low-level code), even though the resulting code modifications are usually more concise. Code instrumentation packages for Java bytecode include the BCEL [Dah03] and JTrek [Coh01].

Wrappers

Wrappers follow the Decorator pattern [GHJV95] and add extra functionality to objects, or, in this context, library calls. Library functionality is extended with event generation code and sometimes even replaced with an entirely new implementation. The libraries are assumed to be part of the execution environment, not the application itself. Otherwise, code replacement would occur inside the SUT and correspond to instrumentation. Wrapping has the advantage that the border between modified and original code is clearly defined. The changes all lie outside the application and can be tested separately. Furthermore the provided functionality can be used for several SUT without a need to change them.

Wrapper code can replace the original API transparently if the execution environment allows for exchanging libraries at load time [Roy02]. The additional library code could already be part of the library and be activated by a special API call or environment variable [JK97]. Alternatively, the compilation or linking process of the SUT may have to be adapted in order to use a new library, which often uses the original one underneath to provide its functionality [Roy02].

Custom execution environment

A custom execution environment replaces the entire execution environment with a new implementation. This new implementation can either emit a trace directly [ASB+04] or allow for on-the-fly code instrumentation (instrumentation at load time rather than at compile time) [NS03]. The specialized execution engine allows to monitor low-level events such as changes to certain memory locations. It is usually combined with wrappers to allow generating high-level events more easily, e.g. events concerned with certain library calls.

5.1.3 Trace monitoring

Monitoring a trace that is generated at run-time includes receiving the event trace and evaluating it against a given set of properties. This task can be performed in two possible ways: on-the-fly and off-line. In on-the-fly monitoring, the monitoring code is embedded in the SUT or the execution environment and executed inside the application immediately after an event is generated. Off-line monitoring, on the other hand, only embeds event logging or event transmission code in the SUT. Events are either written to a file or directly communicated to an analyzer application or thread. The analyzer, also called observer [GH03], evaluates the properties either during execution of the SUT or after its termination.

Different ways of distinguishing trace monitoring approaches have been proposed by Chen et al. [CdR04]. They define "in-line" and "out-line", "on-line" and "off-line" (post-mortem) monitoring. In that terminology, in-line monitoring means that the monitor runs in the same resource space as the program, while out-line monitoring refers to the fact that monitoring takes place in another process or on another processor. The terms "on-line" and "off-line" then refer to whether monitoring occurs while running the SUT or afterwards. In our terminology, "on-the-fly" monitoring refers to in-line, on-line monitoring, while off-line monitoring refers to out-line monitoring.

We argue that our definition is more practical; the distinction whether monitoring takes place during or after execution of the SUT is not important for the design of the system. In out-line monitoring, the means of communication employed between the SUT and the monitoring process are easily exchangeable, being commonly pipes, files, or sockets. The choice usually depends on whether the trace should be stored for later reference and on performance characteristics. Whether the monitoring process receives input data before the SUT finishes may even be scheduler-dependent. Finally, the combination "in-line, off-line" cannot exist, since an in-line monitoring process always runs together with the SUT. Chen's definition of an "in-line, off-line" monitor is a monitor that merely logs relevant states; actual trace evaluation is then performed on the log, off-line [CdR04]. However, such a system is a combination of *two* monitors, on-the-fly (for logging events) and off-line (for evaluation), as discussed below. Therefore, only three out of the four combinations make sense. We believe the distinction between on-line and post-mortem monitoring is possible, but not important. Thus the three variants can be reduced to on-the-fly (in-line, on-line) and out-line, which we call off-line.

Chen [CdR04] furthermore distinguishes synchronized on-line monitors from asynchronous ones. Synchronized monitors block execution of the SUT until an event is evaluated. This is relevant for in-line monitors, where an asynchronous monitor runs in a separate thread. This may be advantageous if several processors are available but does not otherwise affect the design of the system. Event data received by such an asynchronous monitor must not contain data writeable by the SUT since this can lead to data races. Therefore an asynchronous monitor uses its own event data (copies of data of the SUT) and thus effectively becomes an off-line (out-line) monitor. Chen also uses a second meaning for synchronization, extending it to evaluation of entire properties rather than single events. Under that definition, synchronized monitors can be rather restricted for formulae such as future-time LTL, and an asynchronous variant is obviously more efficient [CdR04]. This chapter does not distinguish this property further as it is covered by our on-the-fly/off-line notation in sufficient detail.

5.2 Benefits and Limitations of Each Approach

5.2.1 Code instrumentation

Instrumentation has the advantage that it requires no changes in the execution environment. The verification environment only modifies the SUT and possibly runs extra processes together with the execution environment. Therefore even proprietary execution environments can be used, and any optimizations available (such as JIT compilers) can be fully taken advantage of.

The main disadvantage of instrumentation is that it does not have full state access to the system. For instance, in Java, the thread ID is a private field. This made sense when Java was originally designed, because no application should be dependent on thread ID values. However, for event generation, the thread ID is essential. Therefore an instrumentation-based approach has to construct its own thread ID map by mapping thread objects to integers [AHB03]. This duplication is unavoidable because current VMs do not allow ignoring the "private" attribute.

Another disadvantage of instrumentation is that due to the architecture of byte or machine code, the instrumentation specification can be quite complex. The instrumented code may have to allocate extra local variables to store register data because these registers are needed to create an event record. Furthermore, the verification process is somewhat cumbersome to add to the build tool chain, because it usually requires a post-compilation stage and changes in linking or in the order in which classes are loaded. The latter is necessary such that instrumented and unmodified object or class files can be used together.

5.2.2 Wrappers

The main advantage of wrappers is their conceptual elegance. The locations of event generation are outside the SUT and clearly defined: They always occur inside wrapped library calls and therefore always correspond to an external action of the SUT. The main disadvantage is the inability to access (even public) data of the SUT unless it is provided to a library call. If a wrapper for a certain method is supposed to be entirely transparent, it cannot take extra arguments that could provide more information for run-time verification; thus wrappers used in isolation are fairly limited. However, for specific purposes such as detection of memory leaks, they still are the most elegant approach. A considerable practical advantage of this technique is the fact that the development environment remains unchanged, requiring no additional steps tool chain [NS03, Roy02].

5.2.3 Custom execution environment

A custom execution environment has full access to the entire state and can monitor individual data structures and operations. Usually the number of different operations monitored is relatively small; therefore listeners to handle such events are not hard to write. Conversely, monitoring library calls is somewhat cumbersome in that scenario. The size of the library can be much larger than the number of primitive operations, making this approach, if used in isolation, less useful for such a task.

5.2.4 On-the-fly vs. off-line trace monitoring

On-the-fly monitoring has the advantage that it has full access to available data of the SUT. Data which is not provided in the event record itself may be obtained from the verification environment through callbacks [AB05a]. The drawback of on-the-fly monitoring is that any algorithmic computations of the verification algorithm occur inside the SUT and therefore slow down test execution.

This slow-down is the motivation for off-line monitoring, where the verification algorithm can run on another processor or even post mortem, after test execution. In addition to that, it is simple to distribute events to several analysis algorithms [ABG$^+$03]. On the other hand, off-line monitoring is restricted to data provided in the event record. Callbacks to the SUT are not possible since it continues to run in parallel with the analyzer after event generation, or may even have terminated when an event is processed. Because of this parallelism, only data from the current event is known to be consistent, while other event data may be stale. For example, the exact program location may only be reported from time to time, as generating an event for each line of code would cause too much of an overhead. This may lead to incorrect locations in reports generated. Off-line monitoring also requires a more complex architecture involving at least two processes, the SUT and the analysis application, and an I/O mechanism (such as files, pipes or sockets) to communicate events between the SUT and the analyzer. Therefore the testing environment becomes a lot more complex compared to running the unmodified SUT or using on-the-fly monitoring. The I/O required for off-line monitoring is also a major source of potential overhead [AHB03], the lack of which makes on-the-fly monitoring more light-weight and often faster.

5.2.5 Hybrid approaches

Because individual event generation approaches all have limitations, they are hardly ever used in isolation. Instead, a combination of two technologies usually eliminates the weaknesses of either approach while preserving their strengths. For instance, purify uses wrapping to monitor calls to the malloc memory library and instrumentation to check invididual memory operations (such as pointer dereferences) [HJ92]. JNuke uses wrappers for the Java foundation library and a custom environment to generate events for bytecode operations such as lock acquisitions [ASB$^+$04]. Valgrind relies on instrumentation and its VM for execution of machine code, replacing direct execution on hardware [NS03]: First, machine code is converted to valgrind-specific intermediate code. Then this code is instrumented and run in the VM of valgrind.

For trace monitoring, a combination of on-the-fly and off-line monitoring is thinkable but has, to our knowledge, not yet been implemented. If properties are complex enough to require off-line processing, a first on-the-fly stage may be used to reduce the data required for the second off-line phase. This would cut down on the data communicated and likely speed up analysis. However, not all monitoring algorithms are amenable to such a reduction.

5.3 Property Verification

Tested properties can either be written as black-box properties, without knowledge of the implementation, or as implementation-dependent white box properties [Mye79]. Run-time verification can cover both kinds of properties; therefore this chapter makes no distinction between them.

Run-time verification is by definition incomplete, because a test suite can never cover the entire behavior of a non-trivial SUT. Therefore it is not crucial for verification algorithms to be totally accurate. In practice, achieved accuracy is usually quite high, with few false alarms (false positives) and missed errors (false negatives). This sounds surprising because many faults only manifest themselves as failures when a particular schedule is executed, which is only rarely the case in practice. Run-time verification is successful because it usually does not try to find such failures directly. Instead, *stronger* properties than the failure itself are checked. A stronger property is usually independent of scheduling yet a strong indicator that the failure looked for can occur under a certain schedule [ABG+03]. An occurrence of the failure looked for almost always violates the verification property, but the reverse is not true. A violated property may even be benign and never lead to a failure. For instance, when looking for low-level data races, the Eraser algorithm does not directly check for data corruption but verifies locking discipline, the absence of which often but not necessarily leads to data races [SBN+97].

5.3.1 Generic properties

The Eraser algorithm is an example for a generic property that is applicable to a single class of faults: data races [SBN+97]. The success of such algorithms comes from the fact that such faults can appear in any program and are very hard to find using conventional means; thus the relatively large investment into buildling a fault-detection tool for a single class of faults is worthwhile. This branch of run-time verification is sometimes referred to as *algorithm-based* run-time verification. This thesis focuses on this branch.

Many successful algorithms that verify generic, hard-coded properties exist. It is usually agreed on that violation of such a property signifies a high potential of a critical failure. Such properties include memory corruption (illegal addresses, memory leaks etc. [HJ92, NS03]), low-level data races [SBN+97], high-level data races [AHB03] and stale values [ABH04]. Despite their success, such properties can only cover certain errors. In many cases, they need to be complemented with application-specific properties.

Generic properties properties take a high-level requirement (the absence of certain kinds of data corruption) and translate it into an implementation-specific low-level requirement (locking patterns). The initial requirement is a black-box property while the property checked against is a white-box property. This fine point is another reason why this distinction is rarely made in practice.

5.3.2 Application-specific properties

Properties specific to the SUT can be either expressed directly in the SUT, e.g. by using assertions, or outside, with the help of the verification environment. Many such properties can be expressed by *contracts* [Mey97]. Contracts have become a very widespread way of specifying external properties of public methods. In the Eiffel programming language, they are embedded in the language and the execution environment [Mey97]. Third-party tools can provide the same functionality for other languages [CL02, vdBJ01].

Temporal logics, such as LTL [Pnu77], allow to express certain safety and liveness properties succinctly. Efficient monitoring tools for certain subsets of temporal logics exist. They include Temporal Rover, which can monitor limited-future time and past-time LTL properties [Dru03], or JPaX for past-time LTL [HR02]. EAGLE is a framework for defining and implementing finite-trace monitoring logics, including not only past-time and future-time LTL but also other logics [BGHS04]. The term *specification-based* run-time verification refers to verification of application-specific properties.

5.3.3 Steering

An alternative approach is to use few or no correctness properties beyond test results which are already part of unit tests. Instead, *steering* is used to perturb the application schedule in a way that is most likely to provoke potential failures. The simplest way how this can be achieved is through random scheduling [Sto02]. After insertion of random delays at critical points, the environment is very likely to switch thread contexts in code sections that are normally executed atomically under the default schedule. Newer tools use heuristics to make this search for faults more efficient [BAEF03, FNU03]. Sometimes steering is also used to alter general program properties according to a user-defined script [KKL$^+$01].

5.3.4 Relation to testing

Above approaches all use a manually specified test suite to run the SUT, with the intention to achieve high behavioral coverage that allows run-time verification to detect a large number of failures in the system. Properties verified against are correct output (part of the unit test suite), generic and specific properties. Specific properties are commonly also written manually. Reasons for this large amount of manual work are the needed human insight into system-specific details and the fact that existing systems to specify system properties require languages that are close to programming languages in their expressive power [BS03]. Therefore it is common to use the programming language of the SUT to write test and output specifications [LF03].

Recent projects try to leverage the versatility of model ckecking to *generate* both the input and the property to be tested [ABG$^+$03]. The idea is to explore the behavior of the model of the SUT by exploring specific executions. Thus one obtains, for each execution trace, a valid program input, and an instance of each applicable model property, specific to that trace. The entire set of input/property specifications is then applied to the SUT, making test case generation and property specification automatic, once a model is given.

5.4 Existing Work

A variety of run-time verification tools for different languages and purposes exists. An overview of important current projects is shown in Table 5.1. As outlined in the previous section, the common approach in run-time verification is monitoring of (higher-level) properties. The majority of tools follows this direction. The alternative, steering in the form of schedule perturbation, is implemented by the tools ConTest [FNU03] and raceFinder [BAEF03].

In property monitoring, one can distinguish between hard-coded (but usually widely applicable) properties and user-defined, application-specific ones. The mcheck extension of the GNU C library [JK97], JNuke [ASB+04], mpatrol [Roy02], purify [HJ92], and valgrind [NS03] all monitor several generic properties. Tools written to monitor user-defined properties typically verify contracts [CL02, Mey97], LTL-based specifications [BGHS04, Dru03, HR01] or similar properties [KKL+01].

Code instrumentation, sometimes combined with wrapping, is the prevalent approach for trace generation; only JNuke [ASB+04] and valgrind [NS03] use a custom execution environment designed for trace generation and observation. Many tools still use off-line monitoring [BGHS04, HR01, KKL+01]; however, there is a trend towards the more efficient on-the-fly monitoring [Dru03].

5.5 Practical Experience

This section is an initial attempt to investigate benefits and drawbacks of each approach. Two questions are the focus:

1. Which approach makes it easier to implement new properties in the analyzer? This question can of course not be answered quantitatively.

2. What approach scales best? Compared are on-line vs off-line monitoring, and whether a high-level API impacts performance of verifying the SUT.

Figure 5.1 compares the architectures of the tools investigated. It emphasizes two aspects: the event monitoring technique (on-the-fly vs. off-line) and whether a high-level API exists. Tools have been chosen to represent each category. The main differences between the tools are that JNuke and valgrind use on-the-fly monitoring while JPaX evaluates its events off-line. Both JNuke and the new version of JPaX allow specification of events in a high-level programming language, while valgrind needs the verification to be part of the instrumentation specification. The diagram summarizes these design differences but does not include one other architectural difference: Both JNuke and valgrind use their own execution environment while JPaX uses code instrumentation. However,

Tool	Trace generation	Mon.	Properties
ConTest [FNU03]	none (schedule steering)	–	none (requiring test suite)
EAGLE [BGHS04]	bytecode instrumentation	off-l.	user-defined (in EAGLE logic)
Eiffel [Mey97]	compilation (contracts are integrated in Eiffel)	o-t-f	contracts
GNU C lib. [JK97]	wrapping, manual instrumentation	o-t-f	memory allocation
JML [CL02, vdBJ01]	bytecode instrumentation	o-t-f	contracts written in JML
JNuke [ASB$^+$04]	custom execution environment, wrapping	o-t-f	data races, stale values
JPaX [HR01]	bytecode instrumentation	off-l.	data races
MaC [KKL$^+$01]	bytecode instrumentation	off-l.	user-defined (scripting language)
MOP [CdR04]	source code instrumentation	both	LTL, contracts, user-defined
mpatrol [Roy02]	wrapping	o-t-f	memory allocation and C string function usage
purify [HJ92]	wrapping, machine code instrumentation	o-t-f	memory allocation and usage
raceFinder [BAEF03]	none (schedule steering)	–	none (requiring test suite)
Temporal Rover [Dru03]	source code instrumentation	o-t-f	LTL with additional constraints
valgrind [NS03]	custom environment, instrum. of interm. code	o-t-f	several “skins” with hard-coded properties (such as mem. check)

***Table 5.1**: Existing run-time verification tools. Monitoring approaches, where applicable, are categorized into “off-line” (off-l.) and “on-the-fly” (o-t-f).*

at the current stage of each project, this factor does not yet make the major difference in performance. Of course this difference makes the evaluation less precise. However, no single tool currently implements, to our knowledge, several approaches that could be used interchangeably; therefore the approaches cannot be compared on a single platform. Because of this, we had to resort to using different tools to compare different approaches.

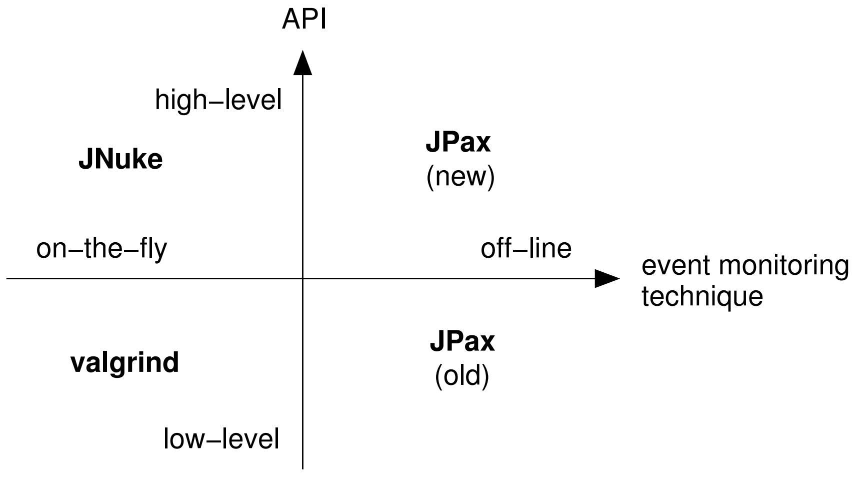


***Figure 5.1**: Comparison of tool architectures.*

5.5.1 Flexibility

The question about the flexibility of each design with respect to writing verification algorithms seems to be easier to answer. Both JNuke and JPaX offer listener-based APIs that make it fairly simple to write verification algorithms. As long as the necessary data is available, the user only has to write the logics of the verifier. The API of JPaX has mainly been designed during recent work [ABG+03], where it has been observed that verification engineers had to write a lot of common low-level code for each algorithm. The new architecture included such generic code, reducing development effort of verification algorithm to about half of what it was before. Performance had not been impacted by this design change [AHB03].

The other tool that is considered here, valgrind [NS03], uses its own intermediate code for both the SUT and the verification algorithm. Therefore writing such an algorithm requires learning and working with this code, which is certainly more difficult than working with a high-level programming language. Since the goal was to implement only a few generic verification algorithms, this drawback is not severe.

5.5.2 Scalability

A series of benchmarks was used to answer the second question and compare tools for detecting multi-threading errors in Java [ASB+04, HR01] with valgrind [NS03], which detects memory allocation errors when using C libraries incorrectly. As a base for experiments on Java code, benchmarks cited by recent publications were taken [ASB$^+$04, AB05a]. Tests for C programs consisted of executing a subset of the currently roughly 1800 unit tests for JNuke [ASB$^+$04], which corresponds to practical usage of valgrind by our group. The unit tests encompass a wide range of functions, from testing container classes to running a virtual machine for Java bytecode, to static analysis, and therefore represent a variety of programming problems.

Memory overhead could only be measured for JNuke and was usually within a factor of two, as summarized by Table 5.2. Execution times could be used to compare all tools. Table 5.3 shows the run-time overhead of different RV algorithms on JNuke, JPaX and valgrind. For JNuke and valgrind, "RV overhead" means the overhead of running the verification algorithm compared to using the custom VM with no verification algorithm running. Numbers given represent the ratio between the two execution times. The total overhead compares running the SUT under an (optimizing) virtual machine or directly on a processor, respectively. For JPaX, the ratio between executing the instrumented code compared to the uninstrumented version on an optimizing Sun VM is given. The best and worst ratios are given along with the geometric mean, so large benchmarks are given the same weight as smaller ones. It was not possible to precisely measure the amount of memory used by other verification environments, so these columns are omitted. The distinction between "RV overhead" and "execution overhead" shows that both the valgrind and the JNuke VM are still about 5 – 8 times slower in the average case than an optimized VM. Because of this, performance using run-time verification was compared both to the tool itself and to an "optimal" environment.

Tool	Best	Worst	Geometric mean
JNuke (stale values) [ABH04]	1.00	1.91	1.27
JNuke (Eraser) [ASB$^+$04]	1.00	4.53	1.88
JNuke (VC) [ASB$^+$04]	1.00	2.59	1.28

***Table 5.2**: Memory usage overhead for run-time verification in JNuke.*

Tool	RV overhead			Total overhead		
	Best	Worst	GM	Best	Worst	GM
JNuke (stale values) [ABH04]	1.19	5.64	2.15	2.90	555.63	28.29
JNuke (Eraser) [ASB$^+$04]	1.42	8.29	3.33	2.50	253.75	24.31
JNuke (VC) [ASB$^+$04]	1.57	12.82	4.14	1.83	240.88	30.28
JPaX (VC/Eraser) [AHB03]	N/A	N/A	N/A	4.50	1386.87	76.75
valgrind (memcheck) [NS03]	2.39	13.78	6.49	11.60	61.44	31.27

***Table 5.3**: Run-time overhead of various tools.*

It is obvious that the run-time overhead for JNuke is much smaller than the one for JPaX, which can mainly be attributed to off-line verification used in JPaX. In certain examples, this slowed down execution tremendously because data-intensive applications generated very large log files. Even for small applications, the overhead of JPaX was very considerable. The problem is somewhat ameliorated by the fact that the same log file could be re-used for several analysis algorithms. On the other hand, the same optimization is also possible in the other architectures: several event listeners could run in parallel. This would cut down the overhead of running the application itself for each analysis.

The poor performance of JNuke in some cases is mainly due to the lack of optimization in some data structures, where there have been no resources available so far to replace correct but slow placeholders. The smaller variation in the run times of valgrind indicate that it is already more mature and optimized.

Unlike valgrind, JNuke has not yet taken advantage of just-in-time compilation, which can improve the performance of Java bytecode significantly. It is not clear to what extent JNuke could be optimized to achieve a performance close to Sun's VM using no verification, because much of architecture of JNuke has been designed to allow for an elegant implementation of verification algorithms. Sun's VM would likely become more heavyweight and slower if run-time verification was a requirement. The same reasoning applies to valgrind, the "skinnable" design of which allows multiple verification algorithms to run on the same platform. This flexibility incurs some extra overhead. Overall, performance of JNuke and valgrind indicate that the custom execution environment is not severely penalized by the fact that not many (complicated) code optimizations are possible.

Both JNuke and valgrind have their own execution environment; however, while JNuke offers a high-level event API, the algorithms for valgrind have to be written in low-level code. Initial observations indicate no major drawback for having a rich API other than the fact that it makes the design of the VM less elegant. So far, however, the number of "API hooks" in the JNuke VM is small, and does not impact its maintainability.

In trace observation, on-the-fly verification certainly pays off for simpler, generic properties. The simpler architecture incurs no communication overhead, compensating for additional algorithmic computations added to the execution of the SUT.

5.6 Capabilities and Limits

> 'Program testing can be used to show the presence of bugs, but never to show their absence!' [Dij72].

This dogma has stigmatized testing for a long time, despite its success in industry. The reason behind this statement is the fact that testing only executes a small subset of all program behaviors [Mye79]. It can help to increase confidence in a system, but never fully verify it. Nonetheless, it is the prevalent technology used in practice because testing is simple and effective at uncovering certain failures [Pel01]. Concurrent programming, though, threatens to render testing futile, since thread scheduling introduces a source of non-determinism that is outside the control of a programmer.

In the area of these hard-to-find concurrency faults, run-time verification can be the crucial component to bring test execution back to a level of confidence that can be obtained for sequential programs. By obeying certain design guidelines concerning concurrent data access, the absence of certain concurrency failures can be ensured [ABG+03]. Such guidelines may prevent certain optimizations but are typically acceptable in practice, as most programs already fulfill them in order to avoid errors. It certainly is not very constraining to use sufficient locking to avoid low-level and high-level data races [AHB03, SBN+97]. The use of stale values may be a possible optimization in certain programs; work still has to be done to investigate this further [ABH04, BL02].

If run-time verification confirms the absence of certain fault patterns, one can conclude that certain faults (such as data races) are also absent, provided that lock usage and data access patterns have been fully covered by the test suite executed. Most commercial programs prevalently use static locks [AB01], which makes it trivial to confirm the first part of that assumption. In traditional software testing, *coverage* metrics such as statement coverage [Pel01] have been used to determine the effectiveness of a test suite. Statement coverage returns information on how many times each statement has been executed by a test suite. When using static locks, full statement coverage will thus execute each lock and data access at least once and provide full "lock coverage", use of all possible locks within the test suite, as well.

```
static Object lock1, lock2;
/* two distinct locks */

void possibleDataRace(Data x, Data y) {
/* can be called concurrently */
    synchronized (lock1) {
        x.value++;
        /* lock1 protects x */
    }
    synchronized (lock2) {
        y.value++;
        /* lock2 protects y */
    }
}
```

Figure 5.2: *Aliasing may easily hide faults during testing.*

A test suite providing full statement coverage is very effective at finding failures but not sufficient, as Figure 5.2 demonstrates. Assume class `Data` to denote shared data of which field `value` can be used by multiple threads. In the example, there are two locks protecting data access: `lock1` and `lock2`. Some convention could always clearly define the lock to be used for data access. For instance, in an array, `lock1` could protect all array elements with an odd index while the other ones could be protected by `lock2`. Therefore the example program could be data race free. However, assume that there cases where the two arguments to method `possibleDataRaces` are aliased, i.e., `x` = `y`. Since the two locks are distinct, there is no common lock protecting data access, and a low-level data race occurs. It could of course be easily possible that a test suite only includes safe uses of this method, and never cause the possible data race to occur. Nevertheless, such constructs are extremely rare in practice, and it has been demonstrated that data race analysis finds faults very effectively [BDG$^+$04].

In the given example, if one requires that lock protection for data is known statically (for instance, by a statically known type), then one can prevent the aliasing of the two method arguments. In such a scenario, simple statement coverage will suffice to prevent data races. Thus there are certainly cases where run-time verification yields no false negatives and is sound. In the area of concurrency errors, the domain covered by this thesis, many properties only depend on locks used and fields accessed. Data access patterns within a lock are usually independent of program input and can usually be tested effectively; yet there is no notion to make the assurance gained by such testing a guarantee. This observation should encourage research for future programming languages that enforce such notions of tying locks to data; also see Section 11.5.

The second multi-threading property investigated in this thesis [ABH04] considered atomicity of actions. Because this property only depends on intra-method data flow, it is much easier to test. For such properties, edge coverage seems to be sufficient, combined with a loop coverage criterion demanding at least zero and one loop executions. Given that coverage, data flow within the current stack and register frame is fully covered. However, block-local atomicity still requires all individual data accesses to be race-free, which cannot guaranteed as easily, as described above.

Still, these observations suggest that there exists a reasonably large class of programs where the absence of certain faults *can* be proved by sufficient testing. It remains to be seen how such coverage metrics and tools will be developed.

5.7 Summary

Run-time verification distinguishes itself from testing by the fact that a much larger part of a program behavior can be monitored, in a systematic way. Monitored properties can be categorized into application-specific properties, and generic ones that should apply to any program. Application-specific properties are often specified as contracts or as temporal logic formulae. Generic properties are commonly built into the verification tool and include the absence of memory corruption (invalid pointers) and data races. Current tools are usually specialized for one category.

Several techniques exist to generate and monitor a trace. Trace generation can be done by instrumentation, which injects event generation code into the application. Alternative approaches are based on either wrapping system libraries or providing an entirely new execution environment. The latter is the most flexible one but sacrifices optimizations available in standard execution environments. Trace monitoring is independent of the trace generation technique used. It can occur within the application, on the fly, or as a separate process, off-line. Initial experiments indicate that the communication overhead of off-line monitoring outweighs its benefits.

6 Combined Static and Dynamic Analysis

Static analysis is usually faster than dynamic analysis but less precise. Therefore it is often desirable to retain information from static analysis for run-time verification, or to compare the results of both techniques. However, this requires writing two programs, which may not act identically under the same conditions. It would be desirable to share the same generic algorithm by static and dynamic analysis. In JNuke, a framework for static and dynamic analysis of Java programs, this has been achieved. By keeping the architecture of static analysis similar to a virtual machine, the only key difference between abstract interpretation and execution remains the nature of program states. In dynamic analysis, concrete states are available, while in static analysis, sets of (abstract) states are considered. Our new analysis is generic because it can re-use the same algorithm in static analysis and dynamic analysis. This chapter describes the architecture of such a generic analysis. To our knowledge, JNuke is the first tool that has achieved this integration, which enables static and dynamic analysis to interact in novel ways.

Section 6.1 gives some background and motivation. Section 6.2 introduces graph-free static analysis as used in JNuke. Section 6.3 describes run-time verification. Generic analysis algorithms, applicable to both a static and dynamic context, are described in Section 6.4. Section 6.5 summarizes this chapter.

6.1 Background and Motivation

Originally JNuke was designed for dynamic analysis, encompassing explicit-state software model checking [ASB$^+$04, Eug03, VHB$^+$03] and run-time verification [ASB$^+$04, RV04]. For generic run-time verification, the engine executes only one schedule defined by a given scheduling algorithm. An observer interface provides access to events occurring during program execution. Event listeners can then query the virtual machine for detailed data and thus implement any run-time verification algorithm.

Static analysis was added to JNuke at a later stage. In the initial version, static analysis in JNuke could not handle recursion and required algorithms to be targetted to a static environment [ABH04]. This chapter describes the solution for recursion and furthermore allows sharing of algorithms in a static and dynamic environment.

JNuke's generic analysis framework allows the entire analysis logics to be written such that they are agnostic of whether the "environment" is a static or dynamic analysis. Both versions require only a simple wrapper that converts environment-specific data into a form that a generic algorithm can use.

Even a fast execution environment is greatly slowed down by run-time verification and thus needs support from a static data flow analysis in order to reduce the amount of data to be monitored at run-time. Ideally this functionality is pluggable in the class loader. Because the entire framework is integrated, conversion of static information for dynamic analysis is not necessary.

It is not always certain whether it is beneficial to implement a static or a dynamic analysis for a specific property. Static analysis can scale easily to a million lines of code per minute or more [AB01, ABH04] if it does not require complex pointer aliasing information. The block-local atomicity analysis algorithm [ABH04] seemed to be most suitable for static analysis because the property checked is context-insensitive (method-local). However, accuracy depends heavily on the quality of the pointer analysis used [WL04]. Imprecise lock information may generate spurious warnings. Therefore it is interesting to see whether a dynamic version of the same algorithm produces better results.

Furthermore, the fact that the algorithm itself is identical for static and dynamic analysis allows a novel kind of combined analysis for fault detection, as outlined in Figure 6.1.

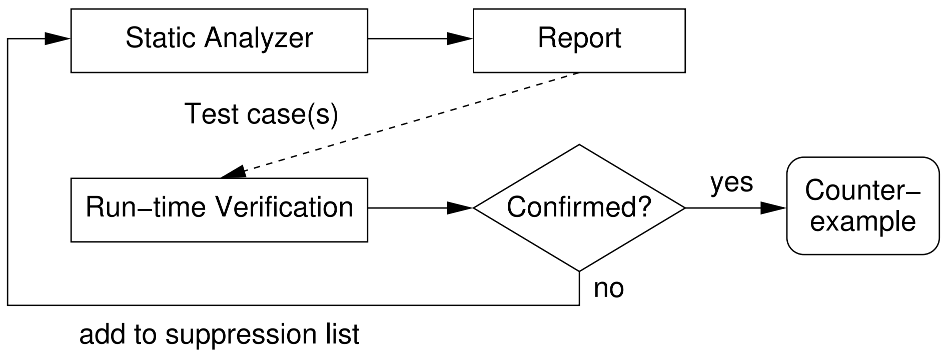


***Figure 6.1**: A new tool flow for fault detection using combined static and dynamic analysis.*

A static analyzer looks for faults. Reports are then analyzed by a human, who writes test cases for each kind of fault reported. Run-time verification will then analyze the program using the dynamic version of the same algorithm, possibly confirming the fault as a failure or counterexample. Confirmed faults are repaired; unconfirmed faults can be investigated further using more tests.

If a failure is not confirmed, even after multiple iterations of creating test cases, given reports can be suppressed in future runs of the static analyzer. Of course this particular methodology gives up soundness. However, it still facilitates fault finding, as current approaches only offer a manual review of reports. The generic algorithm is shared by both tools, which is our contribution and enables this tight integration of static and dynamic analysis.

6.2 Static Analysis in JNuke

Static analysis approximates the set of possible program states. Such an approximated state encompasses a set of concrete program states and will be denoted an *abstract state* in this chapter. Static analysis iterates over these abstract states until a fixpoint is reached or a certain limit, such as a second loop execution, is reached.

Static analysis scales well for many properties, as certain properties may be modular and only require summary information of dependent methods or modules. "Classical" static analysis constructs a graph representation of the program and calculates the fix point of properties using that graph [CC77].

This is very different from dynamic analysis, which evaluates properties against an event trace originating from a concrete program execution. Using a graph-free analysis [Moh02], static analysis is again close to dynamic execution. In this chapter, a graph-free static analysis is extended to a *generic analysis* which is applicable to dynamic analysis as well.

In JNuke, static analysis works in a way which is very similar to dynamic execution. An *environment* treats control flow semantics and implements non-deterministic control flow. The analysis algorithm models the semantics of the set of abstract states at each location, from which the values of program properties can be deduced. In such a graph-free data flow analysis [Moh02] data locality is improved because an entire path of computation is followed as long as valid new successor states are discovered. Each Java method can be executed in this way. The abstract behavior of the program is modelled by the user. The environment runs the analysis algorithm until an abortion criterion is met or the full abstract state space is exhausted.

6.2.1 Graph-free abstract interpretation

Abstract interpretation of a program involves computation of the least or greatest fix point of a system of semantics of the form:

$$\begin{cases} x_1 & = & \Phi_1(x_1,\dots,x_n) \\ & \vdots & \\ x_n & = & \Phi_n(x_1,\dots,x_n) \end{cases}$$

where each index $i \in C = [1,n]$ represents a location of the program, and each function Φ_i is a continuous function from L^n to L. L is the abstract lattice of program properties. Each function Φ_i computes the property holding at i after one program step executed. Applying the equations iteratively therefore computes the solution eventually, but is inefficient [Bou93]. A well-established speed-up technique consists of widening [CC77], where some equations for x_i are replaced with $x_i = x_i \nabla \Phi_i(x_1,\dots,x_n)$, operator ∇ being a safe approximation of the least upper bound such that the iteration strategy eventually terminates. For optimal performance, widening operators should be tuned for a specific iteration strategy [Bou93].

In graph-free abstract interpretation [Moh02], the properties x_i at each program location i are represented by an abstract state, representing a set of concrete states S. Computation of these properties from the abstract state is quite straightforward and can be done uniformly for all locations, by using a property predicate P:

$$\begin{cases} x_1 & = & P(S_1) \\ & \vdots & \\ x_n & = & P(S_n) \end{cases}$$

The central problem is computation of the fix point of all abstract states S_i:

$$\begin{cases} S_1 & = & \text{absexec}(1, S_1) \\ & \vdots & \\ S_n & = & \text{absexec}(n, S_n) \end{cases}$$

Since this technique already uses an execution environment, control flow structures are evaluated correctly and efficiently [Moh02]. Whenever a new abstract state S'_i is computed, the effects of the instruction at i are calculated based on the semantics of the abstract domain: $S'_i = \text{absexec}(i, S_i)$. The difference to dynamic execution therefore lies in the nature of an abstract state S_i which can represent several concrete states $\{s_j, \ldots, s_k\}$, some of them possibly unreachable in real execution. Function absexec may also over-approximate its result S'_i, which can be compared to widening as described in Section 6.4.

6.2.2 Separation of control flow and bytecode semantics

The fix point of all possible program behaviors is calculated by iterating over the set of all reachable abstract program states. In JNuke, iteration over the program state space is separated from analysis logics. A generic *control flow module* controls symbolic execution of instructions, while the analysis algorithm deals with the representation of (abstract) data and the semantics of the analysis. The control flow module implements a variant of priority queue iteration [HDT87], executing a full path of computation as long as successor states have not been visited before, without storing the flow graph [Moh02]. Abstract states S as used in this algorithm refer to a set of program states at a single location l. A single abstract state at l thus usually represents a set of concrete states at that location.

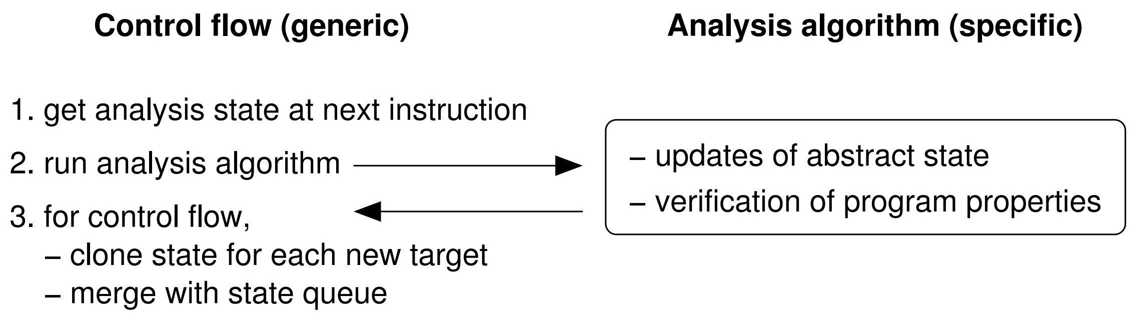

Control flow (generic)
Analysis algorithm (specific)
1. get analysis state at next instruction
2. run analysis algorithm
3. for control flow,
– clone state for each new target
– merge with state queue
– updates of abstract state
– verification of program properties

***Figure 6.2**: Separation of control flow and analysis algorithm.*

Figure 6.2 shows the principle of state space exploration: The generic control flow module first chooses an instruction to be executed from a set of unvisited abstract states. It then runs the specific analysis algorithm on that unvisited abstract state. That algorithm updates its abstract state and verifies the properties of interest. After evaluation of the current instruction, the control flow module adds all valid successor states to the queue of states to visit, avoiding duplicates by keeping a set of seen states. When encountering a branch instruction such as `switch`, all possible successors are added to the state space. Furthermore, each possible exception target is also added to the states that have to be explored.

It is up to the specific analysis algorithm to model data values. At the time of writing, the block-local atomicity analysis for stale values [ABH04] is implemented. This analysis tracks the state of each register (whether it is shared and therefore possibly stale) and includes a simple approximation of lock identities (pointer aliasing [WR99]). It does not require any further information about the state of variables, and thus chooses to execute every branch target. Due to the limited number of possible states for each register, the analysis converges very quickly.

6.2.3 Optimized state space management

After the specific algorithm has calculated the outcome of the current abstract state, the control flow algorithm evaluates all possible successor instructions. To achieve this, the current abstract state is cloned for each new possible successor state. The control flow module then adds this state to the queue of states to visit. This corresponds to a basic model-checking algorithm [Hol91] but is not the most efficient way to perform static analysis for software. The observation was that a lot of states were stored and then immediately re-fetched in the next cycle of the main loop. This is because many instructions in Java do not affect control flow. Therefore the above algorithm from Figure 6.2 was modified to only store a state if (a) it generates multiple successor states or (b) another state with the same program counter had been visited before. This has the effect that the major part of a method is executed “linearly” without storing the current state. Only if a branch instruction occurs, the state is cloned and stored.

The reason why this optimization works well is that many Java bytecode instructions do not affect control flow. Therefore our algorithm does not store the current state if a unique immediate successor instruction is eligible. A state is only stored if it is target of a branch instruction. This reduces memory usage [Moh02] but may visit a state twice: If an instruction i_b is the target of a backward jump, such as in a `while` loop, it is only recognized as such when the branch instruction is visited, which usually occurs after i_b has been visited before. However, this overhead is small since it only occurs during the first iteration. As an example, assume some execution visits states 1 – 5 and then branches back to state 3. No state is stored until state 5 is reached. The current abstract state at 5 is stored since its code consists of a branch instruction. States 3 and 4 are then re-visited because the algorithm has not stored them during its first iteration. During the second iteration, state 3 is stored because it is now known to be the target of a backward jump. Therefore, if the abstract program state at 3 does not change during future loop iterations, that state is not re-visited anymore.

6.3 Run-time verification in JNuke

JNuke implements a virtual machine that can execute the full set of Java bytecode instructions [LY99] and therefore any Java program given an implementation of the native code used by it. An application programming interface (API) allows event listeners to connect to any action of interest and query the VM about its internal state, thus implementing any analysis algorithm of choice.

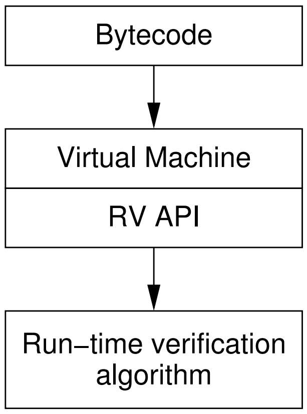


Figure 6.3*: Run-time verification in JNuke.*

Prior to execution, the class loader transforms the Java bytecode into a more abstract, RISC-like version containing only 27 instructions, which is then transformed into a register-based version [AB05b]. Execution of the program generates a series of events, denoted by an event *trace.* During execution, the run-time verification API (RV API) allows event listeners to capture this event trace. Such listeners are used to implement scheduling policies and run-time verification algorithms. The algorithm is responsible to copy data it needs for later investigation, as the VM is not directly affected by the listeners and thus may choose to free data not used anymore. Figure 6.3 shows an overview of the JNuke VM and how a run-time verification algorithm can be executed by callback functions in the VM. For simplicity, the figure omits the fact that some communication from the RV algorithm back to the VM actually occurs in the presence of garbage collection. In such a situation, the RV algorithm must instruct the VM to suppress collection of data it is still using, in order to prevent access to memory locations that are already freed or reallocated for other data [Far04]. Such a protection applies to all algorithms that use references to identify data. This chapter focuses on generic analysis, while Chapter 8 describes the JNuke VM and its RV API in more detail.

6.4 Generic Analysis Algorithms

The goal of this extension to JNuke was to be able to use *generic analysis algorithms.* These algorithms should work equally in both a *static environment* (using abstract interpretation) and a *dynamic environment* (using run-time verification). The problem is that the environments are quite different: the VM offers a single fully detailed state. Abstract interpretation [CC77], on the other hand, deals with sets of states, each state containing imprecise information that represents several concrete states. The challenge was to reconcile the differences between these two worlds and factor out the common parts of the algorithm.

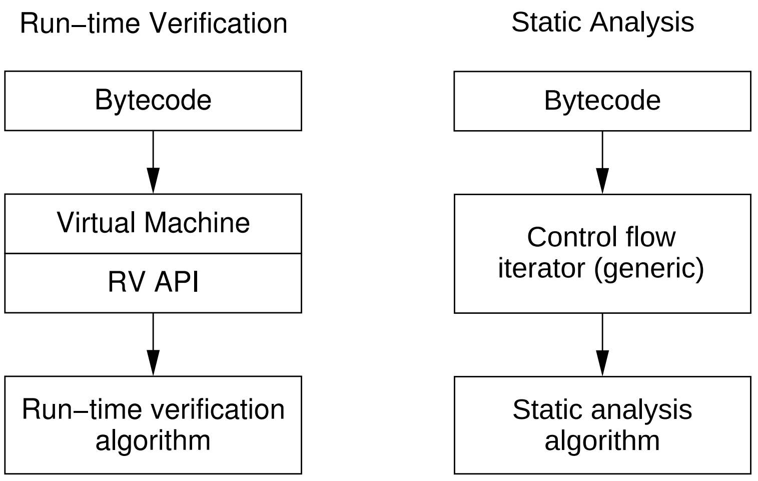

Run–time Verification
Static Analysis
Bytecode
Bytecode
Virtual Machine
RV API
Control flow iterator (generic)
Run–time verification algorithm
Static analysis algorithm

Figure 6.4: *Classical approaches duplicate the analysis algorithm for the dynamic and static case.*

Figure 6.4 illustrates the problem: The analysis algorithm is duplicated for both analysis scenarios. Much genericity and flexibility is already gained by utilizing a generic observer-based run-time verification interface [ABG$^+$03] and a generic iteration module which analyzes control flow [AB05a]. However, the final property-specific part still has to be written twice, adapted to each scenario. This is even though the analysis clearly represents the same rules. The goal is therefore to have a generic analysis. The design that allows to achieve this is the key contribution of this chapter.

A generic analysis represents a single program state or a set of program states at a *single* program location. It also embodies a number of event handlers that model the semantics of bytecode operations. Both static analysis and run-time analysis trigger an intermediate layer that evaluates the events. The environment hides its actual nature (static or dynamic) from the generic algorithm and maintains a representation of the program state that is suitably detailed.

Figure 6.5 shows the principle. Run-time verification is driven by a *trace,* a series of events e emitted by the run-time verification API. An event represents method entry or exit, or execution of an instruction at location l. Conventional run-time analysis analyzes these events directly. The dynamic environment, on the other hand, uses the event information to maintain a *context* c of algorithm-specific data before relaying the event to the generic analysis. This context is used to maintain state information s that cannot be updated uniformly for the static and dynamic case. It is updated similarly by the static environment, which also receives events e, determining that successor states of abstracts states at l are to be computed.

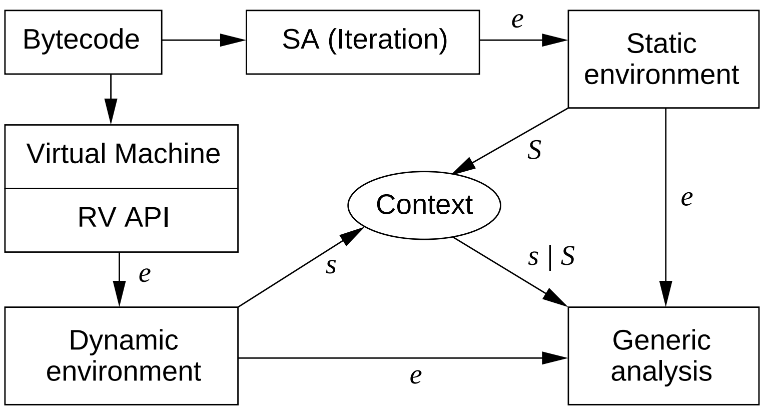


***Figure 6.5**: Running generic analysis algorithms in a static or dynamic environment.*

The key difference for the static environment is that its updates to c concern *sets of states S*. Sets of states are also stored in components used by the generic algorithm. Operation on states (such as comparisons) are performed through delegation to component members. Therefore the "true nature" of state components, whether they embody single concrete states or sets of abstract states, is transparent to the generic analysis. It can thus be used statically or dynamically.

Existing work in software model checking by the Java PathFinder (JPF) project also concerned the relation between abstract and concrete program states: Their way of obtaining a feasible counter-example trace is by using only deterministic choices during state space exploration [PDV03]. This technique corresponds to reducing a set of (abstract) states to a concrete state.

However, property verification algorithms as in run-time verification have so far not been applied to the resulting concrete states. This is because in the JPF/Bandera tool chain, the counter-example trace is already known to be concrete at that stage. Instead, the counter-example trace is used for abstraction refinement [PDV03].

In our generic analysis, the abstract domain is chosen based on the features required to evaluate given properties. Both the domain and the properties are implemented as an observer algorithm in JNuke. Future algorithms may include an interpreter for logics such as LTL [Pnu77]. Interpretation of events with respect to temporal properties would then be encoded in the generic analysis while event generation would be implemented by the static and dynamic environment, respectively.

6.4.1 Context data

Context data c has to be applicable to static and dynamic analysis. The dynamic environment maintains a single (current) context c while the static one maintains one context per location, c_l. In a static environment, certain data may not be defined precisely; for instance, in a field access, the static environment often cannot provide a pointer to the instance of which the field was accessed. There are two ways to deal with this problem: The generic analysis must not require such data, or the static layer must insert artificial values.

The latter was used for modeling static *lock sets,* where the static layer uses symbolic IDs to distinguish locks, rather than their pointers. On each lock acquisition, the lock set in c_l is updated with a new such lock ID. The generic analysis may only read locks or perform non-destructive, abstract tests, such as testing set intersections for emptiness.

Due to polymorphism (in the implementation) of the actual set content, the generic analysis therefore never becomes aware of the true nature of the locks. The environment also maintains contextual information for each lock, such as the line number where it was acquired. Again, polymorphism allows lookup from locks to line numbers without revealing the content of the lock.

In general, the environment must create suitable representations of state information used by the generic analysis. The generic analysis only operates on such data. The environment thus acts as a proxy [GHJV95] for the virtual machine, if present, or replaces that data with appropriate facsimiles in static analysis. These facsimiles have to be conceptually isomorphic with respect to concrete values obtained during run-time analysis. Distinct objects have to map to distinct representations. Of course, true isomorphism is only achieved if pointer analysis is absolutely accurate.

The proxy objects implemented so far incur little overhead but are rather specialized and may only be re-used if another algorithm has equivalent requirements. For example, if the alias information of two locks is not known, one can either assume they are equal or different. The conservative approximation of a lock set depends on the algorithm used. A static version of a low-level data race algorithm [SBN$^+$97] can safely assume that all locks are different, but this will likely lead to many false positives since the intersection of different lock sets is always going to be empty in such cases. The same assumption holds for the high-level data race [AHB03] and block-local atomicity [ABH04] algorithms. Other algorithms may have the reverse requirement, i.e., two locks of which the alias information is unknown have to be treated as equal locks.

Context data	Type	Content for static analysis	Content for dynamic analysis
Current monitor block (ID)	Integer	Integer reflecting approximated non-reentrant lock acquisitions	True count of the total number of lock acquisitions so far
Registers ("stack frame")	Array	Abstract entries containing only information about sets of possible register properties	Shadow values reflecting property of interest (exact status of each register)
Lock set	Set	Integer descriptors for each lock	True lock set
Lock context	Map	Map of integers to locations	Map of locks to locations

Table 6.1: *Context differences for the static and dynamic block-local atomicity analysis.*

The generic block-local atomicity algorithm [ABH04] has the property that it is agnostic to certain concrete values (such as the values of integers) but needs relatively precise information about others (locks). It thus provides a good example of a generic analysis algorithm, as other ones are expected to show similar differences. Table 6.1 gives an overview of the differences between the static and dynamic versions of the algorithm.

```
void atGetField(Bytecode bc) {
    /* Compute effect of GetField instruction w.r.t. stale values. */
    /* Potential data races with the reference to the object instance
     * are discovered by the Eraser lock set algorithm, which monitors
     * individual field accesses. */

    /* Check block-local atomicity property for arguments consumed by
     * this instruction */
    checkRegisters(bc);
    StackElement result = newData(); /* possibly shared, see below */
    localvars.set(bc.getResultRegister(), result); /* store result */
}

StackElement newData() {
    /* Generic case where new data is obtained from a possibly shared
     * field. If data is shared, set correct monitorBlock etc. */

    StackElement data = new StackElement(); /* unshared by default */
    if (context.getLockSet().count() > 0) {
        data.setShared(true);
        data.setMonitorBlock(context.getCurrentMonitorBlock()));
    }
    return data;
}

void checkRegisters(Bytecode bc) {
    /* Check each local variable for local atomicity violation. */

    for (int i = 0; i < bc.getNumRegs(); i++) {
      int idx = bc.getRegisterIndex(i);
      if (registerIsLocalVariable(idx)) {
          StackElement data = localvars.get(idx);
          if (data.getShared() &&
              (data.getMonitorBlock() != getCurrentMonitorBlock()))
            /* report error */
        }
    }
}
```

Figure 6.6: *Excerpt of the block-local atomicity algorithm (simplified).*

In the block-local atomicity algorithm, the static environment approximates the lock set, representing it with proxy objects; the dynamic environment simply queries the VM. The property check itself is completely independent of the environment, as it refers to "shadow data" which reflects the status of each register, i.e., whether their value is stale or not. In the static case, the semantics of sets of states are reflected by approximating the set of all possible values in the operations on registers. Figure 6.6 shows an excerpt of this generic algorithm. Its code has been simplified for clarity, using Java-like syntax and ignoring registers with a size of 64 bits. It contains the essence of the idea outlined above: Class `context` stores the lock set, which is updated by the environment and queried by `context.getLockSet()`. Context data is therefore updated with each evaluation step, and queried on demand. A static environment approximates the lock set using proxy objects. Note that the approximation can be made conservative if pointer alias information is imprecise [ABH04]. The dynamic environment simply queries the VM to obtain the real, concrete lock set. The property check itself is completely independent of the environment, as `localvars` refers to "shadow data" which reflects the status of reach register, i.e., whether their value is stale or not [ABH04]. In the static case, the semantics of sets of states are reflected by approximating the set of all possible values in the operations on `localvars` (such as `get` and `set` shown here). Therefore the generic algorithm performs the same operations on concrete states as on sets of abstract states.

6.4.2 Interfacing run-time verification

Many run-time verification algorithms, such as Eraser [SBN+97], are context-sensitive and not thread-local. Such an algorithm receives events from *all* threads and methods. A run-time variant of such an algorithm therefore requires only a single instance of object holding analysis data. In such a case, creating a static variant is less interesting since the dynamic algorithm, if used with a good test suite, yields excellent results [BDG+04].

Conversely, a context-insensitive (method-local), thread-local property is more amenable to static analysis, but actually makes run-time analysis more difficult. This is counter-intuitive because such properties are conceptually simpler. The block-local atomicity algorithm serves as an example here, being both thread-local and method-local. For run-time verification, a new instance of this analysis has to be created on each method call and thread. Instances of analysis algorithms then correspond to stack frames on the program stack.

Figure 6.7 shows a UML diagram [RJB98] depicting how the dynamic environment creates instances of an analysis algorithm as needed. The first layer, class *thread splitter,* splits events according to their thread ID, creating a separate instance of class *dynamic analysis* as needed, one for each thread. The second layer, driven by class dynamic analysis, creates a new instance of class *analysis algorithm* for each stack frame, at the beginning of each method call.

Each new analysis instance is completely independent of any others, except for a shared, global context (such as lock sets, which are kept throughout method calls) and return values of method calls. The dynamic environment maintains the shared context and relays return values of method calls to the analysis instance corresponding to the

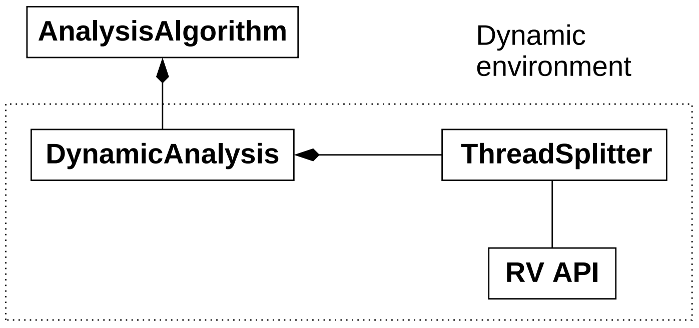


Figure 6.7: *Interfacing run-time verification with a generic analysis algorithm.*

caller. In this case, lock set information is already available by the RV API and does not have to be managed separately. Other global data can be managed by an extra listener that evaluates events before relaying them to class thread splitter. The thread-specific instances of dynamic analysis deal with communicating return values from an "inner" instance, corresponding to the callee, to the "outer" one, referring to the caller.

6.4.3 Interfacing static analysis

Static analysis calculates the set of all possible program states. Branches (test nodes) are treated non-derministically by considering all possible successors and copying *(cloning)* the current state for each outcome. Junction nodes denote points where control flow of several predecessor nodes merges [CC77]. In this chapter, the operation that creates a new set of possible states at a junction node will be called *merging*.

The key is that the generic algorithm is not aware that static analysis requires copying and merging operations. To achieve this, the capabilities of the generic analysis must be extended with the *Mergeable* interface. The extended class inherits the algorithm and delegates cloning and merging states to the components of a state, as shown in Figure 6.8.

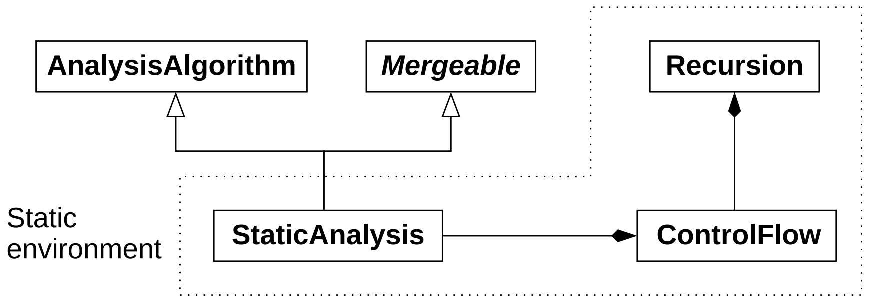


Figure 6.8: *Interfacing static analysis with a generic analysis algorithm.*

By merging states, sets of states are generated. Computations of state components must therefore support set semantics for static analysis. What is important is that the *analysis logics* are unchanged: the generic algorithm is still unaware that cloning, merging, and set operations happen "behind the scenes" and implements its property checking as if only a single state existed. In some cases, static analysis may converge slowly; convergence is improved by using a widening operator [CC77] which can be implemented by the merge operation.[1]

In dynamic analysis, only one program location l is active (per thread), corresponding to a single program state s. This current state s is updated and the result assigned to successor state s'; the original state s is then discarded. In static analysis, abstract states S_i at all program locations i are being analyzed. The abstract states are analyzed in an iterative way, and thus the abstract states at each program location are retained until iteration terminates. Each abstract state S_i is represented by an instance of the generic algorithm. The type of operation performed to model the semantics of each instruction remains the same for static and dynamic analysis.

In our framework, the successor states of one set S_i are calculated in each iteration step. The choice of i is implemented by a control flow module, as described in Section 6.2. This covers intra-method analysis, leaving open the problem of method calls. It is desirable that the entire statically reachable call graph is traversed so each reachable method in a program is analyzed. A *recursion* class solves this challenge. It *expands* a method call by starting a new instance of the control flow class. Figure 6.8 shows an overview of these connections. The recursion class starts with the `main` method and creates a new instance of the control flow class for each called method. The control flow class performs intra-method analysis and delegates method calls back to the recursion class, which also handles multi-threading by exploring the behavior of threads when encountering a thread start point, e.g. a `run` method. This way, the algorithm explores the behavior of all threads.

This leaves open the problem of self-recursion or mutual recursion. It is not possible to model the effects of a recursive method that calls another method higher up in its stack frame using this algorithm. This is because the precise effect of that method call depends on itself.[2] Therefore the static analysis class has to implement a *summary* method, which models method calls without requiring knowledge about the effects of a method. Such a summary method can conservatively assume the worst possible outcome or provide more precise information.

The result of each evaluated method call is stored as a summary. Context-sensitivity is modeled by evaluating each method call once for each possible call context. For a context-insensitive analysis, an empty call context is assumed. Context sensitivity therefore does not directly have an effect on the fact that each method call requires a new instance of control flow and analysis objects. However, once summaries are available, their information will act as a cache. For context-insensitive analysis, the empty call context always matches for a given method, and thus each method call is only evaluated once.

[1] In our block-local atomicity analysis, the only case where widening is applied is the merging of two shared data elements from different monitor blocks. The widening takes one of the two block IDs and discards the other one. This has no effect on the analysis result but discards one possible error trace, thus always reducing the set of error traces to a singleton set.

[2] A bounded expansion of recursion is possible, approximating the unbounded behavior.

In principle, every analysis algorithm can be split up into a generic algorithm and its environment. Most data flow problems can be seen as set-theoretic or closure problems [MR90] and their nature will affect how the merge operation is implemented. Precision of the analysis will depend on the approximation of pointer aliasing [WR99]. If accurate information about data values is needed or when environment-specific optimizations are called for, the generic part of an algorithm may become rather small compared to the size of its (static or dynamic) environment. However, with the block-local atomicity algorithm, it has been our experience that the generic algorithm does indeed embody the entire logics and thus is not just a negligeable part of the whole. Notably, adapting a static algorithm for dynamic analysis is greatly facilitated with our approach.

6.5 Summary

Static and dynamic analysis algorithms can be abstracted to a generic version, which can be run in a static or dynamic environment. By using a graph-free analysis, static analysis remains close enough to program execution such that the algorithmic part can be re-used for dynamic analysis. The environment encapsulates the differences between these two scenarios, making them completely transparent in the evaluation of the generic algorithm. This way, the entire analysis logics and data structures can be re-used, which allows for a comparison of the two technologies with respect to precision and efficiency. Experiments with JNuke, summarized in Section 9.6, have shown that the static variant of a stale-value detection algorithm is significantly faster but less precise than the dynamic version. This underlines the benefit of using static information in order to reduce the overhead of run-time analysis. The fact that both types of analysis share the algorithm also allows for combining them in a tool that applies run-time verification to test cases resulting from static analysis reports.

7 Bytecode Inlining and Abstraction

In Java bytecode, intra-method subroutines are employed to represent code in “finally” blocks. The use of such polymorphic subroutines within a method makes bytecode analysis very difficult. Fortunately, such subroutines can be eliminated through recompilation or inlining. Inlining is the obvious choice since it does not require changing compilers or access to the source code. It also allows transformation of legacy bytecode. However, the combination of nested, non-contiguous subroutines with overlapping exception handlers poses a difficult challenge. This chapter presents an algorithm that successfully solves all these problems without producing superfluous instructions. Previous algorithms were either not published and verified, or had shortcomings such as generating excess instructions. Furthermore, inlining can be combined with bytecode simplification, using abstract bytecode. We show how this abstration is extended to the full set of instructions and how it simplifies static and dynamic analysis.

Section 7.1 shows problems with existing Java bytecode using subroutines. Section 7.2 gives an overview of Java compilation and treatment of exception handlers. The inlining algorithm is given in Section 7.3. Section 7.4 describes conversion to abstract, register-based bytecode. Section 7.5 describes differences between our work and related projects, and Section 7.6 summarizes this chapter.

7.1 Problems with Bytecode

In general, a program written in the Java programing language is compiled to Java *bytecode,* a machine-readable format which can be loaded and executed by a Java Virtual Machine (VM) [LY99]. Prior to execution, such bytecode must pass a well-formedness test called *bytecode verification,* which should allow a regular Java program to pass but also has to ensure that malicious bytecode, which could circumvent security measures, cannot be executed. The Java programming language includes methods, which are represented as such in bytecode. However, bytecode also contains *subroutines,* functions *inside* the scope of a method. A special jump-to-subroutine (`jsr`) instruction saves the return address on the stack. A return-from-subroutine (`ret`) instruction returns from a subroutine, taking a register containing the return address as an argument. This artefact was originally designed to save space for bytecode, but it has three unfortunate effects:

1. It introduces functionality not directly present in the source language.

2. The asymmetry of storing the return address on the stack with `jsr` and retrieving it from a register (rather than the stack) greatly complicates bytecode analysis.

3. A subroutine may read and write local variables that are visible within the entire method, requiring distinction of different calling contexts.

The second and third effect have been observed by Stärk et al. [SSB01], giving numerous examples that could not be handled by Sun's bytecode verifier for several years. The addition of subroutines makes bytecode verification much more complex, as the verifier has to ensure that no `ret` instruction returns to an incorrect address, which would compromise Java security [LY99, SSB01]. Therefore subroutine elimination is a step towards simplication of bytecode, which can be used in future JVMs, allowing them to dispense with the challenge of verifying subroutines.

Correct elimination of subroutines can be very difficult, particularly with nested subroutines, as shall be shown in this chapter. Furthermore, considering the entire bytecode instruction set makes for very cumbersome analyzers, because it encompasses over 200 instructions, many of which are variants of a base instruction with its main parameter hard-coded for space optimization [LY99]. Therefore we introduce a register-based version of abstract bytecode which is derived from [SSB01]. By introducing registers, we eliminate the problem of not having explicit instruction arguments, simplifying analysis further.

Bytecode was the chosen input format because it allows for verification of Java programs without requiring their source code. Recently, compilers for other programming languages have been developed which use Java bytecode as their output format. This includes `jgnat` for Ada [Bri99] and `kawa` for Scheme [Bot98]. However, bytecode subroutines and its a very large, stack-based instruction set make static and dynamic analysis difficult. JNuke eliminates subroutines and simplifies the bytecode instruction set.

7.2 Java Compilation with Bytecode Subroutines

7.2.1 Java Bytecode

Java bytecode [LY99] is an assembler-like language, consisting of instructions that can transfer control to another instruction, access local variables and manipulate a (fixed-height) stack. Each instruction has a unique address or *code index*. Table 7.1 describes the instructions referred to in this chapter. In this table, r refers to a register or local variable, j to a (possibly negative) integer value, and a to an address. The instruction at that address a will be denoted as $\text{code}(a)$, while the reverse of that function, $\text{index}(ins)$ returns the address of an instruction.

The maximal height of the stack is determined at compile time. The type of instruction argument has to be correct. Register indices must lie within statically determined bounds. These conditions are ensured by any well-behaved Java compiler and have to be verified

Instruction	Description
`aload` *r*	Pushes a reference or an address from register *r* onto the stack.
`iload` *r*	Pushes an integer from register *r* onto the stack.
`astore` *r*	Removes the top stack element, a reference or address, storing it in *r*.
`istore` *r*	Removes the top stack element, an integer, storing it in register *r*.
`goto` *a*	Transfers control to the instruction at *a*.
`iinc` *r j*	Increments register *r* by *j*.
`ifne` *a*	Removes integer *j* from the stack; if *j* is not 0, transfers control to *a*.
`jsr` *a*	Pushes the successor of the current address onto the stack and transfers control to *a*.
`ret` *r*	Loads an address *a* from register *r* and transfers control to *a*.
`athrow`	Removes the top stack element, a reference, "throwing" it as an exception to the caller.
`return`	Returns from the current method, discarding stack and local variables.

***Table 7.1**: A subset of Java bytecode instructions.*

by the class loader of the Java Virtual Machine (VM) during *bytecode verification* [LY99], the full scope of which is not discussed here.

7.2.2 Exception Handlers and Finally Blocks

The Java language contains *exceptions,* constructs typically used to signal error conditions. An exception supercedes normal control flow, creates a new exception object *e* on the stack and transfers control to an *exception handler.* The range within which an exception can be "caught" is specified by a `try` block. If such an exception *e* occurs at run-time, execution will continue at the corresponding `catch` block, if present, which deals with the exceptional program behavior. An optional `finally` block is executed whether an exception occurs or not, but always after execution of the `try` and `catch` blocks. Therefore, the presence of a `finally` block creates a dualistic scenario: in one case, an exception occurs, which requires both the `catch` and `finally` blocks to be executed. In the absence of an exception, or if an exception occurs that is not within the type specification of the `catch` block, only the `finally` block has to be executed. Because of this, a default exception handler is required to catch all exceptions that are not caught manually.

In the following text, lower case letters denote single values. Monospaced capital letters such as `C` will denote control transfer targets (statically known). Capitals in italics such as *I* denote sets or ranges of values. In Java bytecode, an exception handler $h(t, I, \mathtt{C})$ is defined by its type t, range I, which is an interval $[i_\alpha, i_\omega]$,[1] and handler code at `C`. Whenever an exception of type t or its subtypes occurs within I, control is transferred to `C`. If several handlers are eligible for range I, the first matching handler is chosen. If, for

[1]In actual Java class files, handler ranges are defined as $[i_\alpha, i_\omega[$ and do *not* include the last index of the interval, i_ω. This is only an implementation issue. For simplicity, this chapter assumes that handler ranges are converted to reflect the above definition.

an instruction index a, there exists a handler h where a lies within its range I, we say that *h protects a*: $\text{protects}(h,a) \leftrightarrow a \in I(h)$.

As specified by the Java language [GJSB00], a `finally` block at `F` always has to be executed, whether an exception occurs or not. This is achieved by using an unspecified type t_{any} for a default handler $h_d(t_{any}, I_d, \texttt{F})$. If a `catch` block is present in a `try`/`catch`/`finally` construct, the exception handler $h'(t', I', \texttt{C}')$ specified by the `catch` clause takes priority over default handler h_d. Handler code at $\texttt{C}'$ is only executed when an exception compatible with type t' is thrown. In that case, after executing the `catch` block, a `goto` instruction is typically used to jump to the `finally` block at `F`. Because this mechanism is a straightforward augmentation of catching any exception by h_d, this causes no new problems for subroutine inlining and verification. Hence `catch` blocks are not discussed further in this chapter.

7.2.3 Finally Blocks and Subroutines

A `finally` block can be executed in two modes: either an exception terminated its `try` block prematurely, or no exception was thrown. The only difference is therefore the "context" in which the block executes: it possibly has to handle an exception e. This lead to the idea of sharing the common code of a finally block. Thus a Java compiler typically implements `finally` blocks using *subroutines.*[2] A subroutine S is a function-like block of code. In this chapter, S will refer to the entire subroutine while `S` denotes the address of the first instruction of S. A subroutine can be called by a special jump-to-subroutine instruction `jsr`, which pushes the successor of the current address onto the stack. The subroutine first has to store that address in a register r, from which it is later retrieved by a return-from-subroutine instruction `ret`. Register r cannot be used for computations. Java compilers normally transform the entire `finally` block into a subroutine. This subroutine is called whenever needed: after normal execution of the `try` block, after exceptions have been taken care of with `catch`, or when an uncaught exception occurs.

The example in Figure 7.1 illustrates this. Range R which handler $h(t, R, \texttt{C})$ protects is marked by a vertical line. The handler code at `C` first stores the exception reference in a local variable e. It then calls the `finally` block at `S`. After executing S, the exception reference is loaded from variable e and thrown to the caller using instruction `athrow`. If no exception occurs, S is called after the `try` block, before continuing execution at `X`. Note that the subroutine block is inside the entire method, requiring a `goto` instruction to continue execution at `X`, after the `try` block. In the control flow graph, S can be treated as a simple block of code which can be called from the top level of the method (main) or exception handler code `C`. In the first case, S will return (with `ret`) to instruction `goto X`, otherwise to the second part of the handler ending with `athrow`.

[2]Sun's J2SE compilers, version 1.4.2 and later, compile `finally` blocks without subroutines. However, in order to ensure backward compatibility with legacy bytecode, the bytecode verifier still has to deal with the complexity of allowing for correct subroutines. This underlines the need for subroutine elimination, as commercial libraries often do not use the latest available compiler but can still be used in conjunction with programs compiled by them. This chapter lays the groundwork for inlining subroutines in legacy bytecode, allowing bytecode verifiers in future VMs to ignore this problem.

```
int m(int i) {
  try {
     i++;
  } finally {
     i--;
  }
  return i;
}
```

```
         |    iinc i 1
     (h) |    jsr S
         |    goto X
           C: astore e
              jsr S
              aload  e
              athrow
           S: astore r
              iinc i -1
              ret r
           X: iload i
              ireturn
```

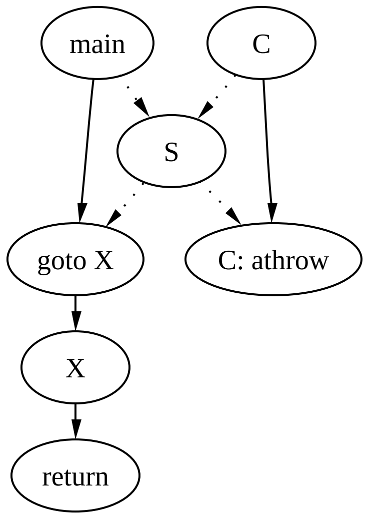


***Figure 7.1**: A simple finally block, its bytecode and its control flow graph.*

7.2.4 Nested Subroutines

The example in Figure 7.2 from [SSB01, Chapter 16] illustrates difficulties when dealing with subroutines. It contains a nested `finally` block with a `break` statement.[3] The compiler transforms this into two exception handlers $h_1(t_1, R_1, \mathtt{C}_1)$ and $h_2(t_2, R_2, \mathtt{C}_2)$ using two subroutines S_1 and S_2, where it is possible to return directly to the enclosing subroutine from the inner subroutine, without executing the `ret` statement belonging to the inner subroutine. Letter e denotes a register holding a reference to an exception, r a register holding the return address of a subroutine call.

The corresponding control flow graph in Figure 7.3 is quite complex. Its two exception handlers h_1 and h_2 contain one `finally` block each. The first `finally` block contains a while loop with test `W` and loop body `L` . If the loop test fails, S_1 returns via `X` to the successor of its caller. This may be the second instruction, or code after $\mathtt{C}_1$, which throws exception e_1 after having executed S_1. Loop body `L` contains in inner `try`/`finally` statement, compiled into exception handler h_2. Execution of `L` results in calling inner `finally` block at $\mathtt{S}_2$, again prior to the `return` statement. This block will test `b` and break to the outer subroutine, which is represented by connection $\mathtt{S}_2 \rightarrow \mathtt{X}$. If `b` was false, the inner subroutine would return normally using its `ret` instruction at `Y`. There, control will return to the inner `return` statement within `L`, which then returns from the method. Both `try` blocks are also protected by default exception handlers, where the control flow is similar. The main difference is that an exception will be thrown rather than a value returned.

[3]The body of the method does not contain any semantically relevant operations for simplicity. The resulting code, compiled by Sun's J2SE 1.3 compiler, includes a handler protecting a `return` statement, even though that instruction cannot throw an exception. The handler may come into effect if the `try` block contains additional instructions. Therefore it is preserved in this example.

```
static void m(boolean b) {
  try {
    return;
  } finally {
    while (b) {
      try {
        return;
      } finally {
        if (b) break;
      }
    }
  }
}
```

```
(h1) |     jsr S1
     |     return
       C1: astore e1
           jsr S1
           aload e1
           athrow
       S1: astore r1
           goto W
(h2) |  L: jsr S2
     |     return
       C2: astore e2
           jsr S2
           aload e2
           athrow
       S2: astore r2
           iload b
           ifne X
        Y: ret r2
        W: iload b
           ifne L
        X: ret r1
```

Figure 7.2: *Breaking out of a subroutine to an enclosing subroutine.*

7.3 Inlining Java Subroutines

Once all subroutines with their boundaries have been found, they can be inlined. Inlining usually increases the size of a program only sightly [Fre98] but significantly reduces the complexity of data flow analysis [Fre98, SSB01].

Table 7.2 defines potential successors of all bytecode instructions covered here. Without loss of generality, it is assumed that instructions are numbered consecutively. Thus $pc + 1$ refers to the successor of the current instruction, $pc - 1$ to its predecessor. Conditional branches (`ifne`) are treated non-deterministically. The `jsr` instruction is modeled to have two successors because control returns to $pc + 1$ after execution of the subroutine at a. Certain instructions leave the scope of the current method (`return`, `athrow`) or continue at a special address (`ret`).

Instruction (at address pc)	Addresses of possible successors
`aload, iload, astore, istore, iinc`	$\{pc+1\}$
`goto` a	$\{a\}$
`ifne` a, `jsr` a	$\{a, pc+1\}$
`ret, athrow, return`	$\{\}$

Table 7.2: *Possible successors of Java bytecode instructions.*

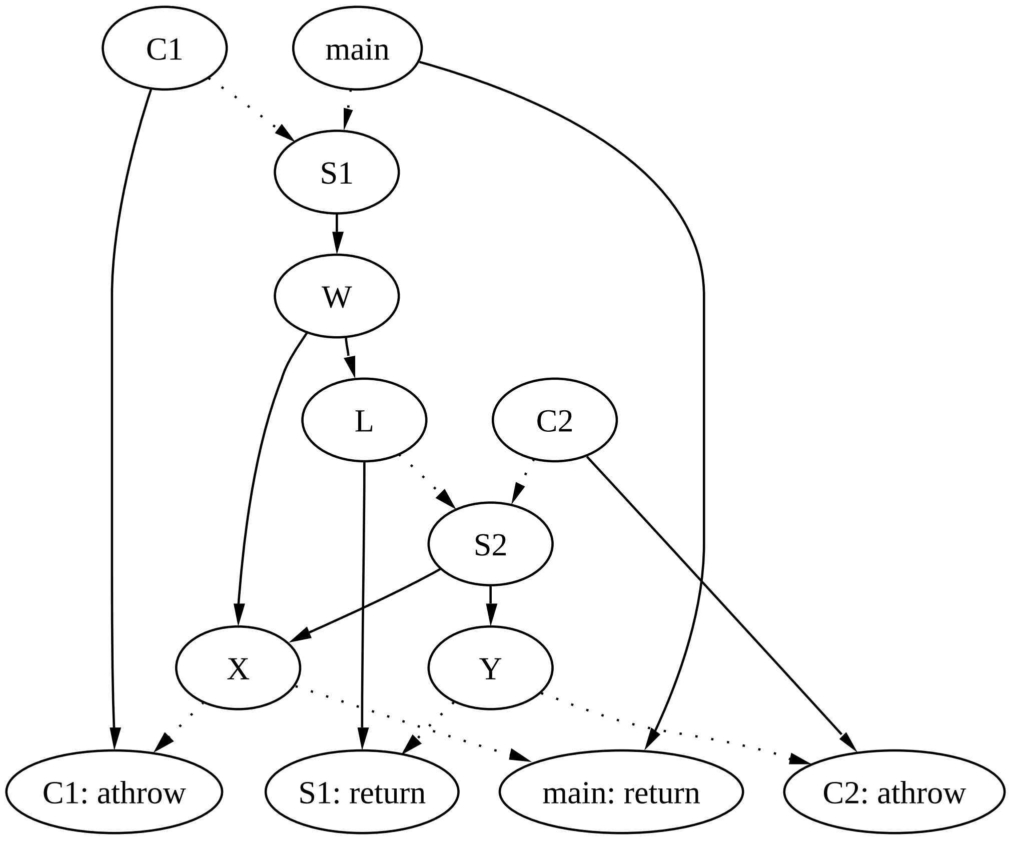


Figure 7.3: *Control flow graph of nested subroutines.*

The first instruction of a method is assumed to have code index 0. A code index i is *reachable* if there exists a sequence of successors from instruction 0 to i. S is a subroutine iff i is reachable and code(i) is `jsr S`. A code index `X` is a *possible return from a subroutine* if code(`S`) is `astore` r, code(`X`) is `ret` r, and `X` must be reachable from `S` on a path that does not use an additional `astore` r instruction. A code index i *belongs to subroutine* S, $i \in S$, if there exists a possible return `X` from that subroutine `S` such that $\texttt{S} \leq i \leq \texttt{X}$. The *end of a subroutine* S, eos(S), is the highest index belonging to S. Note that this definition avoids the semantics of nested exception handler ranges, thus covering each nested subroutine individually.

For the purpose of inlining, we also need the following definitions: The *body* of a subroutine is the code which belongs to a subroutine S, where for each code index i, $\texttt{S} < i < \text{eos}(S)$ holds. This means the body does not include the first instruction, `astore` r, and the last instruction, `ret` r. A subroutine S_2 is *nested* in S_1 if for each code index i which belongs to S_2, $i \in S_1$ holds. From this, $\texttt{S}_1 < \texttt{S}_2$ and $\text{eos}(S_1) > \text{eos}(S_2)$ follows. Furthermore, code($\texttt{S}_2 - 1$) must be instruction `goto` $\text{eos}(S_2) + 1$. A subroutine S_1 is *dependent on* a (possibly nested) subroutine S_2, $S_1 \prec S_2$, if there exists an instruction `jsr` $\texttt{S}_2$ which belongs to subroutine S_1, where $S_2 \neq S_1$. Dependencies are transitive.

A subroutine S_1 which depends on S_2 must be inlined after S_2. When S_1 is inlined later, the calls to S_2 within S_1 have already been replaced by the body of S_2. Other than that, the order in which subroutines are inlined does not matter. During each inlining step, all calls to one subroutine S are inlined.

7.3.1 Sufficient and Necessary Well-formedness Conditions

Java bytecode can only be inlined if certain well-formedness conditions hold. A set of necessary conditions is given by the specification of bytecode verification, which includes that subroutines must have a single entry point and that return addresses cannot be generated by means other than a `jsr` instruction [LY99]. Beyond these given conditions, extra conditions have to hold such that a subroutine can be inlined. Note that it is not possible that programs generated by a Java compiler violate these conditions, except for a minor aspect concerning JDK 1.4, which is described below. Furthermore, artificially generated, "malicious" bytecode that does not fulfill these well-formedness criteria will likely be rejected by a bytecode verifier. Bytecode verification is in essence an undecidable problem, and thus verifiers only allow for a subset of all possible bytecode programs to pass [LY99, SSB01].

One extra condition not described here arises from the current artificial size limit of 65536 bytes per method [LY99]. Other limitations are structural conditions that bytecode has to fulfill. Given here is an abridged definition taken from [SSB01]:

Boundary. Each subroutine S must have an end eos(S).
If subroutine S does not have a `ret` statement, then all instances of `jsr S` can be replaced with `goto S`, and no inlining is needed.

No recursion. A subroutine cannot call itself.

Correct nesting. Subroutines may not overlap:
$\nexists S_1, S_2 \cdot \mathtt{S}_1 < \mathtt{S}_2 < \mathrm{eos}(S_1) < \mathrm{eos}(S_2)$.

No mutual dependencies. If $S_i \prec S_j$, there must be no dependencies such that $S_j \prec S_i$. Note this property is not affected by nesting.

Exception handler containment. If code `C` of a handler $h(t, R, \mathtt{C})$ belongs to S, then its entire range R must belong to S as well: $\forall h(t, R, \mathtt{C}), S \cdot (\mathtt{C} \in S \rightarrow R \subseteq S)$.

Handler range containment. If any $i \in R$ of a handler $h(t, R, \mathtt{C})$ belongs to S, then its entire range R must belong to S: $\forall h(t, R, \mathtt{C}), S \cdot (\exists i \in R \cdot i \in S \rightarrow R \subseteq S)$.

Subroutine containment in handler range.
If the entire range R of a handler $h(t, R, \mathtt{C})$ belongs to S, then any instructions `jsr S` must be within R: $\forall h(t, R, \mathtt{C}), S \cdot (R \subseteq S \rightarrow (\forall i \cdot \mathrm{code}(i) = \mathtt{jsr\,S} \rightarrow i \in R))$.

For the last six conditions, Figure 7.4 shows an example violating it. Existing Java compilers do not violate them except as described in Subsection 7.3.5.

7.3.2 Control Transfer Targets

When inlining subroutines, the body of a subroutine S replaces each subroutine call `jsr S`. This part of inlining is trivial, as shown by the example in Figure 7.5. The two inlined copies of S which replace the `jsr` instructions are shown in **bold** face. Difficulties arise with jump targets, which have to be updated after inlining. Inlining eliminates `jsr` and `ret` instructions; therefore any jumps to these instructions are no longer valid.

Furthermore, there can be jumps inside a subroutine to an enclosing subroutine or the top level of the code. Such a case is shown in Figure 7.6, which depicts the inlining of the nested subroutines in Figure 7.2. Therefore, the inlining algorithm has to update jump targets during several inlining steps and also to consider copies of instructions, for each instance of a subroutine body that gets inlined.

The algorithm uses two *code sets,* the *current* set B and the *new* set B'. During each inlining step, all instructions in B are moved and possibly duplicated, creating a new set of instructions B' which becomes input B of the next inlining step.

Each address in B must map onto an *equivalent* address B'. Each possible execution (including exceptional behavior) must execute the same sequence of operations, excluding `jsr` and `ret`, in B and B'. Code indices in B referring to `jsr` or `ret` instructions must be mapped to equivalent indices in B'. The most straightforward solution is to update all targets each time after inlining *one* instance of a given subroutine. This is certainly correct, but also very inefficient, because it would require updating targets once for each `jsr` instruction rather than each subroutine.

Instead, our algorithm inlines *all* instances of a subroutine in one step, with the innermost subroutines being inlined first. Instructions not belonging to the subroutine which is being inlined and which are not a `jsr S` operation are copied over from B to B'. Each occurrence of `jsr S` is replaced with the body of S. In order to update code addresses, the

```
       S: astore r
          ...
          jsr S
          ...
  eos(S): ret r
```

No recursion.

```
      S1: astore r1
          ...
      S2: astore r2
          ...
 eos(S1): ret r1
          ...
 eos(S2): ret r2
```

Correct nesting.

```
      S1: astore r1
          ...
          jsr S2
          ...
 eos(S1): ret r1
          ...
      S2: astore r2
          ...
          jsr S1
          ...
 eos(S2): ret r2
```

No mutual dependencies.

```
       | :
     h |   ...
       | :
          ...
       S: astore r
          ...
       C:
          ...
  eos(S): ret r
```

Exception handler containment.

```
     | :
     |    ...
   h | S: astore r
     |    ...
     | :
          ...
  eos(S): ret r
          ...
       C:
```

Handler range containment.

```
          jsr S
          ...
    |  :
    |     ...
  h |  S: astore r
    |     ...
 eos|(S): ret r
    |     ...
    |  :
```

Subroutine containment in handler range.

Figure 7.4*: Instruction sequences violating well-formedness conditions.*

```
    |    iinc i 1
(h) |    jsr S
    |    goto X
      C: astore e
         jsr S
         aload  e
         athrow
      S: astore r
         iinc i -1
         ret r
      X: iload i
         ireturn
```

```
(h1) |   iinc i 1
         iinc i -1
(h2) |   goto X
      C: astore e
         iinc i -1
         aload  e
         athrow
      X: iload i
         ireturn
```

Figure 7.5: *Inlining a subroutine.*

```
(h1) |      jsr S1
     |      return
       C1:  astore e1
            jsr S1
            aload e1
            athrow
       S1:  astore r1
            goto W
(h2) |  L:  jsr S2
     |      return
       C2:  astore e2
            jsr S2
            aload e2
            athrow
       S2:  astore r2
            iload b
            ifne X
        Y:  ret r2
        W:  iload b
            ifne L
        X:  ret r1
```

```
(h1) |      jsr S1
     |      return
       C1:  astore e1
            jsr S1
            aload e1
            athrow
       S1:  astore r1
            goto W
        L:  iload b
            ifne X
(h2) |      return
       C2:  astore e2
            iload b
            ifne X
            aload e2
            athrow
        W:  iload b
            ifne L
        X:  ret r1
```

Figure 7.6: *Inlining a nested subroutine: Step 1.*

algorithm uses a *mapping* M, a relation $I \times I'$ of code indices mapping an index $i \in I$ to a set of indices $\{i'_0, i'_1, \ldots, i'_k\} \in I'$. This relation, initially empty, records how an address in B is mapped to one or several addresses in B'. Each time an instruction at index i is moved or copied from the current code set B to the new code set B' at index i', $i \mapsto i'$ is added to the mapping.

The key to handling jumps to ins_j, the `jsr S` instruction itself, and to ins_r, the `ret` instruction in the subroutine, is adding two extra entries to M. The first one is $i_j \rightarrow i'_0$ where $i_j = \text{index}(ins_j)$ and $i'_0 = M(\texttt{S})$, the index where the first instruction of the subroutine has been mapped to. The second one is $i_r \rightarrow i'_r$ where $i_r = \text{index}(ins_r)$ and $i'_r = M(\text{eos}(S)+1)$, the index of the first instruction *after* the inlined subroutine.

In the following discussion, a *forward jump* is a jump whose target code index is greater than the code index of the instruction. Similarly, a *backward jump* is a jump leading to a smaller code index. If bytecode fulfills the correctness criteria described above, the correctness of the algorithm can be proved as follows:

- A target outside S is mapped to a single target and therefore trivially correct.

- A target in the body of S is mapped to several targets in the inlined subroutines S', S'' etc., one for each `jsr S` in B. Let a jump instruction in B be at code index i and its target at a. Given i', the index of the jump instruction in B', the *nearest* target in the current mapping has to be chosen which still preserves the fact that a jump is either a forward or backward jump.
  For a forward jump, index $\min_{a'} \cdot (a \mapsto a' \in M) \wedge (a' > i')$ is the correct index. This can be shown as follows: Address a is either outside S, in which case the $code(a)$ has not been duplicated and there is only one $a' \cdot a \mapsto a' \in M$. If a is inside S, a' is necessarily the nearest target to i' in that direction: The code at index a has been copied to a' during the inlining of S to S'. The first instruction of the inlined copy of S' is at index $\texttt{S}'_\alpha$ and the last instruction is at $\texttt{S}'_\omega$. Since i belongs to S, $\texttt{S}'_\alpha \leq i' \leq \texttt{S}'_\omega$ holds. No other code than S' has been copied to positions inside that interval, and $\texttt{S}'_\alpha \leq i' < a' \leq \texttt{S}'_\omega$ holds because a belongs to S and the jump is a forward jump. Any other copies of the instructions at a are either copied to an index $a'' < \texttt{S}'_\alpha$, and therefore $a'' < i'$, or $a'' > \texttt{S}'_\omega$, and therefore $a'' > a'$. Backward jumps are treated vice versa.

- A jump to a `jsr S` instruction in B indirectly executes code at `S`. Mapping it to the $\texttt{S}'_\alpha$ preserves the semantics.

- A jump to the last instruction in a subroutine will return to the successor of its `jsr S` instruction. Therefore mapping the code index of the `ret` instruction to the successor of the last inlined instruction of the body of S produces the same effect in the inlined code. Note that there always exists an instruction following a `jsr` instruction [LY99], such as `return`.

Two of these cases are shown in the second inlining step of the nested subroutine, shown in Figure 7.7. In both inlined instances of S_1, the outer subroutine, there is a jump to `W` inside the subroutine and to X, the index of the ret instruction of S_1. By inlining S_1, both code indices are mapped to two new indices, $\{W_1,W_2\}$, and $\{X_1,X_2\}$, respectively. The semantics of jumps are preserved as described above.

```
(h1) |      jsr S1
     |      return
        C1: astore e1
            jsr S1
            aload e1
            athrow
        S1: astore r1
            goto W
         L: iload b
            ifne X
(h2) |      return
        C2: astore e2
            iload b
            ifne X
            aload e2
            athrow
         W: iload b
            ifne L
         X: ret r1
```

```
                goto W1
           L1:  iload b
                ifne X1
(h2_1) |        return
          C2_1: astore e2
                iload b
                ifne X1
                aload e2
                athrow
           W1:  iload b
                ifne L1
(h1)   |   X1:  return
           C1:  astore e1
                goto W2
           L2:  iload b
                ifne X2
(h2_2) |        return
          C2_2: astore e2
                iload b
                ifne X2
                aload e2
                athrow
           W2:  iload b
                ifne L2
           X2:  aload e1
                athrow
```

***Figure 7.7**: Inlining a nested subroutine: Step 2.*

7.3.3 Exception Handler Splitting

If a jsr S instruction ins_j is protected by an exception handler $h(t, R, \mathtt{C})$, where $R = [r_\alpha, r_\omega]$ does *not* extend to the subroutine itself, then that handler must *not* be active for the inlined subroutine. A simple example is shown in Figure 7.5, where the jsr instruction is in the middle of the exception handler range. Therefore, to solve this problem, the exception handler must be *split* into two handlers $h_1(t, R_1, \mathtt{C}')$ and $h_2(t, R_2, \mathtt{C}')$. The new ranges are $R_1 = [r'_a, r_\beta]$ and R_2, with $r'_\alpha = M(r_\alpha)$ and $r_\beta = M(index(ins_j) - 1)$, the mapped code index of the predecessor of the jsr instruction. In $R_2 = [r_\gamma, r'_\omega]$, $r_\gamma = M(index(ins_r))$, the mapped code index of the successor of the last instruction of the inlined subroutine body, and $r'_\omega = M(r_\omega)$.

Splitting handlers is necessary to ensure correctness of the inlined program. There exist cases where R_1 or R_2 degenerates to an interval of length zero and can be removed altogether. This is the case in Figure 7.7 where handler h_1 does not protect any instructions prior to the `jsr` instruction. Splitting may increase the number of exception handlers exponentially in the nesting depth of a subroutine. This depth is almost never greater than one, though, and only few exception handlers are affected by splitting.

7.3.4 Exception Handler Copying

If a subroutine S, but not the `jsr S` statement, is protected by an exception handler, this protection also has to be ensured for the inlined copy of the subroutine. Therefore, all exception handlers protecting subroutine S have to be *copied* for each inlined instance of S. Figure 7.7 shows a case where inlining the outer subroutine S_1 causes the exception handler h_2 inside that subroutine to be duplicated.

Note that this duplication does not occur if *both* the `jsr` instruction and the subroutine are protected by the same handler. In this case, the inlined subroutine is automatically included in the mapped handler range. Copying handlers may increase the number of handlers exponentially, which is not an issue in practice because the *innermost* subroutine, corresponding to the innermost `finally` block, is never protected by an exception handler itself, reducing the exponent by one.

7.3.5 Violation of Well-formedness Conditions in JDK 1.4

The original implementation exhibited problems with some class files compiled with Sun's JDK 1.4 compiler. The reason were changes in the compiler, designed to aid the bytecode verifier of the VM in cases where older versions used to reject correct programs generated by Sun's own compiler.

When compiling the program from Figure 7.1 with JDK 1.4 instead of 1.3, the resulting instructions are the same, but the exception handlers are different: The original handler covered three instructions, the initial increment instruction, the `jsr`, and the `goto` which jumps to the end of the program. The handler from the 1.4 compiler does not protect the `goto` instruction. This does not change the semantics of the code because the `goto` instruction cannot raise an exception.

However, a *second* handler $\overline{h}$ is installed by the newer compiler, which covers the first two instructions of the exception handler code (at label `C`), `astore` *e* and the second instance of `jsr S`. The situation is exacerbated by the fact that $\overline{h}$ is recursive; the handler code has the same address as the first instruction protected by it. This could (theoretically) produce an endless loop of exceptions. The result of inlining $\overline{h}$ is a handler covering only the `astore` instruction (since the inlined subroutine is outside the handler range). Fortunately, the `astore` instruction cannot throw an exception, so no changes are needed in the VM to avoid a potentially endless loop.

Newer JDK compilers (1.4.2 and later) generate subroutines in-place. The result is identical, including spurious handler $\overline{h}$, to code from JDK 1.4 where our inlining algorithm has been applied.

7.3.6 Costs of Inlining

Inlining subroutines increases code size only slightly. Subroutines are rare. In Sun's Java run-time libraries (version 1.4.1), out of all 64994 methods from 7282 classes (ignoring 980 interfaces), only 966 methods (1.5 %) use 1001 subroutines. None of them are nested. Table 7.3 shows that subroutines are usually called two to three times each, with a few exceptions where a subroutine is used more often.

Number of calls	1	2	3	4	5	6 – 10	11 – 20	28
Number of subroutines	1	783	173	23	9	8	3	1

Table 7.3: *Number of calls per subroutine, determining how often its code is inlined.*

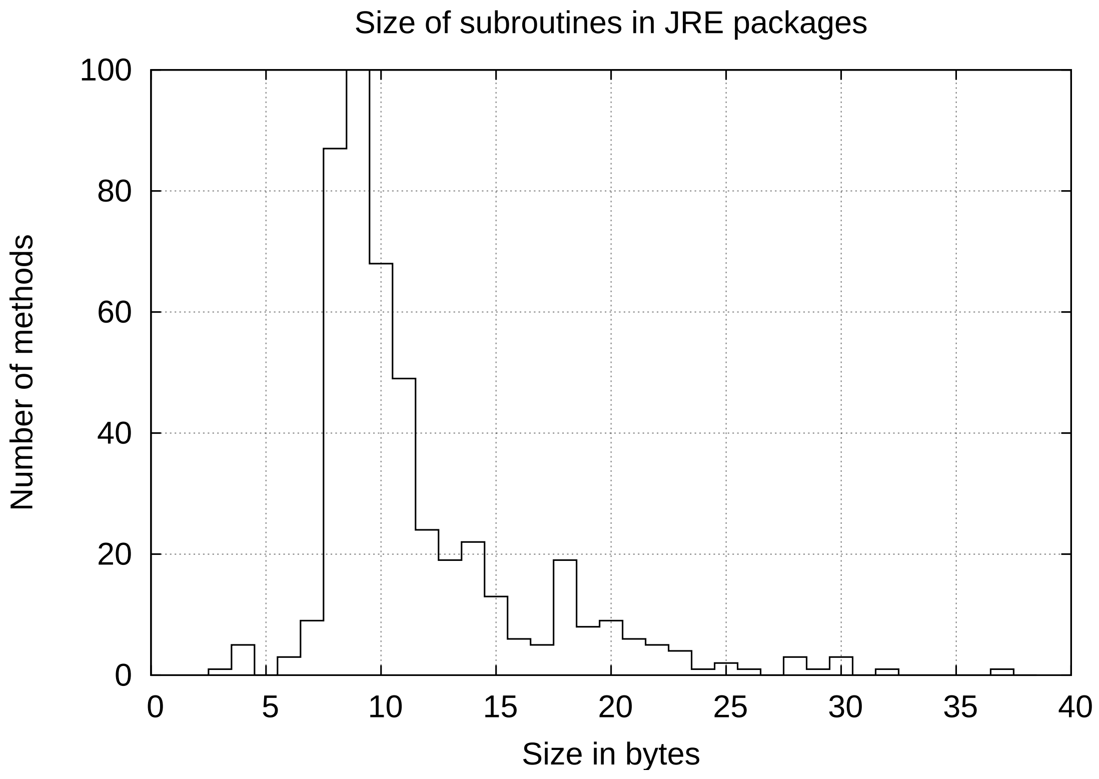


Figure 7.8: *Sizes of subroutines in all JRE packages of Java 1.4.1.*

The histogram in Figure 7.8 shows that most subroutines measure only between 8 and 12 bytes; 626 subroutines were 9 bytes large, hence that entry is off the scale. No subroutine was larger than 37 bytes. Inlining usually results in a modest code growth of less than 10 bytes. This is shown by the histogram in Figure 7.9 where entries with an even and odd number of bytes are summarized in one bucket. Entries off the scale are values 0 (64041 methods, including those without subroutines) and 2, representing 571 methods where code size increased by 2 or 3 bytes. 10 methods grew by more than 60 bytes, 186 bytes being the worst case. Inlining all subroutines of JRE 1.4.1 would result in a code growth of 5998 bytes, which is negligible compared to the entire run-time library, measuring 25 MB.

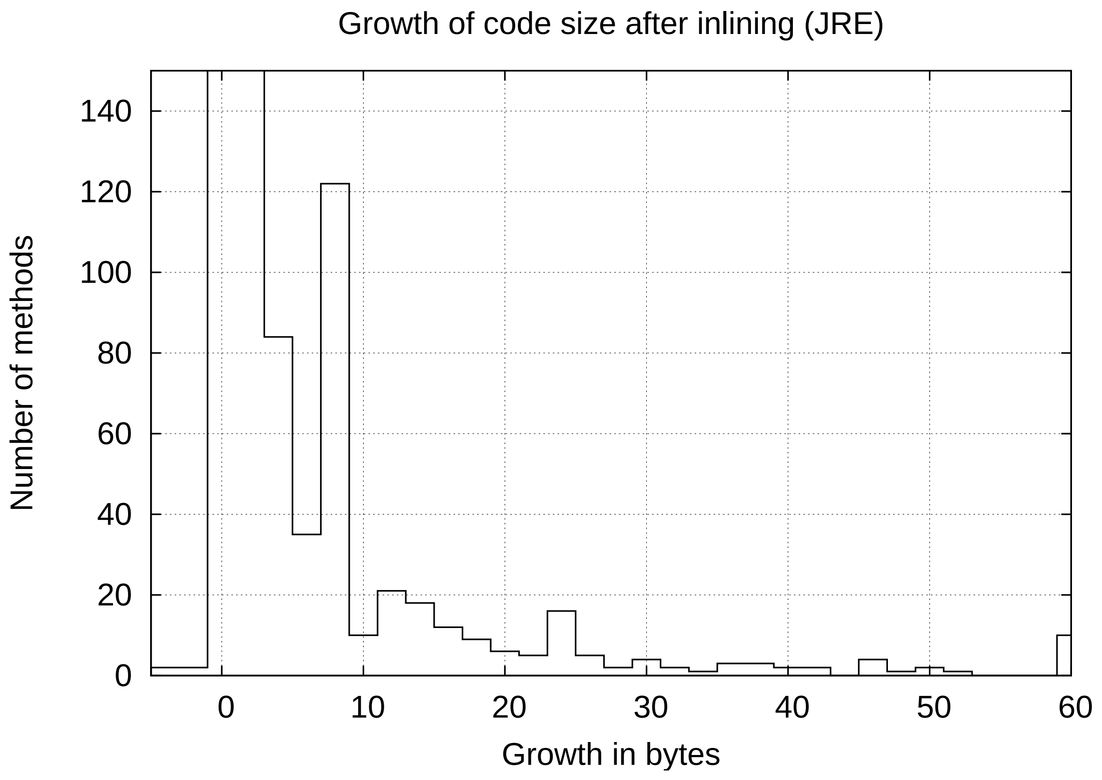


***Figure 7.9**: Size increase after inlining all subroutines in each method.*

7.4 Abstract, Register-based Bytecode

Java bytecode contains 201 instructions [LY99], many of which are variants of the same type. For instance, 25 instructions load a register on the stack. Variants include several instructions for each data type, one generic variant (e.g. `iload` *r*) and short variants like `aload_0`, where *r* is hard-coded. A reduction of the instruction set is an obvious simplification. We use abstract bytecode [SSB01] as the reduced format, where argument types and hard-coded indices are folded into the parametrized version of a generic instruction. For instance, `aload_0` becomes `Load` "ref" 0. This reduction is independent of bytecode inlining. The previous section described inlining using normal bytecode to allow for stand-alone inlining algorithms.

Instructions not implemented in [SSB01] include arithmetic instructions. Implementation of these instructions is straightforward. Unsupported instructions are `switch` (for control flow), `monitorenter` and `monitorexit` (for multi-threading), and the `wide` instruction that modifies the parameter size of the subsequent instruction. The first three instructions have to be implemented according to the standard bytecode semantics [LY99] while the `wide` instruction is an artefact of the fact that Java bytecode was initially targetted to embedded systems with little memory for instructions. In our implementation [ASB+04] of the abstract bytecode instruction set, we extended the size of any instruction parameters to four bytes and thus could eliminate the wide instruction trivially, by converting all instruction arguments to a four-byte format.

Abstract bytecode only has 31 instructions, which is already a great simplification of the original instruction set. However, the usage of a (fixed-size) stack makes data flow analysis needlessly difficult, since the exact stack height at each index, though known at compile-time, has to be computed first after loading a class file. This computation is normally part of bytecode verification in the class loader. Furthermore, the treatment of stack and local variables (registers) results in pairs of instructions that essentially perform the same task: `load` pops the top element from the stack while `store` pushes a register onto the stack. Finally, 64-bit values are treated as a single stack element, but as a pair of local variables. This creates a need for case distinctions for many instructions [LY99]. The specification requires that the second slot of the local variables holding a 64-bit value is never used, and that the stack semantics are preserved when pushing a 64-bit value onto it.

Because the height of the stack is statically known for each instruction, we converted the stack-based format of abstract bytecode to a register-based representation where each stack element is converted to a register. When using registers, stack elements and local variables can be treated uniformly, merging `Load` and `Store` into a `Get` instruction, and eliminating more instructions such as `Pop`, `Swap`, or `Dup`.

Of all conversions, converting the `Dup` instruction was the only non-trivial one and actually proved to be quite difficult. Some variants of this instruction do not only copy the top element(s) of the stack, but insert it "further down", below the top element. There exist six variants of `Dup` instructions, and the treatment of data flow requires up to four case distinctions per instruction variant, due to 64-bit values [LY99]. We convert all `Dup` instructions into an equivalent series of `Get` instructions. This unfortunately introduces sequences of instructions that corresponds to only one original instruction, which makes further treatment slightly more complex; but it still eliminates the case distinctions for 64-bit values, which is the greater overhead.

The conversion to abstract, RISC-like instructions reduces the size of the final instruction set to only 25 instructions. The final instruction set comprises (for the exact semantics, refer to [LY99, SSB01]): `ALoad`, `AStore`, `ArrayLength`, `Athrow`, `Checkcast`, `Cond`, `Const`, `Get`, `GetField`, `GetStatic`, `Goto`, `Inc`, `Instanceof`, `InvokeSpecial`, `Invoke-Static`, `InvokeVirtual`, `MonitorEnter`, `MonitorExit`, `New`, `NewArray`, `Prim`, `Put-Field`, `PutStatic`, `Return`, `Switch`. This instruction set was used in JNuke and has been tested in over 1,000 unit and system tests using static analysis, run-time verification, and software model checking [AB05a, ABH04, ASB+04].

Bytecode variant	Java [LY99]	Abstract [SSB01]	Register-based
Instruction set size	201	31	25
Variants (type/index) per instruction	up to 25	1	1
Bytecode subroutines	yes	yes	no
Wide instructions	yes	not impl.	eliminated
Special treatment of 64-bit values	yes	not impl.	eliminated
Register location	implicit	implicit	explicit

***Table 7.4**: The benefits of register-based bytecode.*

7.5 Related Work

Previous work has investigated difficulties in analyzing Java bytecode arising from its large instruction set and subroutines. Inlining bytecode subroutines has been investigated in the context of just-in-time-compilation [LYK$^+$00] or as a preprocessing stage for theorem proving [Fre98]. The latter paper also describes an alternative to code duplication for inlining: by storing a small unique integer for each subroutine call instruction in an extra register, subroutines can be emulated without using a `jsr` instruction. However, the size saving by this strategy would be small, and bytecode verifiers would again have to ensure that the content of this extra register is never overwritten inside the subroutine, leaving one of the major problems in bytecode verification unsolved. Therefore this direction was never pursued further.

Challenges in code analysis similar to those described here occur for *decompilation,* where the structure of subroutines must be discovered to determine the correct scope of `try`/`catch`/`finally` blocks. The Dava decompiler, which is part of the Soot framework, analyzes these structures in order to obtain an output that correctly matches the original source program [MH02]. Soot also eliminates `jsr` instructions through inlining [VR00]. However, no algorithm is given. Details on how to handle nested subroutines are missing.

As a part of work on μJava [KN03], another project also performs a kind of subroutine inlining called subroutine *expansion* [Wil02]. The main difference is that the expanded code still contains `jsr` instructions, making it easier to ensure correctness of the inlined code, but still posing a certain burden on the bytecode verifier that our work eliminates. The inlining algorithm differs in several points. First, it uses “complex addresses” to track code duplication. Second, it does not inline subroutines in the order of their nesting. This has two side-effects: treatment of nested subroutines creates a very complex special case, and the expanded code may be larger than necessary [Wil02]. Our algorithm uses a simple mapping instead of complex addresses, which, together with inlining subroutines in the order in which they are nested, greatly simplifies the adjustment of branch targets and exception handler ranges. Furthermore, with nesting taken care of by inlining subroutines in nesting order, no special treatment of nested subroutines is necessary in the inner loop that performs the actual inlining.

Instruction set reduction on Java bytecode has been performed in other projects in several ways. The Carmel [Mar01] and Jasmin [MD97] bytecode instruction sets both use a reduced instruction set similar to abstract bytecode [SSB01]. The Bytecode Engineering Library (BCEL) does not directly reduce the instruction set but features an object-oriented representation of bytecode instructions where super classes combine related instructions [Dah03]. The project most similar to ours with respect to instruction abstraction is Soot. The Jimple language from Soot is a bytecode-like language using 3-address code instead of stack-based instructions, making it suitable for analysis and optimization [VRHS$^+$99].

7.6 Summary

Java bytecode is far from ideal for program analysis. Subroutines, a construct not available in the Java language but only in Java bytecode, make data flow analysis very complex. Eliminating subroutines is difficult because subroutines can be nested, and they can overlap with exception handlers. In practice, inlining does not increase program size much, while greatly simplifying data flow analysis. This is especially valuable as subroutines are disappearing in modern compilers but still have to be supported by virtual machines for backward compatibility.

Abstracting sets of similar instructions to a single instruction greatly reduces the instruction set. Converting the stack-based operation semantics to a register-based one makes computational operands explicit and further reduces the instruction set. Finally, eliminating certain bytecode-specific issues, such as wide instructions and differences of 64-bit variables and stack elements, simplifies the code even further. The resulting instruction set was successfully used in the JNuke framework for static and dynamic analysis, which greatly benefits from the simplified bytecode format.

8 Implementation

This chapter describes the implementation-specific parts of this work, focusing on architectural aspects. Due to its novely, static analysis using a generic algorithm has been described in a previous chapter, Chapter 6. Section 8.1 gives an overview of all components present in JNuke. The idea of the observer architecture is described in Section 8.2. Two possible ways to implement this architecture, based on technologies described in Chapter 5, are shown here: A VM-based architecture, as used in JNuke, is documented in Section 8.3 while Section 8.4 outlines an instrumentation-based implementation, as used by JPaX [HR01, GH03]. The remainder of this chapter focuses on technical aspects of JNuke. Section 8.5 shows the division of JNuke into modules. How object-orientation in C was achieved is explained in Section 8.6 while Section 8.7 shows the testing environment used and experience gained with it.

8.1 Overview of JNuke

Figure 8.1 shows an overview the entire functionality implemented in JNuke. Its class loader takes well-formed Java bytecode [LY99] as input and allows execution in the JNuke Virtual Machine (VM).[1] The default scheduling policy of that VM may be replaced with a special scheduler that performs model checking. The model checker uses milestones to save the state of the VM in order to allow exploration of all possible successor states at locations where different thread switches may lead to different results in the program [ASB$^+$04, Eug03]. Schedules leading to an incorrect result can be used to instrument the original program, such that it will exhibit this particular faulty behavior when run again in a normal debugger [SBB04].

Alternatively, run-time verification (RV) algorithms may be used to operate on a single program trace. Dynamic analysis requires an execution environment, such as a Java Virtual Machine (VM). However, typical Java VMs only target execution and do not offer all required features, in particular, full state access. Code instrumentation, used by JPaX [HR01], can solve this problem to some extent only [GH03].

JNuke contains a specialized VM allowing for both backtracking and full state access [ASB$^+$04]. Custom checking algorithms can be implemented using an API that allows an algorithm to register for any event of interest. Finally, a static analysis module

[1]Malformed bytecode is recognized as such by the bytecode verification pass in the class loader, and rejected.

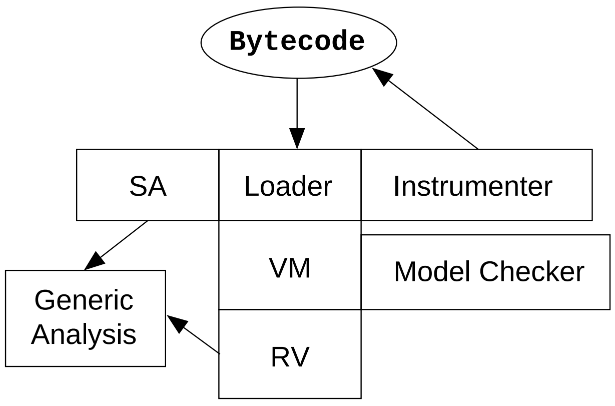


***Figure 8.1**: Overview of JNuke.*

has been added to JNuke [ABH04] and subsequently been augmented to a generic analysis, which can be used in a static or dynamic context [AB05a]. That aspect has been covered in Chapter 6. The remainder of this chapter focuses on the observer architecture and implementation aspects that apply to the entire project.

8.2 Observer Architecture

An observer-based architecture [GHJV95] was chosen to allow several run-time verification modules to supervise the same program trace. Furthermore, this architecture decouples event generation from event interpretation, which is crucial because the run-time environment is typically unaware of whether and how a program trace generated by it is interpreted.

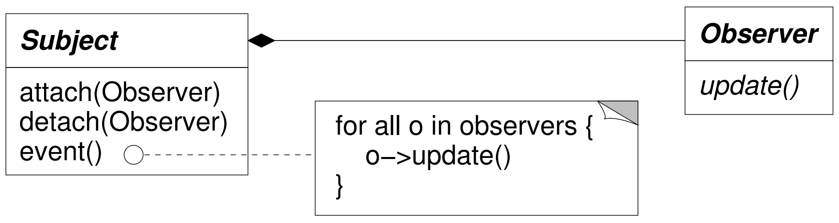


***Figure 8.2**: The generic observer pattern.*

Figure 8.2 outlines the principle of this design pattern: An observer implements a certain interface which allows notification of events updates. Such an update method may be polymorphic and take different kinds of events. In JNuke, the approach of having a larger interface was chosen, where an entire set of events is covered by it. This eliminates the requirement of writing one observer for each event type, of which there are currently ten, which are documented in Section 8.3. Figure 8.3 shows the slightly adapted design pattern. Note that it is not necessary to allow an observer to detach itself from the subject, as a run-time verification algorithm typically runs until a program has terminated. Instead, there is an event to notify the listener about program termination, in order to allow the verification algorithm to issue a final report and release internal resources.

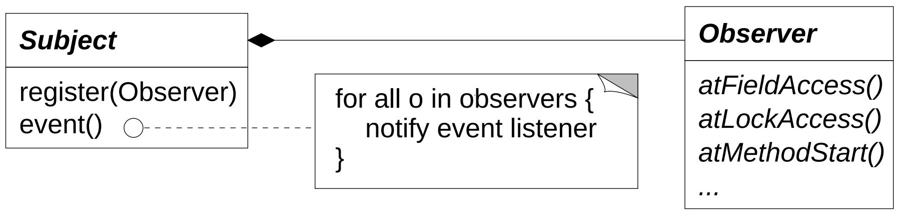


Figure 8.3: *Event observers in JNuke.*

Strictly speaking, the event observer as shown here is an fact the second stage of a two-stage observer. Low-level events are directly emitted by the JNuke VM. However, they do not have a consistent API and are directly tied to the issuing module of the VM (such as the lock manager for lock events). This is useful for implementing low-level functionality such as a modified scheduler but cumbersome for run-time verification.

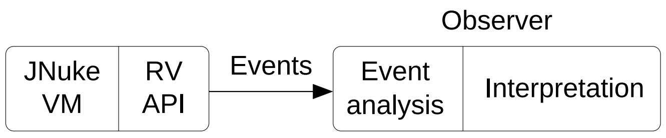


Figure 8.4: *The observer-based architecture for run-time verification.*

Figure 8.4 illustrates the two-stage approach to alleviate this problem: The first stage of the observer converts events to the format given by the RV API, while the second stage, event interpretation, acts as the observer shown in Figure 8.3. For full flexibility, the event observer consists of two stages: event analysis, which preprocesses events, and event interpretation, which contains the actual run-time verification algorithm.

8.3 VM-based Implementation: JNuke

As explained above, the goals of JNuke's run-time verification architecture are primarily twofold. The first goal is to decouple verification algorithms from the internal structure of the VM. This is achieved by using the observer design pattern. The second goal is to allow to use generic algorithms, which can also be used in a static context. This idea is described in detail in Chapter 6.

Figure 8.5 shows the scenario when using such a generic algorithm for run-time verification: An event analysis layer prepares data for the generic analysis algorithm. The dynamic environment maintains context data, which is updated internally (and invisibly to the generic algorithm). The generic algorithm is only notified of events it has explicitly

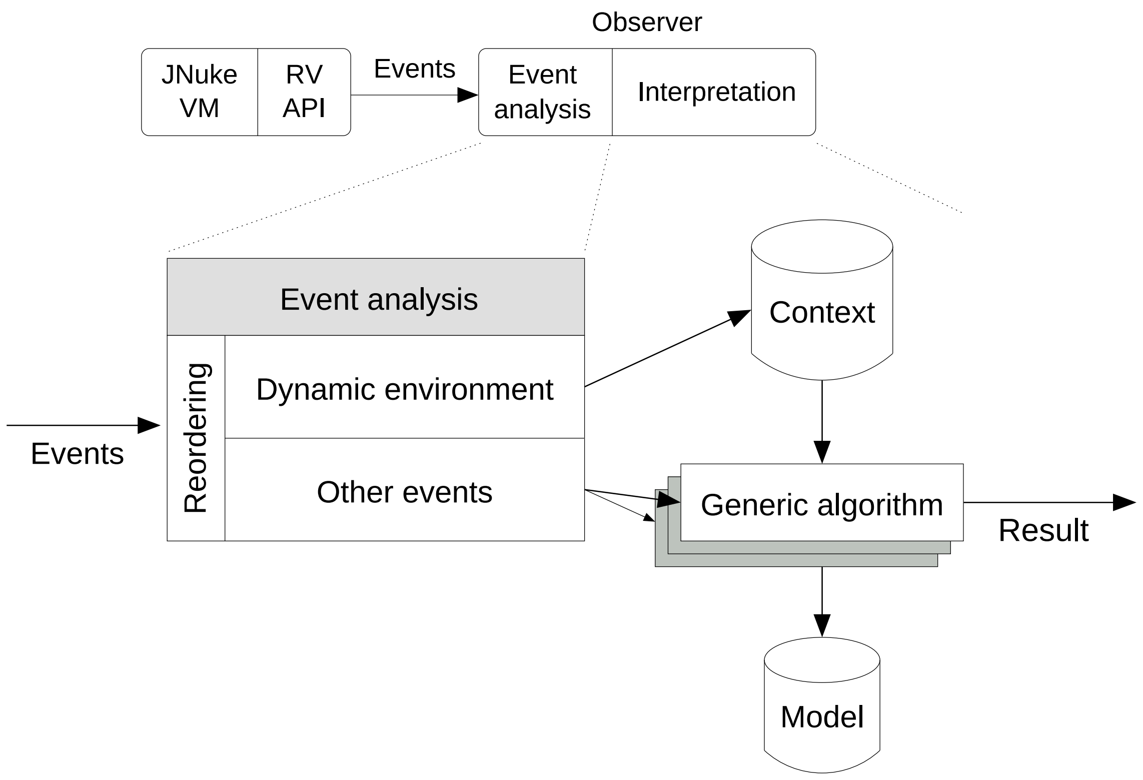


Figure 8.5: *Detailed view of run-time verification architecture.*

subscribed to. It uses context data provided by the dynamic environment to obtain its of model of the properties to be verified.

At the time of writing, only the block-local atomicity algorithm [ABH04] is implemented as such a generic algorithm. The other algorithms were implemented earlier. Therefore the Eraser low-level data race algorithm [SBN$^+$97] and the high-level data race algorithm [AHB03] are both implemented as "classical" run-time verification algorithms obtaining data directly from JNuke's RV API. When considering Figure 8.5 for the case of a classical run-time algorithm, the event analysis layer only consists of some generic helper modules that track history information, such as thread names or the last location where a certain lock was acquired. In that mode, the event analysis module therefore does not have the same format as the dynamic environment described in Chapter 6.

8.3.1 JNuke VM

Figure 8.6 shows the key components of the VM [Eug03]. The core parts are subsumed by the *run-time environment*, which controls the execution and effects of single instructions. It also loads classes on demand (using a linker module which is not shown in the figure). After each instruction, an exchangeable *scheduler* decides whether a thread switch should occur. If this is the case, the run-time environment puts the current thread to sleep and enables a new one. This action may involve updates on the lock sets of each thread, which is done via the *lock manager.* Inter-thread communications are queued by the *waitset manager,* while any heap content is updated by the *heap manager.* The heap manager allocates and frees data and is partially accessed directly by the run-time environment, partially by the *garbage collector.* The run-time environment can run with or without garbage collection, which was added at a later stage [Far04].

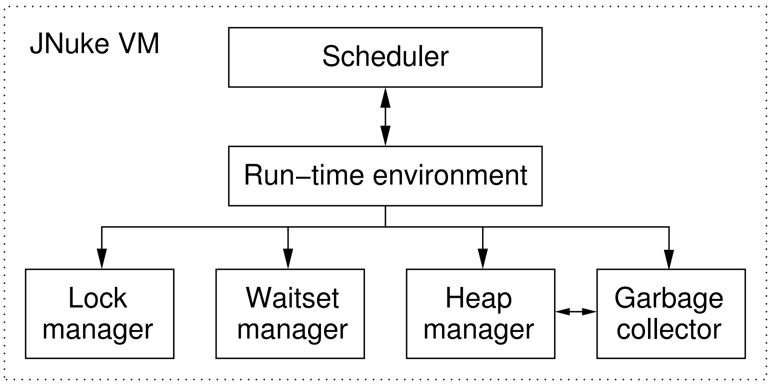


***Figure 8.6**: Overview of the key components of the JNuke VM.*

This modularization of the VM allows for a more flexible design. After an initial implementation, each module was augmented with a rollback capability which allows storing and restoring the entire state of the VM. This can be used to explore each possibility in the presence of non-determinism, such as non-determinism arising from thread switches [Bru99]. A special scheduler uses this to perform explicit-state model checking for Java programs [ASB$^+$04, Eug03]. These interfaces also ensure that native methods needed to perform system calls operate in a well-behaved manner and do not corrupt the Java heap.

8.3.2 Run-time verification API

The run-time verification API (RV API) allows algorithms to register *listeners* for events of interest, such as lock acquisition. These listeners are notified through an observer interface [GHJV95] whenever such an event occurs. After registering all event handlers, the virtual machine is started as usual. It will call the registered listeners whenever an event of interest occurs. The call includes light-weight event data, containing the exact type of event and a pointer to the run-time environment. The first part of the data is used to distinguish subclasses of events. For instance, read and write accesses may share the same event handler, but the handler may still need to know the exact nature of the access in one decision. The second part of event data serves to query the virtual machine about more information, such as the exact state of each thread. Some events include certain information which is certainly used in any case for efficiency. This allows to eliminate queries for basic information that is always needed when an event occurs (for instance, which lock was used in a lock release).

Event	Subclasses (if available)	Purpose/possible checks
Field access	Read/write access	Locking discipline (e.g. Eraser)
Lock event	Lock acq./release	Locking (e.g. deadlock detection)
Method event	Method start/end	Call graph construction
Thread creation	–	Recording thread name and type
Bytecode execution	Events for all 25 abstract instructions	Modelling of instruction-specific properties
Caught exception	–	Exception handler coverage
Program termination	–	Final report, cleaning up RV data

Table 8.1: *Run-time verification events in JNuke.*

Events are separated into different classes, as shown in Table 8.1. This allows RV algorithms to install generic event handlers to deal with common aspects of a super class of events, which then delegates fine points to particular subclasses. These event handlers may also have to perform *event re-ordering,* because one instruction may generate several events, which do not necessarily occur in the right order. For instance, entry to a `synchronized` method causes three events: the lock acquistion on method entry, method entry itself, and the first bytecode instruction of the method. Certain algorithms may require these events to occur in a certain order in order to work properly. For simplicity, the current API does not allow for specifying the order in which such simultaneous events are received. However, this minor problem, which does not occur often, can be easily solved in the listener implementation.

The RV API itself builds on low-level listeners provided by the VM. The low-level listeners are embedded in the reponsible module: field access events are treated by the heap manager while the lock manager deals with lock events. Other events are issued by the run-time environment itself, such as bytecode execution events. The RV API was created to provide a single front end to all these different event callbacks. It also allows to activate certain auxiliary listeners that log history information as the program executes. This is very useful for printing more detailed trace information. These two features, a simple front end and history information, greatly reduce the amount of work required for implementing a new run-time verification algorithm.

8.4 Instrumentation-based Implementation: JPaX

This section describes an alternative implementation approach for run-time verification, using the Java Path Explorer (JPaX) [HR01, GH03]. Initial experiments concerning high-level data races were carried out with JPaX [AHB03]. It consists of two parts: the instrumenter and the observer. The instrumenter produces an instrumented version of the program, which when executed, generates an event log with the information for the observer to determine the correctness of examined properties. Figure 8.7 illustrates the situation.

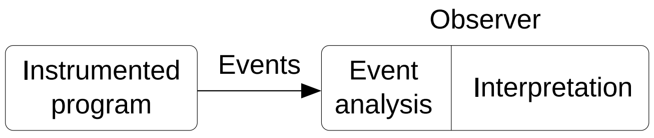


***Figure 8.7**: Instrumentation-based approach to run-time verification.*

The observation of events generated by the instrumented program is divided into two stages: event analysis and interpretation of events. The former reconstructs the context required for event interpretation. The latter contains the actual observation algorithms. The observer used here only checks for high-level data races. For these experiments, a new and yet totally un-optimized version of JPaX was used. It instruments every field access, regardless of whether it can be statically proven to be thread-safe. This is the reason why some data-intensive applications created log files which grew prohibitively large ($>$ 0.5 GB) and could not be analyzed.

8.4.1 Java Bytecode Instrumentation

Part of JPaX is a very general and powerful instrumentation package for instrumenting Java bytecode [GH03]. The requirements of the instrumentation package include power, flexibility, ease of use, portability, and efficiency. Alternative approaches were rejected, such as instrumenting Java source code, using the debugging interface, and modifying the Java Virtual Machine because they violated one or another of these requirements.

It is essential to minimize the impact of the instrumentation on program execution. This is especially the case for real-time applications, which may particularly benefit from this approach. Low-impact instrumentation may require careful trade-offs between the local computation of the instrumentation and the amount of data transmitted to the observer. The instrumentation package allows such trades to be made by allowing seamless insertion of Java code at any program point.

Code is instrumented based on an *instrument specification* consisting of a collection of predicate-action rules. A predicate is a filter on source code statements. These predicates are conjunctions of atomic predicates including predicates that distinguish statement types, presence of method invocations, pattern-matched references to fields and local variables, etc. Actions are specifications describing the inserted instrumentation code. Actions are inserted where predicates evaluate to true. Actions include reporting the program point (method, and source statement number), a time stamp, the executing thread, the statement type, the value of variables or an expression, and invocation of auxiliary methods. Values of primitive types are recorded in the event log. If the value is an object, a unique integer descriptor of the object is recorded.

The instrumentation has been implemented using Jtrek [Coh01], a Java API that provides lower-level instrumentation functionality. In general, use of bytecode instrumentation, and use of Jtrek in particular, has worked out well, but there are some remaining challenges with respect to instrumenting the concurrency aspects of program execution.

8.4.2 Event Stream Format

All operations in the instrumented application which write to the event log have to be as fast as possible. Among other factors, light-weight locking, incurring as little lock contention as possible, helps achieving this goal. When several pieces of information are logged by the instrumentation, they are therefore recorded separately, not atomically. As a result of this, one event can generate several log entries. Log entries of different threads may therefore be interleaved, as shown in Figure 8.8.

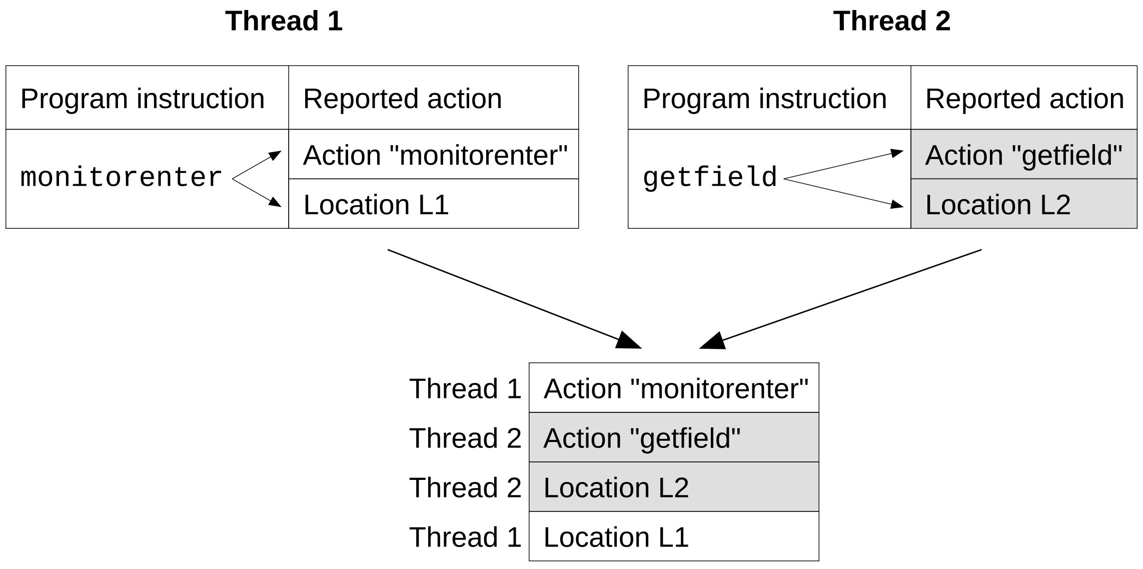


Figure 8.8: *Interleaving of light-weight event entries.*

The example shows two events that each result in two log entries: The action itself, and the location where it occurred. The events of each thread are streamed individually, which preserves the total order of events within a thread. In order to allow a faithful reconstruction of the events, each log entry includes the hash code of the active thread creating the log entry. Therefore the events can all be assigned to the original threads. The contextual information in events includes thread names, code locations, and reentrant acquisitions of locks (lock counts). The event analysis package maintains a database with the full context of the event log. It is not desirable that each single low-level event is visible to the event interpretation algorithm, which is described below. Because contextual data for an event is only complete after the event location is updated, an intermediate layer maintains run-time context data and only relays events when that data with is consistent respect to the current thread.

8.4.3 Observer Architecture

As described above, run-time analysis is divided into two parts: instrumenting and running the instrumented program, which produces a series of events, and observing these events. The second part, event observation, can be split into two stages: event analysis, which reads the events and reconstructs the run-time context, and event interpretation (see Figure 8.9). Note that there may be many event interpreters.

Reusing the context reconstruction module allows for writing simpler event interpreters, which can subscribe to particular event types made accessible through an observer interface [GHJV95] and which are completely decoupled from each other.

Each event interpreter builds its own model of the event trace, which may consist of dependency graphs or other data structures. It is up to the event interpreter to record all relevant information for keeping a history of events, since the context maintained by the event analysis changes dynamically with event evaluation. Any information that needs to be kept for the final output, in addition to context information, needs to be stored by the event interpreter. If an analysis detects violations of its rules in the model, it can then show the results using stored data.

Besides clearly separating two aspects of event evaluation, this approach has other advantages: Many algorithms dealing with multi-threading problems require similar information, namely lock and field accesses. If a log generated by an instrumented program includes this information, then several analysis algorithms can share the same events. Furthermore, splitting event observation into two steps also allows writing an event analysis front-end for event logs generated by tools other than JPaX, reusing the back-end, event interpretation.

8.5 Module Overview

The remainder of this chapter focuses again on JNuke and describes architectural and implementation-specific design decisions taken. It uses the following notational convention: Names written in `typewriter` font represent variable names, file names, or commands. Names written in *`italicized typewriter`* font denote placeholders that have to be replaced with actual names.

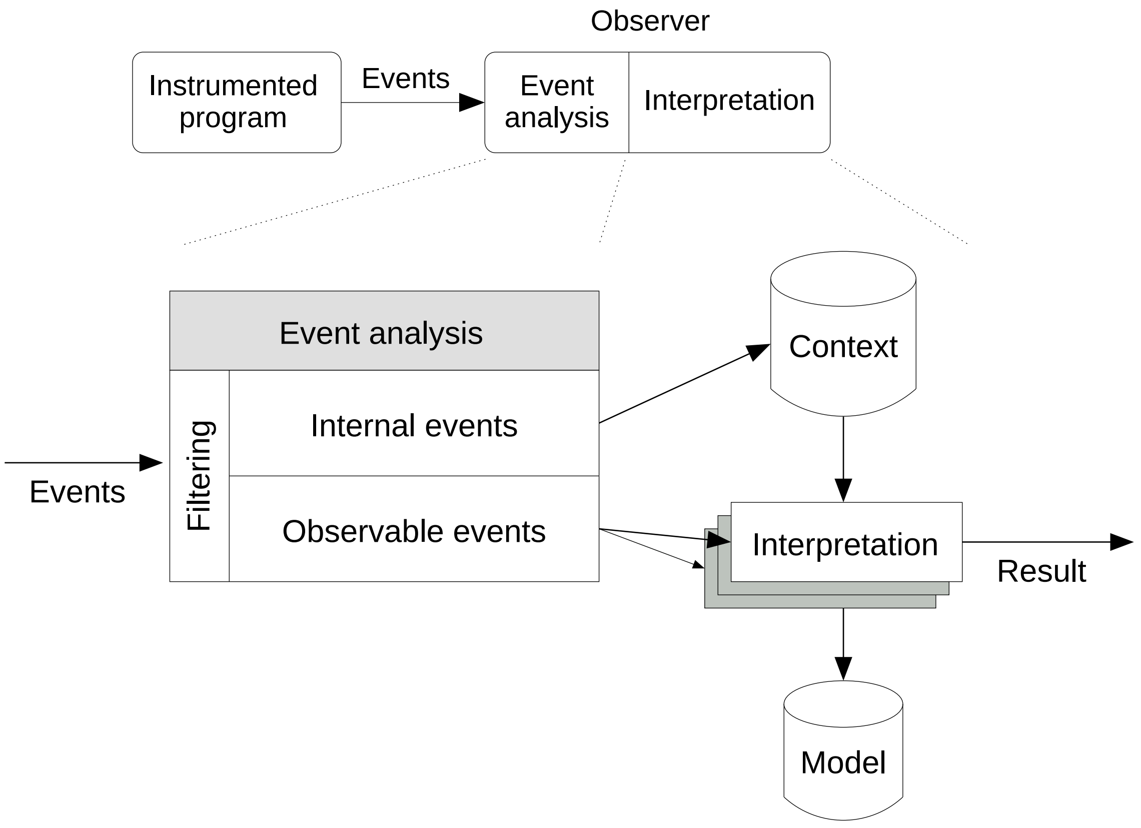


***Figure 8.9**: Detailed view of instrumentation-based event observers.*

8.5.1 Description

Table 8.10 gives a brief overview of each module. Each module resides in its own directory and contains at least one header file, *m*/*m*.h, declaring data types and interfaces. Most modules use additional header files, which are needed when using functionality that it outside the core scope of a module, or used when certain code is generated automatically.

Module	Purpose
`algo`	Generic software analysis algorithms
`cnt`	Container classes
`jar`	Reading and writing of `jar` files
`java`	Classes representing Java data; Java class loader
`rv`	Run-time verification algorithms
`sa`	Static analysis algorithms
`sys`	Low-level classes encapsulating system dependencies
`test`	Test driver
`vm`	JNuke Virtual Machine

***Table 8.2**: Short description of each module.*

8.5.2 Module Dependencies

A module corresponds to a directory in the file layout. A module (or package) is declared in its header file *m*/*m*.h. Dependencies arise if the implementation (any C file) includes other header files (such as `test.h`).

Figure 8.10 shows dependencies between JNuke packages. Each node represents a module; an arrow between nodes represents a module being dependent on the one the arrow points to. The modules at the bottom contain the fewest dependencies.

Each module is dependant on both `sys` and `test`, which have been omitted for simplicity. The goal was to calculate only "true" dependencies arising from the implementation. When computing module dependencies (using scripts `util/moddep.sh` and `util/moddep.pl`), test code has been omitted since some white-box tests may access implementation-specific details.

Even then, the graph is somewhat cluttered. Figure 8.11 shows dependencies of modules which are on a higher level than `test`, `sys`, `cnt`, and `pp`. It shows clearly how `algo`, the module for generic analysis algorithms, is independent of run-time verification (`rv`) and static analysis (`sa`). Like the virtual machine (`vm`), it uses module `java` in order to handle class file data and class loading.

8.6 JNuke's OO in C

In order to avoid some problems with C++ and to get a better performance than in Java, it was decided to create our a custom, simple OO extension for C. It lacks some "syntactic

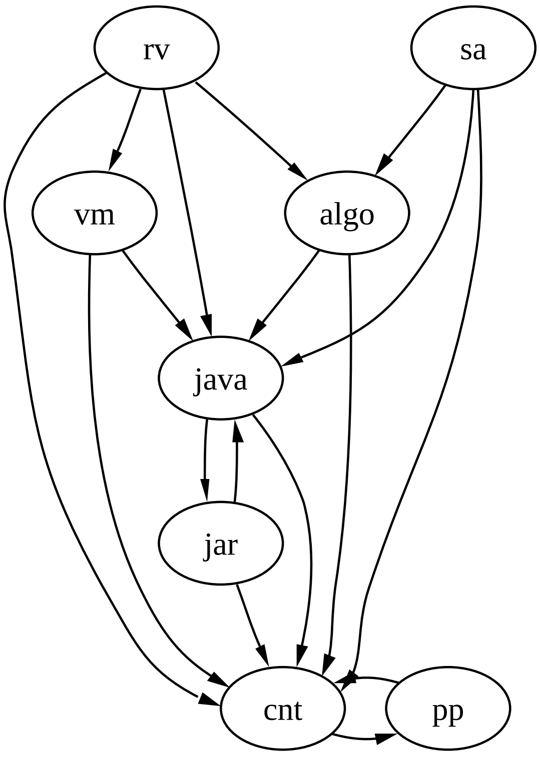


Figure 8.10: *Module dependencies.*

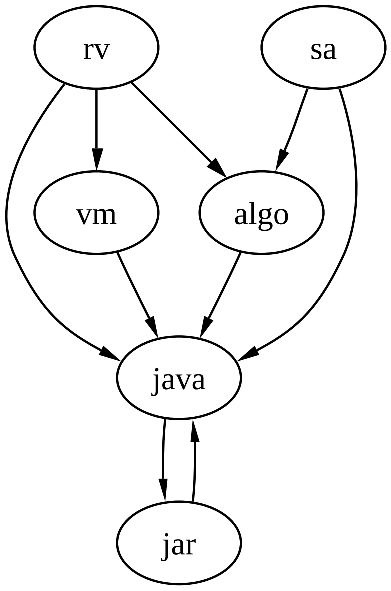


Figure 8.11: *Dependencies of higher-level modules.*

sugar" of "true" OO languages but does not need a special preprocessor. This section describes this OO extension.

Beyond performance reasons the C programming language [KR88] was chosen because it is far easier to master and more portable than C++, and more stable than the Java language which is still evolving [GJSB00]. Furthermore, common C compilers are about an order of magnitude faster than their C++ and Java counterparts, which leads to a much faster edit → compile → debug cycle than with other programming languages.

8.6.1 Memory Management

For a better separation of control of memory management and potential future use of multi-threading within JNuke itself, a special variable `JNukeMem *` is always used when allocating memory. The standard memory allocation functions have been superseded counterparts as shown in Table 8.3. The common difference to their standard counterparts from `stdlib.h` is that they all require an explicit pointer to the heap that the thread is using (`JNukeMem *`).

Original function	Replacement in JNuke
void * `malloc (`**int**`)`	**void** * `JNuke_malloc (JNukeMem *, `**int**`)`
void * `realloc (`**void** *`, `**int**`)`	**void** * `JNuke_realloc (JNukeMem *,` **void** *`, `**int**`, `**int**`)`
void `free (`**void** *`)`	**void** `JNuke_free (JNukeMem *, `**void** *`, `**int**`)`

Table 8.3: Replacement (wrapper) functions for memory management.

Furthermore, the functions `JNuke_realloc` and `JNuke_free` expect as an extra argument the *size* of the currently allocated block at the address the pointer refers to. When compiling JNuke with the standard options, the size of each allocated memory block is stored internally and validated when it is reallocated or freed. When compiling optimized code, that validation is turned off, resulting in a performance improvement over the standard memory allocation library, which does not include the size as a parameter for `free` but maintains that data on its own.

8.6.2 Comparison to Java

The OO model in JNuke is semantically very similar to the one used in Java or C++, but there are a few differences and restrictions. Figure 8.12 shows a comparison between Java's and JNuke's OO model.

1. In JNuke, any object is generically declared as `JNukeObj *`. There is no possibility to explicitly declare a (statically type safe) object of a specific type.

```
              Java                                 JNuke (in C)
(1) Integer Num;                       (1) JNukeObj *Num;
    int value;                             int value;
(2) Num = new Integer(8);              (2) Num = JNukeInt_new (mem);
                                           JNukeInt_set (Num, 8);
(3) value = Num.getInteger();          (3) value = JNukeInt_value (Num);
                                       (4) JNukeObj_delete (Num);
```

***Figure 8.12**: JNuke's OO model.*

2. The two constructors are very similar. JNuke lacks some syntactic sugar, so the `new` operand is written as a function, which is given by the type with `_new` appended to it. Each constructor explicitly requires the memory manager as its single argument. In C, it is not possible to have several functions with the same name, but different signatures. Therefore, optional arguments in constructors are not possible. This choice was taken to make serialization and deserialization simpler [Bau02].

3. Method calls to most methods work by appending _*`methodname`* to the class name (which begins with JNuke to ensure uniqueness). The object instance itself (`this` in Java) is always written explicitly as the first argument of the method. Static methods do not need that argument. Unlike in Java, there is no syntactic sugar hiding the type (which may not be known statically in Java) and the `this` parameter.

4. Unlike in Java but like in C++, a destructor is required for each class. This method is *polymorphic* and is wrapped with `JNukeObj_delete`. That function resolves the dynamic type and calls the appropriate destructor, which has been set in the type information of the object instance in the constructor.

8.6.3 Type `JNukeObj`

Each object has associated with it a *type* information, its memory manager, and the (private) instance data; see Figure 8.13. The latter is a generic pointer which has to be cast to a (private) struct type which contains all the fields of the object.

```
struct JNukeObj
{
  JNukeType *type;      /* type information */
  JNukeMem *mem;        /* memory manager */
  void *obj;            /* object instance */
};
```

***Figure 8.13**: The C `struct JNukeObj`.*

The callee has to retrieve and convert the `this->obj` pointer to access the data structure. A macro `JNuke_cast (Type, ref)` does this and also performs a type check on the data used. An example of how the private data is "unpacked" from the struct is given in Figure 8.14, where the class JNukePair returns the second field of the pair. First, the pointer to the private data is re-cast to the record containing the data (JNukePair). Then, the field of JNukePair (in this case, first) can be accessed by normal C struct access. In very rare cases, where type checking needs to be disabled for the cast of this->obj, the macro JNuke_fCast can be used. Such cases include access to subclasses, where the simple JNuke_cast macro fails because it can only cast an instance to a single type.

```
JNukeObj *
JNukePair_first (const JNukeObj * this)
{
  JNukePair *pair;
  assert (this);
  pair = JNuke_cast (Pair, this);
  return pair->first;
}
```

***Figure 8.14**: Retrieving an object's instance data.*

Each type in JNuke contains a fixed set of information: the type itself, stored as a string constant, and a set of function pointers to polymorphic functions (see Section 8.6.5). Note that type comparison works by comparing the addresses of statically allocated type structs, not type name strings. It is therefore very fast. Figure 8.15 shows the declaration taken from sys/sys.h. Further extensions via subtypes are supported by an extra pointer and described below.

Type comparison

Since every type has a unique string with the type name associated to it, types can be compared quickly using the pointer values of this type name string. The operation JNukeObj_isType encapsulates this for all objects. It returns 1 if types match:

```
if (JNukeObj_isType (ref, UCSStringType))
  /* object "ref" is of type UCSString */
```

This function is hardly ever used, though, because the JNuke_cast macro (see above) itself checks for the correct type, so a manual check is hardly ever needed. For type checking purposes, all object types must be public and declared in the corresponding header files. The convention for type declarations is to include them before listing the object methods:

```
extern JNukeType JNukeThreadType;
JNukeObj *JNukeThread_new (JNukeMem * mem);
```

```
struct JNukeType
{
  /* name of object type as a string */
  const char *name;

  /* returns a deep copy of the instance */
  JNukeObj *(*clone) (const JNukeObj *);

  /* deletes instance of object */
  void (*delete) (JNukeObj *);

  /* returns comparison result as in strcmp(3); the result
   * only needs to be defined for objects of same type;
   * otherwise, pointer value is used (no true polymorphism)
   */
  int (*cmp) (const JNukeObj *, const JNukeObj *);

  /* returns hash code of object */
  int (*hash) (const JNukeObj *);

  /* returns new string with string representation of the
   * current instance */
  char *(*toString) (const JNukeObj *);

  /* clears object data (only used for containers: calls
   * delete for each element. "Deep" (recursive) clears
   * have to be done manually, but occur rarely in practice.
   */
  void (*clear) (JNukeObj *);

  /* allow for subtyping */
  void *subtype;
};
```

Figure 8.15: *The type information of each object.*

8.6.4 Statically Typed Method Calls

Statically typed method calls have been shown in Figure 8.12. To reiterate, the lack of preprocessing means that the entire "ugly" syntax with all parameters is required, such as in `JNukeInt_set (Number, 8)`. Modern object-oriented languages usually hide the static type and use the first parameter as the invocation target. In Java-like syntax, the above call would be written `Number.set(8)`. In JNuke, each type is prefixed with `JNuke`, and the function name which uniquely identifies the statically known method call consists of this prefixed type, an underscore, and the method name.

8.6.5 Polymorphism

Functions which are part of `struct JNukeType` (see Figure 8.15) can be called polymorphically. For example, calling `toString` on a container will result in the recursive pretty-printing of all the data that the container holds at the moment. Containers may be nested like in other OO languages.

8.6.6 Inheritance

Inheritance in the first level is achieved through pointer `subtype` in struct `JNukeType`. This step is necessary so the "primitive" type is always available for direct comparison in `JNuke_cast`. For nested subtyping, this extra indirection is not necessary. Therefore the struct of the super type can be used directly (without a pointer) in nested subtypes. This leads to nested C structs, as shown in Figure 8.16.

8.6.7 Constructor and Destructor

Each class has to declare its type statically. This is done by a global struct `JNukeType` in each class, which contains the given values for all the fields. The first part of Figure 8.17 shows this static variable. Note that the C keyword `static` refers to the visibility of the global variable, which is only visible in the current source file. Static memory allocation is in fact determined by the compiler and not expressed by the keyword `static`. Furthermore, the fields are *not* set as constants by the C compiler, but considered to be read-only as a convention.

The duties of the constructor entail:

- Allocation of the `JNukeObj` record itself.
- Setting the fields of the `JNukeObj` record. This includes setting type information, private data, and the memory manager, since the certain operations such as `delete` have to use the same memory manager.[2] In order not to have to create new type information for each object instance, the constructor simply assigns this pre-initialized common static type to the type information of each instance, thus sharing type information.

[2]The equality of memory managers may be a non-trivial requirement for certain interactions between objects. Such a case is given in `java/bctrans.c`.

```
/* in header file */
struct JNukeSubType JNukeSubType;

struct JNukeSubType
{
  ...  (*method1);
  ...  (*method2);
};

struct JNukeSubSubType
{
  JNukeSubType super;
  /* inheritance of super class methods */
  ...  (*method3);
};

/* in C file, implementation */
/* sub class of JNukeType */
static JNukeSubType JNukeSubImplementation = {
  method1,
  method2
};

static JNukeSubSubType JNukeSubSubImplementation = {
  { &JNukeSubImplementation },
  method3
};
/* note the nested curly braces */
```

***Figure 8.16**: Nested inheritance.*

```
/*------------------------------------------------------*/
/* type declaration */
/*------------------------------------------------------*/

static JNukeType JNukePairType = {
  "JNukePair",
  JNukePair_clone,
  JNukePair_delete,
  JNukePair_compare,
  JNukePair_hash,
  JNukePair_toString,
  NULL
};

/*------------------------------------------------------*/
/* constructor */
/*------------------------------------------------------*/
JNukeObj *
JNukePair_new (JNukeMem * mem)
{
  JNukePair *pair;
  JNukeObj *result;

  assert (mem);

  result = JNuke_malloc (mem, sizeof (JNukeObj));
  result->mem = mem;
  result->type = &JNukePairType;
  pair = JNuke_malloc (mem, sizeof (JNukePair));
  result->obj = pair;
  pair->type = JNukeContentObj;
  pair->isMulti = 0;
  pair->first = NULL;
  pair->second = NULL;

  return result;
}
```

Figure 8.17: *Constructor of the type `JNukePair`.*

- Initialization of private data. This always includes allocating the record which holds the private data. In some cases, reasonable default values are set for data, although the programmer should not rely on that behavior for non-container objects holding data.

The destructor has to free all the memory that was allocated in the constructor.

8.6.8 Containers

JNuke includes some container classes, which store a number of other objects (including containers), pointers or integers. Note that the three types of data cannot be mixed, and only objects can contain other objects recursively.

8.6.8.1 Data Types

The three data types are declared in type `enum JNukeContent` in `cnt/cnt.h`. Each container defaults to `JNukeContentObj`, storing objects. For efficiency, most containers also support direct storage of integers or pointers, through method `JNuke`*`Type`*`_setType (JNukeObj *, JNukeContent)`. In that case, the content of that data is never accessed, and recursive operations are not possible.

8.6.8.2 Reference Semantics

With the exception of the object pool `JNukePool`, most containers have a reference semantics for most operations. While `clone` will perform deep cloning for object data, `delete` will not recursively delete the content in order to allow sharing of objects. Therefore, the content of a container has to be deleted with `JNuke`*`Type`*`_clear`. In containers supporting deep cloning, cloning of each data object can be suppressed by temporarily changing the container data type to `JNukeContentPtr` for the `clone` operation.

8.6.8.3 The **`JNukePool`** Object Pool

Class `JNukePool` implements a special container. Its purpose is to implement a separate memory management pool that automatically destroys all objects it contains upon deletion. This is achieved by inserting *clones* of objects when `insert` is called. The pool afterwards only operates on the clone and destroys the clone upon deletion, not the original. The pool keeps a reference count of each object used, cloning the inserted object only when it is inserted for the first time. The test whether a new copy of the object should be inserted is based on the equality function `JNukeObj_cmp`.

Sometimes it is known when `insert` is called that the original instance should be deleted in any case. In such a scenario, the method `insertThis` can be used. That method inserts the reference itself, not a clone of the instance the reference points to. It returns the resulting valid reference in the object pool. If the same object already exists, then the given argument is deleted and the pool reference is returned. This ensures consistency.

The caller must therefore always use the return value of the `insertThis` call instead of the reference passed to it after the call! A typical call looks as follows:

```
instance = JNukePool_insertThis (pool, instance);
/* updates instance such that reference from pool
 * is used; return value may be discarded iff
   pointer "instance" is not used again */
```

8.6.8.4 Iterators

JNuke contains two types of iterators: a read-only iterator and a read-write iterator. Both are semantically similar to C++ STL or Java iterators. Only the read-only iterator, `JNukeIterator`, is described in detail. The second iterator is called `JNukeRWIterator`, and its usage is very similar. Both types are polymorphic and declared in header file `cnt/iterator.h`.

The read-only iterator can be initialized from any container object. Three operations are used for read-only iterators:

1. Initializing the iterator. Each type has a method `JNuke`*`Type`*`Iterator` to initialize an iterator. Since an iterator is usually stack-allocated, these methods do *not* allocate memory for it.

2. Checking for completion: Method `JNuke_done` returns true when there are no more elements accessible by the iterator in that container.

3. Accessing the next field. For the read-only iterator, this operation will automatically advance the iterator as well. This updates the iterator state and makes the iterator point to the *next* element, although the *current* element is returned. Therefore no other operation is needed before checking again for termination of the iterator.

```
    JNukeIterator it;
    JNukeObj *data;
(1) it = JNukeVectorIterator (vector);
(2) while (!JNuke_done (&it))
      {
(3)     data = JNuke_next (&it); /* STL: *(it++) */
        /* do something */
      }
```

Figure 8.18: *An example for using a read-only iterator.*

The read-write iterator works similarly, with two differences:

1. All operation names are capitalized, i. e., iterator methods are called `JNuke_Done`, `JNuke_Next`, etc.

2. `JNuke_Next` moves the iterator forward, but does not return the element; this is done with `JNuke_Get`, which does not affect the position of the iterator. An element the iterator points to can be deleted with `JNuke_Remove` after processing.

`JNuke_Remove` allows the deletion of elements, if supported by the container used. Note that this will set the iterator back to the previous element (or a sentinel at the beginning of the iteration), so `JNuke_Next` has to be called to obtain a valid iterator again. (Therefore `JNuke_Next` has to be called regardless of whether an element was removed or not.)

8.7 Unit tests

Unit tests are the core of the quality assurance effort which tries to keep JNuke as reliable as possible while making it easy to change its design or implementation (by refactoring). As shown by statistics below, work on JNuke was not continuous due to varying student participation and work at NASA during summer 2002 and 2003; overall, about three man-years of work went into producing JNuke and its unit tests.

8.7.1 Structure

The unit tests are organized in several *test suites.* Each package has its own test suite. Test suites are divided into *test groups,* where there is normally one group for each class. Within a class, several tests can be registered for each group.

Each class (each C source file) has a section

```
#ifdef JNUKE_TEST
/* Unit tests */
...
#endif
```

which contains all the test code of a class. For some classes, the test code is longer than the actual implementation itself.

Each unit test is a function of the format `JNuke_`*`pkg_class_num`*. Figure 8.19 shows the code of a simple unit test which creates a `JNuke_Int` object and deletes it again, checking the class against simple memory leaks. Each test case is a function taking a single `JNukeTestEnv *` argument and returning an integer. One of the members of the `JNukeTestEnv` struct is the pointer to the memory manager, `env->mem`. The return value is non-zero for a successful test and zero for a failed one.

Typically, a test starts by initializing an integer `res` to `1` or a different value that indicates success or failure. Then, the tests are run, where the result variable `res` is updated each time a condition is checked. Note that the idiom `res = res &&` *`cond`* ensures that the variable is set to `0` upon failure. If it is not possible to express *`cond`* in a single line, a construct like the one in Figure 8.20 should be used. This ensures that there are no statements which are skipped when the test case is successful and makes it possible to achieve 100 % code coverage for a successful test. Code like `if (!cond) res = 0;` will have the second statement uncovered in case of success, and will result in a statement coverage lower than 100 % if the second statement is not on the same line.

```
int
JNuke_sys_JNukeInt_0 (JNukeTestEnv * env)
{
  /* creation and deletion */
  JNukeObj *Int;
  int res;

  Int = JNukeInt_new (env->mem);
  JNukeInt_set (Int, 1 << 31);
  res = (Int != NULL);
  res = res && (JNukeInt_value (Int) == 1 << 31);
  if (Int != NULL)
    JNukeObj_delete (Int);

  return res;
}
```

Figure 8.19: *A simple test case.*

```
if (res)
  {
    res = 0;
    if (cond)
      {
        /* do something conditionally */
        ...
        /* perhaps more checks */
        res = 1;
      }
  }
```

Figure 8.20: *A construct ensuring maximal coverage for successful test cases.*

8.7.2 Registering Test Cases

Each test case has to be registered for the test driver. This is done in special files in each package, usually `black.c`, sometimes `white.c`. Inside each such file, macro `SUITE` registers the entire test suite for that package. The developer usually only has to register new classes as a new test group with macro `GROUP("`*`string`*`")` and add new test cases for a class with macros `FAST` and `SLOW`. Both macros take the same three arguments: package name, class name, and test case number. By convention, fast test cases do not take more than 0.01 s to run on an unloaded "fast" workstation (such as the ones in the research labs). This ensures that a large subset of all unit tests can be run quickly even when extra checking tools, such as valgrind [NS03], are active.

Note that the package and class names are not quoted for the `FAST` and `SLOW` macros but only for the `GROUP` and `SUITE` macros.

8.7.3 `JNukeTestEnv`

Figure 8.21 shows the entire environment which is available to test cases, taken from `test/test.h`. It should be noted that tests which write to the standard error channel (`stderr`) will have their output automatically redirected to the error output file when in test mode.

```
struct JNukeTestEnv
{
  FILE *in, *log, *err;
  /* input stream, log stream, error output stream */
  int inSize;             /* size of input stream */
  const char *inDir;
  /* input directory where tests read or write files */
  JNukeMem *mem;          /* memory manager */
};
```

Figure 8.21: *The struct* `JNukeTestEnv`.

8.7.4 Log Files

The success of a test case is determined by its return value. In addition to that, the output of the standard error channel and log files written can also be used to determine correctness of a unit test. In this case, if a given output file (template) exists, the output of a test will be compared to this file after execution. If the outputs do not match or the current test produced no output where an output was expected, the test fails.

The log files are stored in directory `log/`*`pkg`*`/`*`class`*. Each test case can write to `env->log`, which is directed to log file *`num`*`.log` in the directory of that class. A template log file is stored under *`num`*`.out`. The standard error output is directed into *`num`*`.err`, and its template is in *`num`*`.eout`, respectively.

8.7.5 Code Coverage

The aim of the testing process was to achive 100 % code (statement) coverage, in order to find most errors early and to avoid dead code. This goal was almost achieved, with over 99.9 % code coverage in general and 100 % coverage for those modules that are considered finished. The only exception is the instrumentation package, where an inefficient loop in the code was never fixed. That loop lead to poor performance in pathological cases.

Figure 8.22 shows how the project has grown over the nearly four years so far, plotting the overall size of the code against the number of test cases, where about 1800 test cases were used at the end of the project. While unit testing was part of the project from its very start, code coverage was initially not measured. Later on, the goal was to have full code coverage in order to avoid "surprises" later on in development, when previously unused code would be executed for the first time, usually producing a failure. Locating such faults can be very costly in older code, which may have been developed by people who no longer participate at the project. Therefore we strongly believe that the effort of preventing such failures pays off in the long term. Figures 8.23 and 8.24 show that only after a bit more than a year, towards late 2002, a serious attempt was made to achieve (and hold) the goal of full code coverage. Continued development afterwards introduced a few lines in some files that had not yet been fully tested. Sometimes successes in quality assurance were quickly thrown back by this. Figure 8.25 illustrates this fact. Nonetheless, towards the end of development of a certain module, care was taken to finish that project with a suite of test cases that achieves full code coverage.

8.8 Summary

This chapter described the implementation of dynamic part of JNuke, its VM and RV API. It covered the principle of the observer architecture and its application in the context of run-time verification. As an alternative approach, instrumentation-based analysis using JPaX was outlined. Technical aspects included the organization of JNuke into modules, an object-oriented layer in C, and unit tests.

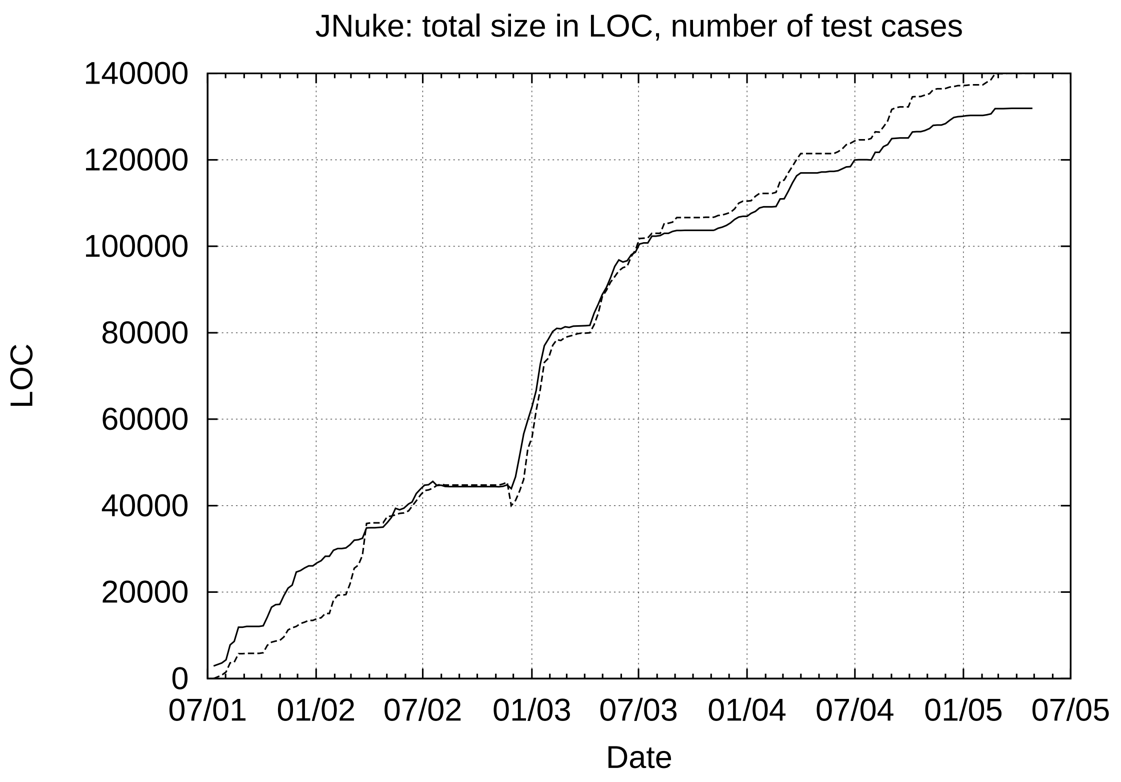


Figure 8.22: *Project size in lines of code (solid) and number of test cases (dashed).*

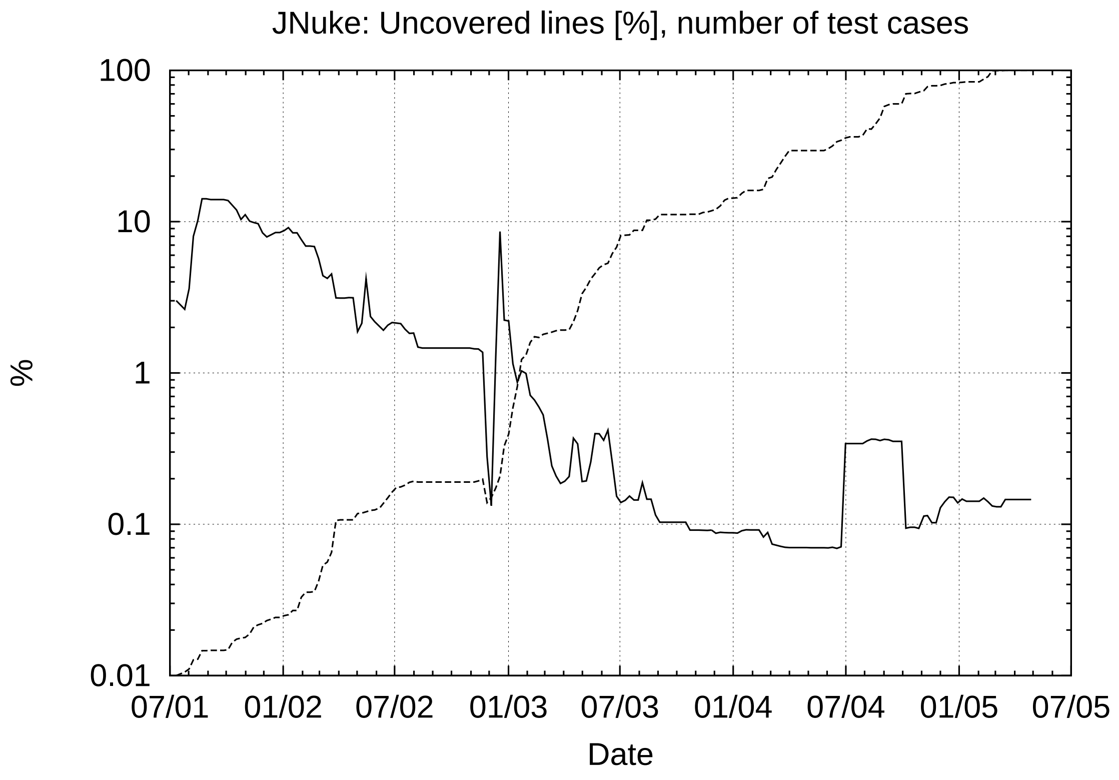


***Figure 8.23**: Uncovered code in percent (solid) and number of test cases (dashed).*

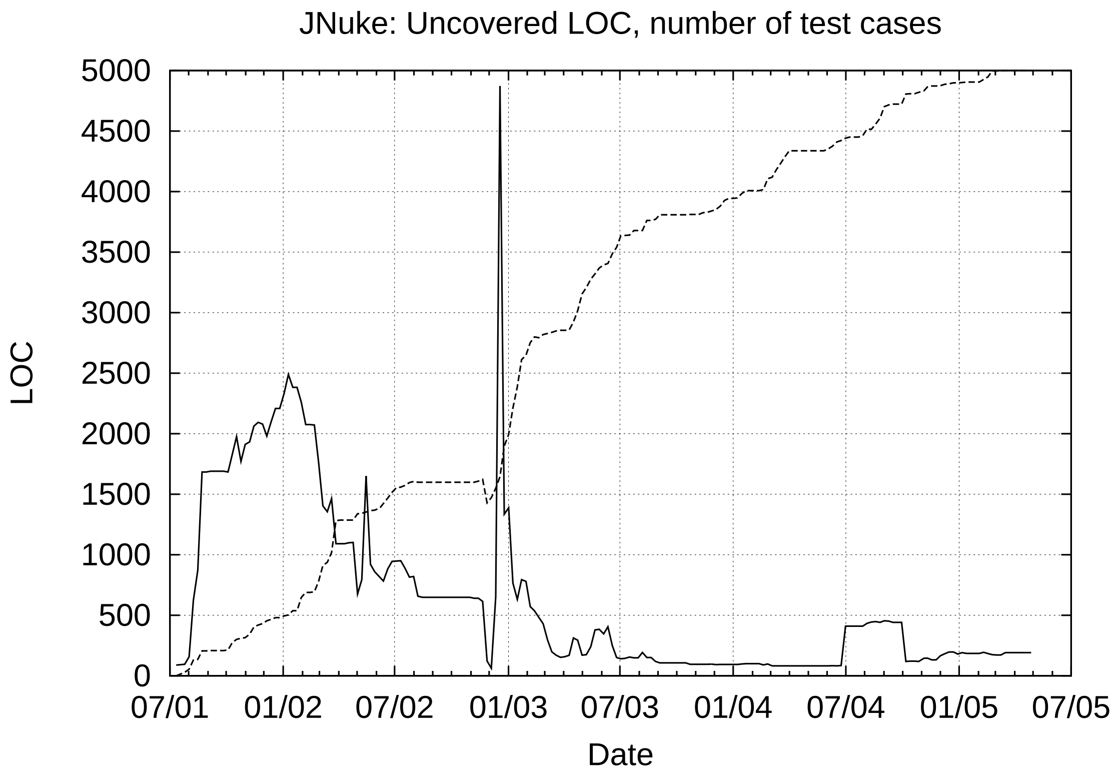


***Figure 8.24**: Uncovered lines of code (solid) and number of test cases (dashed).*

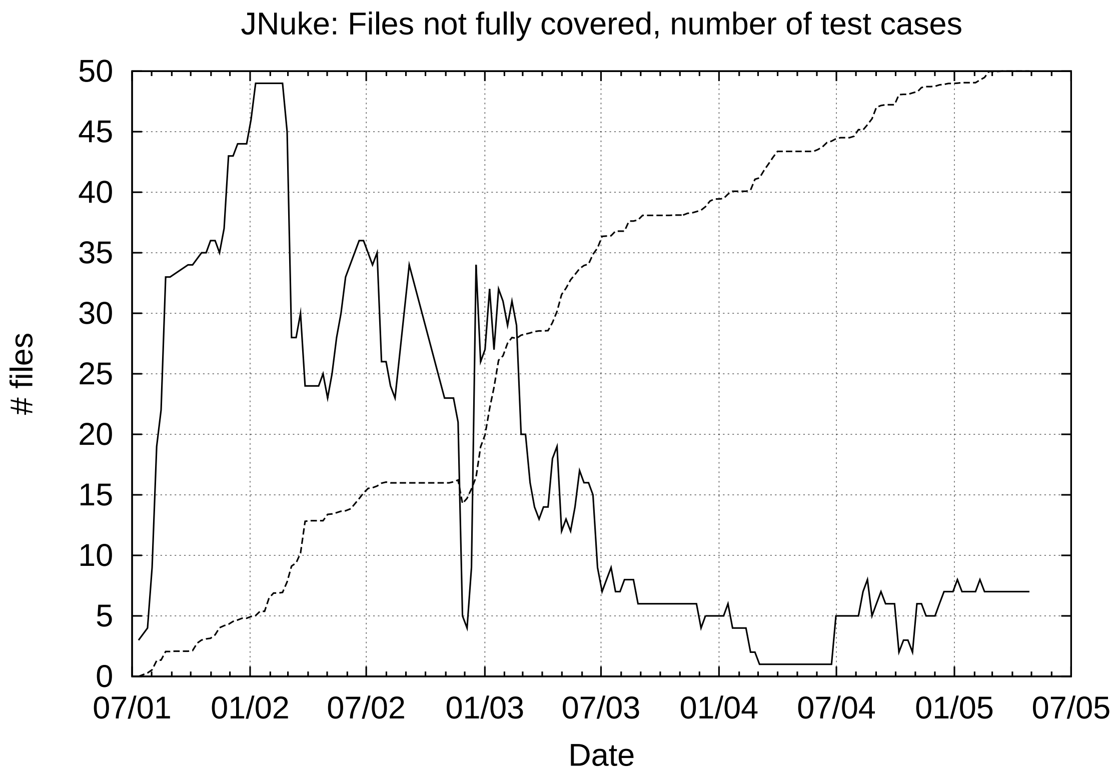


Figure 8.25: *Number of files containing uncovered lines (solid) and number of test cases (dashed).*

9 Experiments

This chapter describes experiments carried out with JNuke and JPaX using the algorithms presented in this thesis. First, an overview is given of all benchmark applications which were used to evaluate the effectiveness of the fault patterns described in this thesis. Subsequent Sections illustrate experiments carried out. In all experiments, multiple warnings for the same field or set of fields were counted as a single warning, because they all refer to the exact same artifacts in source code. Such duplicates can easily filtered out, the process of which is outlined in Chapter 11. An *overhead* given in percent denotes the extra run time (or memory) required when compared to the base setting. Therefore an overhead of 0 % corresponds to the same resource usage as in the base case, while 100 % corresponds to twice the original number.

9.1 Applications

Table 9.1 summarizes benchmark applications used to evaluate the different tools. These applications will be referred to throughout the entire chapter. Some other applications or variants thereof were used for parts of the experiments only. (For instance, single-threaded, serial applications were only used as part of the performance evaluation of the JNuke VM.)

Certain applications are part of a multi-threaded benchmark suite that has recently been initiated [EU04]. Several multi-threaded applications were included in the experiments listed here, which contain seeded concurrency faults. They are described under Bank, Deadlock, Lottery, Shop, and TicketOrder.

The Daisy application implements a very simple multi-threaded file system [FQ04]. It only includes a minimalistic test suite, so its unit tests do not execute enough code to find many faults in the system. Furthermore, it implements its own locking scheme on top of Java locks; in order to detect lock problems on that higher abstraction layer, rules of checking tools have to be adjusted accordingly. This was not done for JNuke. The Dining Philosopher application [Eug03] is a correct implementation of a well-known problem. The implementation avoids a cyclic deadlock by having a different lock acquisition order for one of the threads representing a philosopher. The hedc tool [vPG01] accesses astrophysics data from web sources and models the processes of a Data Mining tool. Our line count drastically differs from the one given in other publications because unused concurrency library classes [Lea99] and given copies of standard Java library

Benchmark	Size [LOC]	Description
Bank [EU04]	150	Bank w. frequent transfers but no synchr.
Daisy [FQ04]	1,900	Multi-threaded (simulated) file system
Deadlock [EU04]	250	Worker thread sim. w. deadlock detection
DiningPhilo [Eug03]	100	Dining Philosophers (3 threads, 5,000 it.)
hedc [vPG01]	8,900	Data Mining tool
JGFCrypt [BSW$^+$99]	1,700	Large cryptography benchmark
JGFMontecarlo [BSW$^+$99]	3,600	Montecarlo simulation
JGFSparseMult [BSW$^+$99]	300	Sparse matrix multip. (50000×50000)
Lottery [EU04]	400	Assign a random number to each user
mtrt [Cor98]	11,300	Multi-threaded ray-tracing program
ProdCons [Eug03]	100	Producer/Consumer sim. (12,000 it.)
Santa [Tro94]	300	Santa Claus problem
Shop [EU04]	300	Shop sim. with customers and suppliers
SOR [vPG01]	250	Successive Over-Relaxation (2D grid)
TicketOrder [EU04]	200	Flight ticket order simulation
TSP [vPG01]	700	Travelling Salesman Problem
Web server [Sun97]	500	Simple multi-threaded HTTP server

***Table 9.1**: Benchmark programs.*

classes were not included in the line count. The Java Grande Forum (JGF) benchmarks includes various multi-threaded benchmarks modelling cryptographic and scientific calculations [BSW$^+$99]. The mtrt program [Cor98], a ray-tracer, also performs many calculations using multiple threads. What makes it relatively large is the large number of classes used for the test harness and the use of its own wrapper classes for input/output. The Producer/Consumer program [Eug03] simulates several threads using a common queue. The Santa Claus program [Tro94] was implemented as a part of the unit tests for JNuke's garbage collector [Far04]. SOR [vPG01] implements a multi-threaded, iterative algorithm on a 250×250 floating point matrix, using barriers [Lea99] to coordinate work between threads. The TSP program [vPG01] features a main thread, which reads map data from a file and then spawns a given number of worker threads to solve the problem exactly (which of course is only feasible for small maps). Finally, the Web server [Sun97] is a functional web server supporting HTML documents and images, written as a multi-threaded Java application in order to demonstrate different concurrency mechanisms in Java. It requires client requests and therefore was only used for testing correctness of the implementation, but not for measuring performance.

9.2 JNuke VM

The goal of these experiments was a comparison between JNuke's VM and the one publically available by Sun, which is highly optimized [Sun04b]. Most benchmarks were

Benchmark	Description	Sun's VM	JNuke's VM	
		Time [s]	Time [s]	Mem. [MB]
Array (Test3)	10,000,000 array accesses	0.50	96.04	38.3
Bank	Bank with frequent transactions	1.11	2.29	1.7
BubbleSort	10,000 array elements	0.57	176.64	0.5
Daisy	Simple file systems	0.24	6.40	16.9
Deadlock	Worker thread simulation	0.15	0.17	1.6
Dining Phil.	3 philosophers, 3,000 iterations	0.68	52.68	51.5
DoubleOp (Test8)	10,000,000 double operations	7.78	33.20	0.1
IOBenchmark1	100 threads writing 5 KB each	0.15	1.51	3.0
IOBenchmark2	100 threads reading 50 KB each	0.17	0.28	20.4
IOBenchmark3	1 thread writing 0.5 MB, 100 it.	0.25	1.95	2.5
IOBenchmark4	1 thread reading 0.5 MB, 100 it.	0.22	1.41	2.5
Iteration (Test)	Empty loop with 10 million iter.	0.25	42.05	0.1
Jaspa (MccaJaspa)	Sparse matrix multiplication	0.31	37.74	12.6
JGFCrypt (20 thr.)	IDEA encryption algorithm	1.93	177.09	35.7
JGFSeries (20 thr.)	Fourier coefficient series	22.49	566.72	1.4
JGFSparseMatmult (20 threads)	Sparse matrix multiplication	1.38	461.12	9.9
Lottery	Random number assignment	0.21	1.20	3.4
MethodInvocation (Test7)	4,000 invocations of 400 methods	0.42	10.57	2.0
MultiArray (Test5)	Six-dim. array, 10^6 entries	1.23	31.09	84.9
Producer/Consumer	120,000 it., 1 + 1 threads	3.31	28.04	32.1
Producer/Cons.[a]	120,000 it., 1 + 100 threads	226.87	47.70	25.7
ReadManyFields	200 iter. w. 5,000 field accesses	0.38	24.46	10.6
Santa[b]	Santa Claus problem	6.15	0.20	1.3
Shop	Shop simulation	0.21	0.12	1.6
SOR	1 iteration, 3 threads	0.16	1.64	2.7
TicketOrder	Flight ticket order simulation	0.13	5.80	1.8
TSP	Map size 10, 3 threads	0.17	0.95	3.2
TSP	Map size 15, 3 threads	0.17	12.22	3.2
Geometric mean of all execution times		0.66	8.72	

[a]The overhead is primarily caused by an inefficient pThreads implementation in the 2.4 Linux kernel, scaling poorly to 100 threads as used in this benchmark [Eug03].

[b]The JNuke VM skipped idle periods caused by all threads sleeping.

***Table 9.2**: Benchmarks to evaluate the performance of the VM. If a benchmark name in brackets is given, it refers to the Java program in the JNuke test suite.*

taken from the original implementation used to evaluate the initial version of the JNuke VM [Eug03]. In the meantime, advances in Sun's VM have made it a little faster while the usage of the older gcc-2.95.4 compiler under Debian 3.0 (stable) and run-time verification extensions in the JNuke VM have slowed it down by about 10 %. The I/O Benchmarks have been run according to the documentation in [Zwe03], where experiments were run on a Pentium III with a clock frequency of 733 MHz and 256 KB of level II cache. Advances in processor technology have meanwhile resulted in very small execution times, so the benchmark sizes were scaled up by at least one order of magnitude for each I/O benchmark. Table 9.2 shows the results from running the experiments again on a Pentium 4 with a clock frequency of 2.8 GHz and 1 MB of level II cache. Garbage collection in JNuke [Far04] was disabled for these experiments, because memory usage was low enough not to require it. The first two columns describe the benchmark used. The third column gives the run time for Sun's VM while the fourth one shows the one of JNuke. Memory consumption in JNuke is given in the last column. This figure could not be evaluated for Sun's VM, which only allows to set an upper limit to memory consumption but not to measure the actual amount of memory used.

Results are still similar to the original findings [Eug03] but the overhead of the JNuke VM is aggravated slightly. One should note that the numbers for Sun's VM benefitted a lot from its just-in-time (JIT) compiler, which JNuke does not have. Especially for applications performing a lot of computations in small loops, such a JIT compiler is indispensable for obtaining a performance that can approach the one of a compiled program. While the lack of such a compilation step is a slight advantage in the small benchmarks, it results in a slowdown of a factor greater than 300 in pathological cases. In order to get an overall comparison, the geometric mean of execution times was chosen. Unlike the arithmetic average, it does not skew results towards a few long-running benchmarks. The average slowdown thus computed amounts to only a factor of 13, which is partially due to the two examples where JNuke runs faster than Sun's VM (101 threads and excessive use of sleep). Without these examples, the average slowdown increases to a factor of 20, and 38 for benchmarks running for at least 0.5 s under Sun's VM. This is slightly higher than previous results which used an older version of Sun's VM, containing a less optimized JIT compiler [Eug03].

9.3 JNuke Model Checker

Complex properties are usually not only application-specific, but even local to a specific module [ABG$^+$03, Mey97], and still require a user-defined specification and full state space exploration in order to ensure that they are fulfilled by a given program. Therefore model checking for software is gaining popularity as its tools become more mature [BPR01, Pel01, VHB$^+$03].

The JNuke VM has been designed with model checking in mind, and thus efficient storage and access to the state space is crucial. Experiments carried out with the JNuke model checker aimed at evaluating the performance of the underlying engine, rather than the state space exploration heuristics [ASB+04]. Therefore correct, relatively small instances of the Dining Philosophers and Producer/Consumer example programs [Eug03] were chosen. Because the programs do not violate any assertions and produce no deadlocks, the entire state space has to be explored exhaustively by the model checker. JNuke offers the ExitBlock and ExitBlock/RW (EB and EB/RW, [Bru99]) heuristics to reduce the state space. They both rely on lock releases as boundaries of atomic blocks. The ExitBlock/RW heuristics also takes advantage of the fact that two atomic blocks that have no data fields in common can be treated independently. This additional partial-order reduction requires data-race freedom, which can be verified at run-time [SBN+97].

Unlike in the original publication [ASB+04], when the garbage collector for JNuke was not available yet [Far04], garbage collection was enabled for these experiments. Garbage collection resulted in reducing memory usage by about 15 % while not increasing the run time measurably (only by about 2 – 3 %).

Application	JNuke (EB/RW)				JNuke (EB)			
	Time [s]	Mem. [MB]	#ins	#ins/s	Time [s]	Mem. [MB]	#ins	#ins/s
DP 2 10	2.65	19.4	292	110	3.64	26.6	502	138
DP 3 1	0.10	1.4	5	50	0.69	5.1	89	129
DP 3 2	2.05	15.5	196	96	8.25	49.7	1,060	128
DP 3 3	15.91	80.5	1,271	80	47.76	209.4	4,620	97
PC 100	40.50	4.6	97	180	0.55	4.6	117	213
PC 1000	8.74	37.8	1,161	132	9.81	38.4	962	98
Geo. mean	1.85	13.4	185	100	4.18	25.1	539	129

Table 9.3: *Benchmark results for the JNuke model checker. The number of instructions is given in thousands.*

Table 9.3 shows the results. In the first column, application "Dining Philosophers" is abbreviated as "DP", followed by the number of threads and iterations. Application "Producer/Consumer" is labelled "PC", followed by the number of iterations. The two available partial-order reduction heuristics ExitBlock and ExitBlock/RW are compared.

It is clear that the ExitBlock/RW heuristics is very effective at reducing the search space. In the average case, the state space (number of instructions) is reduced by 66 %, improving memory usage by 49 %. However, the resulting computational overhead of the heuristics amounts to 23 %, which results in fewer instructions per second handled. Thus overall computation time drops by 57 % when using the better heuristics.

For comparison, the same tests were run on Java PathFinder (JPF) 3.1.1. This older version does not yet use thread reachability information for partial-order reduction, which is very effective at pruning the search space [VHB+03]. This is not a problem in this evaluation because it does not emphasize search heuristics.

It should be noted that JPF run times do not include the time to load the class files, because this is handled by Java itself. Furthermore, memory used to store JPF and application class files is not measured. In JPF, memory usage is given after garbage collection, because the value prior to garbage collection was not consistent between runs.

Application	JNuke (EB/RW)				JPF (lines)			
	Time [s]	Mem. [MB]	#ins	#ins/s	Time [s]	Mem. [MB]	#ins	#ins/s
DP 2 10	2.65	19.4	292	110	21.41	2.6	503	23
DP 3 1	0.10	1.4	5	50	7.76	1.8	151	19
DP 3 2	2.05	15.5	196	96	41.47	3.5	1,112	27
DP 3 3	15.91	80.5	1,271	80	151.23	8.0	3,670	24
PC 100	40.50	4.6	97	180	12.78	2.5	279	22
PC 1000	8.74	37.8	1,161	132	98.92	12.2	2,795	28
Geo. mean	1.85	13.4	185	100	33.11	4.0	539	24

Table 9.4: *Comparison between JNuke and JPF. The number of instructions is given in thousands.*

The results are summarized by Table 9.4. Results of these experiments underline the importance of effective state-space reduction methods. The ExitBlock/RW algorithm [Bru99] is already quite an improvement over the original ExitBlock algorithm. Because the given JPF version only uses single lines as an atomic block, direct run time comparisons are misleading. The number of instructions executed per second gives a much more accurate measure, showing that JNuke's model checking engine is roughly four times faster than JPF's, or over five times faster when the simpler heuristics in JNuke (EB) is taken for comparison. However, data structures in JNuke require about 3.4 times more memory.

This is partially because JNuke's low-level data structures are optimized for portability across different hardware architectures, introducing some redundancy in its data structures to ensure a consistent iteration order for hash tables regardless of insertion order or memory layout. Furthermore, data structures are optimized for fast access through uniform memory alignment for all register types, which results in an overhead of at worst a factor of two on 64-bit architectures [Eug03].[1] Most importantly, the depth-first ExitBlock/RW state space exploration is not efficient with respect to garbage collection, because only a small percentage of references can be collected at any given time, resulting in a relatively high peak memory usage [Far04].

Experiments have shown that model checking of small Java programs is feasible, modelling the entire semantics of bytecode. However, even with a fast core engine, only very small programs can be explored exhaustively with model checking. More effective state space reduction methods are needed to make this technology scale better. Furthermore, program abstraction is crucial for achieving a better performance, but not yet automated in JNuke. Chapter 11 describes future work for the model checker in more detail.

[1]Java has originally been designed for 32-bit architectures, which is the reason why references are treated as if they occupied 32 bits in memory [LY99].

9.4 Eraser

For performance evaluation and comparison against other tools [ASB$^+$04], the well-known Eraser algorithm [SBN$^+$97] has been implemented in JNuke. Table 9.5 summarizes the results. Running the Eraser algorithm in addition to the application on the JNuke VM results in an overhead of 262 % for our Eraser implementation, which is quite encouraging. However, the slowdown compared to a JIT-enabled VM is still around a factor of 25, which is comparable to other checking tools [NS03] but still offers room for improvement.

The reports can be explained as follows: The benchmark collection (Bank, Deadlock, Lottery, Shop, TicketOrder) containing seeded faults [EU04] gives rise to many reports, as was to be expected. These reports are true positives. In the JGFCrypt and Santa benchmarks, four warnings each refer to instances "leaked" by the constructor, i.e., made accessible to other threads before the constructor has finished execution. This is not part of the original Eraser algorithm but an extension that is a very strong indicator of a possible fault [Lea99, vPG01].

An implementation of thread segmentation [Har00] could yield a definite answer for these cases: If higher-level synchronization is used, then the accesses are safe. The final case reported for Santa is a plain low-level data race. The data races reported for the TSP application reach report one field. The different reports refer to four methods where accesses are not properly protected by locks. In these cases, other mechanisms such as thread-locality [tl] of data ensure thread safety [Lea99].

9.5 High-level Data Races

9.5.1 Java Path Explorer

Initial work on high-level data races, including the first implementation, was carried out at NASA Ames using the Java Path Explorer (JPaX) [AHB03]. The JPaX RV tool uses bytecode instrumentation to verify properties such as view consistency, and is described in Section 8.4.

Before analyzing applications, the implementation of the algorithm was tested using ten hand-crafted programs exhibiting different combinations of tuple accesses, such as the ones shown in Section 3.3.3. The test set included applications which contain high-level data races and others that do not. The primary purpose of this test set was to test the implementation of the view consistency algorithm. Furthermore, the tests served to fine-tune the output so it is presented in an easily readable manner. This makes evaluation of warnings fairly easy, as long as the semantics of fields used in conflicting views is known and it can be inferred whether these fields have to be used as an atomic tuple or not.

Once these tests ran successfully, four real applications were analyzed with JPaX. In addition to some of the applications described above, a Java model of a NASA planetary rover controller, named K9, was analyzed. The original code is written in C++ and contains about 35,000 lines of code, while the Java model is a heavily abstracted version with 7,000 lines. Nevertheless, it still includes the original, very complex, synchronization patterns.

Application	Sun's VM	JNuke VM				
		No check		Eraser		
	Time [s]	Time [s]	Mem. [MB]	Reports	Time [s]	Mem. [MB]
Bank	1.11	2.29	1.7	3 [1 race, 2 cons]	6.40	1.7
Daisy	0.24	6.40	16.9	0	9.14	22.8
Deadlock	0.15	0.17	1.6	2 [cons]	0.50	1.7
Din. Phil. (5000 it.)	0.31	5.90	10.8	0	8.03	10.8
JGFCrypt (3 threads)	1.99	176.75	36.6	4 [cons]	786.82	36.9
JGFSparseMatmult (3 threads)	1.66	447.72	9.9	0	3723.96	9.9
Lottery	0.21	1.20	3.4	5 [cons]	24.17	3.8
Prod./Cons. (12,000 iterations)	0.50	2.76	4.0	0	4.01	4.1
Santa[a]	6.15	0.20	1.3	5 [1 race, 4 cons]	0.29	1.6
Shop	0.21	0.12	1.6	5 [1 race, 4 cons]	0.12	1.6
SOR (1 iteration, 3 threads)	0.16	1.64	2.7	0	10.03	2.7
TicketOrder	0.13	5.80	1.8	4 [1 race, 3 cons]	62.62	4.8
TSP (size 10, 3 threads)	0.17	0.95	3.2	5 [tl]	4.56	5.3
TSP (size 15, 3 threads)	0.17	12.22	3.2	9 [tl]	101.35	5.5
Geometric mean	0.41	3.17			11.46	

[a]The JNuke VM skipped idle periods caused by all threads sleeping.

***Table 9.5**: Low-level data race analysis results using JNuke.*

Note that all Java foundation classes were excluded from the analysis. This would have increased the overlapping sets to a point where the analysis would have produced too many warnings: Every print statement using `System.out` would have resulted in overlapping sets with any other view containing an access to `System.out`.

Application	Run time [s], uninstr'd	Run time [s], instrumented	Log size [MB]	Warnings issued
Elevator	16.7	17.5	1.9	2 [cond]
SOR	0.8	343.2	123.5	0
TSP, small run (4 cities)	0.6	1.8	0.2	0
TSP, larger run (10 cities)	0.6	28.1	2.3	0
K9 Rover controller	–	–	–	1 [opt]

***Table 9.6**: High-level data race analysis results using JPaX.*

Table 9.6 summarizes the results of the experiments. All experiments were run on a Pentium III with a clock frequency of 750 MHz using Sun's Java 1.4 Virtual Machine, given 1 GB of memory. Only applications which could complete without running out of memory were considered. It should be noted that the overhead of the built-in Just-In-Time (JIT) compiler amounts to 0.4 s, so a run time of 0.6 s actually means only about 0.2 s were used for executing the Java application. The Rover application could not be executed on the same machine where the other tests were run, so no time is given there.

It is obvious that certain applications using large data sets incurred a disproportionately high overhead in their instrumented version. Many examples passed the view consistency checks without any warnings reported. For the elevator example, two false warnings referred to two symmetrical cases. In both cases, three fields were involved in the conflict. In thread t_1, the views $V(t_1) = \{\{1,3\},\{3\},\{2,3\}\}$ were inconsistent with the maximal view $v_m = \{1,2,3\}$ of t_2. While this looks like a simple case, the interesting aspect is that one method in t_1 included a *conditional* access to field 1. If that branch had been executed, view $\{2,3\}$ would actually have been $\{1,2,3\}$, and there would have been no inconsistency reported. Since not executing the branch corresponds to reading data and discarding the result, both warnings are false positives.

One warning was also reported for the NASA K9 rover code. It concerned six fields which were accessed by two threads in three methods. The responsible developer explained the large scope of the maximal view with six fields as an optimization, and hence it was not considered an error. The Remote Agent space craft controller was only available in LISP, so it could not be directly tested. However, the tool used was successfully applied to test cases reflecting different constellations including the particular high-level data race presented in Chapter 3.

9.5.2 JNuke

After the initial implementation of view consistency in JPaX [AHB03], the same algorithm was implemented in JNuke in order to allow running more benchmarks and for performance comparison [ASB$^+$04]. In the experiments given here, analysis of individual array element accesses was disabled. This setting sacrifices little precision but can speed up run-time analysis significantly for programs using large arrays. It corresponds to the configuration used in JPaX. Given a time-out of one hour, JNuke was capable of running more benchmarks successfully than JPaX, allowing to get a broader evaluation of the novel high-level data race algorithm. Table 9.7 summarizes the results from using JNuke. The most remarkable result is the fact that several examples resulted in reports due to conditional accesses. All such conditional view consistency violations are benign, and the warnings are spurious. In the TSP example, such an access pattern was only present for the second, larger test run, with size 15. The views involved in these runs were rather large, with up to 13 fields per view. The smaller test run with size 10 did not execute the code in a way that would violate view consistency.

Application	Sun's	JNuke VM				
	VM	No check		High-level data races		
	Time [s]	Time [s]	Mem. [MB]	Reports	Time [s]	Mem. [MB]
Bank	1.11	2.29	1.7	0	5.86	1.7
Daisy	0.24	6.40	16.9	0	12.76	27.6
Deadlock	0.15	0.17	1.6	0	0.44	2.0
Din. Phil. (5000 it.)	0.31	5.90	10.8	0	14.06	27.7
JGFCrypt (3 threads)	1.99	176.75	36.6	0	557.29	36.9
JGFSparseMatmult (3 threads)	1.66	447.72	9.9	0	2198.40	9.9
Lottery	0.21	1.20	3.4	0	33.59	5.3
Prod./Cons. (12,000 iterations)	0.50	2.76	4.0	0	4.48	7.0
Santa[a]	6.15	0.20	1.3	1 [cond]	0.35	1.6
Shop	0.21	0.12	1.6	0	0.12	1.6
SOR (1 iteration, 3 threads)	0.16	1.64	2.7	0	7.60	2.7
TicketOrder[b]	0.13	0.13	1.8	0	0.98	1.7
TSP (size 10, 3 threads)	0.17	0.95	3.2	0	3.61	3.2
TSP (size 15, 3 threads)	0.17	12.22	3.2	2 [cond]	83.94	3.2
Geometric mean	0.41	2.42			8.35	

[a]The JNuke VM skipped idle periods caused by all threads sleeping.

[b]A smaller data set was chosen because the efficient lock set implementation was not available yet at the time of writing, the lack of which would have skewed the benchmark for large data sets.

***Table 9.7**: High-level data race analysis results using JNuke.*

Performance wise, the view consistency algorithm performs well as a run-time verification algorithm, with an average run-time overhead of 246 %. However, as shown above, the additional overhead of the JNuke VM incurs an extra overhead compared to the fully optimized VM from Sun. The total overhead is acceptable for programs that do not perform a lot of computations, but still too high for applications performing many numerical computations.

9.5.3 View Consistency as a Fault Pattern

Experiments indicate that experienced programmers intuitively adhere to the principle of view consistency. Violations can be found, but are not very common, as shown in the experiments. Some optimizations produce warnings that constitute no error. The fact that almost all false positives result from conditional accesses show that the definition of view consistency still needs some refinement. If one takes this into account, then the view consistency property can be seen as a very effective way of inferring atomic sets of data. The fact that most programs did not show any violations underlines this. It is therefore a fault pattern that can be applied in practice, without requiring annotations. User-defined annotations may serve to describe atomic sets of data, increasing precision even further, at the cost of requiring manual specification.

9.6 Block-local Atomicity

9.6.1 Comparison to Other Atomicity-based Approaches

The block-local atomicity algorithm analyzes method-local data flow, checking for copies of shared data *(stale values)* that are used outside the critical section in which shared data was read [BL02]. Initial experiments aimed at comparing our approach to previously published atomicity-based algorithms [FF04, vPG03]. For this, a preliminary version of a static analyzer that checks for block-local atomicity has been implemented in JNuke [ASB$^+$04] and later extended to handle method calls. This extension lead to the creation of generic analysis algorithms [AB05a], which allowed more benchmarks be analyzed. Run times of our tool were faster than Praun's figures but in the same order of magnitude [vPG03] and are given below. Most of the analysis time of Praun's tool occurred in general-purpose symbolic execution while symbolic execution in our analysis was optimized for the block-local atomicity property. Flanagan's run-time verification tool was a lot slower than static analysis but its performance is comparable to our run-time analysis tool. Due to the small number of benchmarks available and differences in hardware, an exact comparison is not possible.

Table 9.8 shows the results of our experiments. The applications analyzed do not exhibit atomicity violations, except for the hedc benchmark, where a hash table is used to store data. Individual accesses to the hash table are synchronized, but the use of the data from it is not. This is a classical atomicity violation. However, it is possible that higher-level synchronization prevents actual failures when using that data; our knowledge of that program is too limited to judge this. In fact, one publication counted these warnings as benign [vPG03] while another one did not [FF04].

Benchmark	Method views [vPG03]	Reduction-based atomicity [FF04]	Block-local atomicity
DiningPhilo	0	not available	0
Elevator	2	2	2 [double check]
hedc	**5**	**4**	**3** [hash table]
JGFCrypt	0	not available	0
JGFMontecarlo	0	1	1 [segmentation]
mtrt	**3**	**6**	**1** [cache]
SOR	0	0	0
TSP	**1**	**7**	**0**

***Table 9.8**: Comparison of the block-local atomicity algorithm to other atomicity-based approaches: Number of warnings reported.*

Other warnings issued are false positives: In the elevator example, the two warnings refer to a case where a variable is checked twice, similarly to the example in Figure 10.2. In the JGFMontecarlo benchmark, only one thread is active where a potential fault was reported. This corresponds to higher-level synchronization, known as thread segmentation [Har00]. Praun's case study either missed that report or counted it towards the Java library itself [vPG03]. For mtrt, the false positive is a case where a cache structure is used. Information is only written once to that cache, but this code idiom is not accounted for by our static analyzer.

The approach presented here necessarily reports fewer atomicity violations than the run-time checker from Flanagan and Freund [FF04]. This can be expected since block-local atomicity implies method-local atomicity, and thus the number of violations of method-local atomicity constitutes an upper bound to the number of block-local atomicity violations. The number of warnings is usually the same as for Praun's approach [vPG03], but is smaller for benchmarks shown in bold face.

Compared to a previous prototype checker for stale values [BL02], our checker is significantly faster. Burrows reported 2000 source lines (LOC) per minute on unspecified hardware. JNuke checked a binary resulting from 500 lines in 0.02 s, on a Pentium 4 with a clock frequency of 2.8 GHz. Accounting for different hardware, a difference of about two orders of magnitude remains. Results show that while the method is useful for finding errors, thread locality and thread segmentation have to be taken into account to make the analysis more precise. These two properties can be implemented statically or dynamically.

9.6.2 Performance and Results of the Generic Analysis

The block-local atomicity algorithm [ABH04] has been adapted as a generic algorithm that can be used to compare static and dynamic analysis. This analysis only requires reference alias information about locks, making it a suitable candidate for both static and dynamic analysis.

The static analysis module includes a *suppression list* to avoid a few common cases of false positives. The list contains several methods which return thread-local information, corresponding to the hand-over protocol for return data [Lea99]. In addition to such methods, it contains various native methods that do not use multi-threading, but which have to be suppressed because the static analyzer cannot interpret native code. Mostly due to native methods, the suppression list has grown with each program analyzed.

Benchmark	Run-time verification			
	Reports	No ch. [s]	Time [s]	Mem. [MB]
Bank	0	2.29	4.19	1.7
Daisy	0	6.40	11.03	23.9
Deadlock[a]	0	0.25	0.33	1.8
DiningPhilo	0	5.90	9.45	20.4
JGFCrypt	0	176.75	1127.92	36.6
JGFSparseMat.	0	447.72	2102.97	9.9
Lottery	0	1.20	3.89	3.8
ProdCons	1 [buf]	2.76	4.35	7.0
Santa	0	0.20	0.25	1.4
Shop	0	0.14	0.14	1.6
SOR	0	1.64	32.95	2.5
TicketOrder	0	5.80	13.20	8.9
TSP, size 10	0	0.95	2.76	3.2
TSP, size 15	0	12.22	48.62	3.2
Geometric mean		3.25		8.49

[a] A different setting had to be chosen due to an implementation problem in the RV listener.

***Table 9.9**: Benchmark results for the block-local atomicity analysis used in run-time verification. Column "No ch." lists execution times in the JNuke VM with no run-time verification running.*

All experiments were run on a Pentium 4 with a clock frequency of 2.8 GHz and 1 MB of level II cache. Table 9.9 shows the results of run-time verification, Table 9.10 for static analysis. Both for run-time verification and static analysis, the number of reports (warnings), run time, and memory consumption are given.

Some benchmarks could only be run for static analysis because not all Java library classes used were implemented in the JNuke run-time library. They are listed in Table 9.10 only. Static analysis was applicable in cases where very few classes were missing, which did not depend on other classes and did not influence the analysis result. Such missing classes had to be investigated manually. The analysis time for hedc is rather small because a part of the application could not be analyzed with the current version of the analyzer due to the lack of full support for polymorphism. Again, missing classes were checked manually. The table omits experiments based on about 30 small programs used for testing, which were all verified correctly.

Benchmark	Static analysis		
	Reports	Time [s]	Mem. [MB]
Bank	0	0.07	1.0
Daisy	3 [ro, tl, tl]	0.15	1.9
Deadlock	0	0.06	0.9
DiningPhilo	0	0.02	0.3
Elevator	2 [dc]	0.01	0.1
hedc	3 [hash tbl.]	0.18	2.0
JGFCrypt	0	0.10	1.3
JGFMontecarlo	1	0.17	2.1
JGFSparseMat.	0	0.02	0.3
Lottery	0	0.06	0.8
mtrt	1 [cache]	0.34	3.7
ProdCons	1 [buf]	0.01	0.2
Santa	0	0.03	0.4
Shop	0	0.08	1.4
SOR	0	0.08	1.1
TicketOrder	0	0.08	1.1
TSP	0	0.09	1.1

Table 9.10: *Results for block-local atomicity used in static analysis.*

Run times for dynamic analysis are excellent, with an average overhead of 161 % compared to normal execution in the JNuke VM. This is even though Java foundation methods have been omitted from being monitored. A very effective optimization would therefore exclude any methods that can be statically proven to be safe.

Given warnings are all false positives, except for the hedc benchmark, which is explained above.[2] Those spurious warnings which are not explained above include Daisy, where they were caused by read-only [ro] and thread-local [tl] values. For the ProdCons benchmark, the stale value comes from a `synchronized` buffer [buf] and is thread-local [Lea99]. These warnings can be filtered out easily by method-local reference analysis. Such a method-local pointer analysis will greatly decrease the number of spurious warnings.

However, pointer alias analysis is an unsolvable problem in general and therefore can never prevent all possible spurious warnings. An example where static analysis would then still provide a false positive because of lock aliasing can be constructed easily. Figure 9.1 shows a contrived example where two nested locks are used. Assume that `getLock()` is too complex for a precise pointer analysis. Then static analysis will conservatively assume the two locks are different and report the use of a stale value at statement `tmp++`. However, run-time analysis will only use a single monitor block, since

[2] A more precise pointer analysis could suppress most of these warnings. Run-time verification would never report false positives concerning thread-local data, such as in the five cases in Daisy and TSP, due to fully accurate pointer information.

```
Object getLock() {
    /* assume some really complex obfuscated code here */
    return this;
}

void correctMethod() {
    Object lock1, lock2;
    int tmp;
    lock1 = getLock();
    lock2 = getLock();
    synchronized (lock1) {
        synchronized (lock2) {
            tmp = getData();
        }
        tmp++;
    }
}
```

Figure 9.1: *A false positive resulting from redundant locks.*

the two locks are equal, and not report a false positive. This scenario has been outlined before [ABH04].

The overall experience shows that the approach using generic analysis algorithms works as envisioned. Experiments clearly indicate that static analysis is a lot faster, while being less precise. The staggering difference in execution times for the two analysis types is easily explained: for SOR, for instance, the dynamic version generates many thousands of objects, on which a series of mathematical operations is performed. Therefore a test suite that only aims at verifying block-local atomicity should be designed for smaller test runs; also see Section 5.6 for coverage criteria that may be sufficient for certain multi-threaded properties. In the static version, each method is only executed once, because the algorithm is context-insensitive. This by itself reduces complexity by many orders of magnitude. In summary, given experiments show that the framework is fully applicable to real-world programs, analyzing them both statically or dynamically depending on whether one requires a fast analysis or high precision.

9.6.3 Block-local Atomicity as a Fault Pattern

The danger of possible atomicity violations in concurrent programs is undisputed [BL02, FF04, vPG03]. Nonetheless, experiments have shown that many correct programs violate atomicity by using code idioms such as thread-local data [Lea99]. By including information about thread locality and thread segmentation, it will be possible to reduce the number of spurious warnings significantly. Even at the current state, the block-local atomicity algorithm is already very useful in practice due to its speed and the difficulty of finding atomicity violations by testing. Other means of finding such failures, namely model checking, do not scale up to larger programs [Fla04].

9.7 Summary

Building a custom VM requires a lot of effort if a performance comparable to the one of commercial VMs should be achieved. JNuke's VM is getting towards such a point, but still requires more optimization. Nonetheless, it is a very competitive run-time verification and model checking engine and performs better than comparable tools in both areas. Because model checking is still not feasible for large programs, run-time verification has been the focus of this thesis. JNuke's RV API allows for an efficient implementation of common verification algorithms, which only slow down execution about three or four times, compared to normal execution in the JNuke VM. This performance makes JNuke a useful tool for applying its run-time verification algorithms to automated test suites.

All run-time verification algorithms presented here embody fault patterns, which are applicable to general-purpose software. They do not require manual annotations or human interaction, and deliver only few false positives. At the same time, they find errors quickly which are very hard to detect with other technologies, making these algorithms a very valuable contribution.

10
Related Work

As described in Chapter 2, view consistency was partially inspired by the Eraser algorithm [SBN+97]. Beyond this algorithm, related work does not only exist in software analysis, but also in database and hardware concurrency theory. Stale-value errors also cover detection of errors not found by data race analysis. The algorithm is related to previous work on atomicity violations but is an independent approach to that problem. The data flow analysis used in our algorithm is at its core an escape analysis, although it uses different entities and scopes for its analysis. Finally, atomicity is related to serializability, a weaker property that is more difficult to verify.

10.1 Data Races

Low-level data races denote access conflicts when reading or writing individual fields without sufficient lock protection [SBN+97]. For detecting data races, the *set of locks* held when accessing shared fields is checked. High-level data races turn this idea upside down and consider the *set of fields* accessed when holding a lock. View consistency serves as a consistency criterion to verify whether these accesses are semantically compatible [AHB03].

Block-local atomicity is a property which is independent of high-level data races. Figure 2.9 in Chapter 2 showed that certain faults result in high-level data races but do not violate block-local atomicity. However, the reverse is also possible, as shown in Figure 10.1, where no high-level data races occur, but stale values are present in the program. Hence the two properties are independent [WS03].

Both high-level data races and block-local atomicity build on the fact that the program is already free of underlying low-level data races, which can be detected by lock set algorithms such as Eraser [SBN+97]. The intent behind block-local atomicity is to use it in conjunction with low-level and high-level data race analyses, because these notions do not capture atomicity violations.

10.2 Atomicity of Operations

High-level data races cover inconsistencies in value accesses. Another kind of fault that is closely related to high-level data races is the idea of *atomicity* of sequences of operations, such as an entire method [FQ03]. Atomicity of operations is not directly concerned with data accessed within individual critical (`synchronized`) regions, but with the question whether these regions are sufficiently large to guarantee atomic execution of certain operations. Atomicity is a desirable property in concurrent programs [FF04, FQ03, vPG03, WS03]. In conjunction with the absence of data races, program correctness with respect to concurrently accessed data can be guaranteed.

```
synchronized(lock) {
  tmp = x.getValue();
}
tmp++;
synchronized(lock) {
  x.setValue(tmp);
}
```

***Figure 10.1**: A non-atomic operation that does not violate view consistency.*

The key idea of the most common approach to atomicity checking is the reduction of sequences of operations to serializable (atomic) actions based on the semantics of each action with respect to Lipton's reduction theory [Lip75]. In Figure 10.1, the actions of the entire increment method cannot be reduced to a single atomic block because the lock is released within the method. The reduction-based atomicity algorithm verifies whether an entire shared method is atomic. A static analysis algorithm checking an implementation against an atomicity specification is presented in [FQ03]. Recent work includes a run-time checker that does not require annotations [FF04].

A different approach to verify the atomicity of methods is based on the high-level data race algorithm. It extends the views used therein with an extra view containing the fields accessed within each method [vPG03]. The idea behind this *method view* assumes that all shared fields accessed within the scope of a method should be used atomically. For each method call m by thread t, its view $V_m(t)$ consisting of all fields accessed by m are added to the views of thread t. The intuition behind this is that `synchronized` blocks within a method generate views that may be in conflict with $V_m(t)$. In addition to this, several method calls result in several views. This can detect inconsistencies between invididual method calls (where a different control flow results in different fields being used).

High-level data races (if used without method views) do not cover such atomicity violations, although it is possible that an atomicity violation can lead to a high-level data race. Figure 10.1 shows a possible scenario where reading the value, performing an operation using it, and writing the result back are not carried out atomically. The result will be based on a possibly outdated value, because other threads may have updated the shared field, x, in the meantime. Because view consistency deals with sets of values, it cannot capture this kind of error, as shown by Wang and Stoller [WS03]. Only full knowledge about the desired atomicity can achieve this. Theoretically, view consistency could be augmented with such information to improve its precision. However, requiring a list of allowed views for each thread would defeat the purpose of requiring no user specification.

Instead, block-local atomicity is the ideal property to cover such errors. It is more precise than such previous approaches, as shown in Section 4.4. At the same time it is conceptually simpler, because modeling the data flow of instructions is much simpler than deciding whether a sequence of instructions is atomic. Atomicity by itself is not sufficient to avoid data corruption.[1] However, augmenting data race checks with view consistency and our atomicity algorithm finds more errors than one approach alone.

10.3 Database Concurrency

In database theory, shared data is stored in a database and accessed by different processes. Each process performs *transactions*, sequences of read and write operations, on the data. A sequence of these operations corresponding to several transactions is called a *history*. Based on this history, it can be inferred whether each transaction is *serializable*, i.e., whether its outcome corresponds to having run that transaction in isolation [Pap79, BHG87]. Database accesses try to avoid conflicts by construction, by structuring operations into transactions. The view consistency approach attempts to analyze behavior patterns in multi-threaded programs and to verify a similar behavior in an existing program.

There are several parallels to multi-threaded programs, which share their data in memory instead of in a database. Data races on shared fields in a multi-threaded program can be be mapped to database access conflicts on shared records. Lock protection in a multi-threaded program corresponds to an encapsulation of read and write accesses in a transaction. The key problem addressed by this thesis, having intermediate states accessible when writing non-atomically a set of fields, maps to the *inconsistent retrieval* problem in databases. In such a history, one transaction reads some data items in between updates of another transaction on these items. A correct *transaction scheduler* will prevent such an access conflict, as long as the accesses of each process are correctly encapsulated in transactions.

High-level data races concern accesses to sets of fields, where different accesses use different sets. Similar problems may be seen in databases, if the programmer incorrectly defines transactions which are too fine-grained. For example, assume a system consists of a global database and an application using reading and writing threads. The writing threads use two transactions to update the database, the reading threads access everything in a single transaction. Here, the reader's view is inconsistent, since it may read an intermediate state of the system. If the writer uses a single transaction, the fault is corrected. It is likely that the abstraction provided by database query languages such as SQL [CB74] prevents some of these problems occurring.

Meanwhile, concurrency theory as used for databases and transaction systems has been moving towards richer semantics and more general operations, which are called *activities* [SABS02]. Activities are atomic events in such a system. Like in classical

[1]Flanagan and Qadeer ignored the Java memory model when claiming that low-level data races are subsumed by atomicity [FQ03].

transactions, low-level access conflicts are prevented by a scheduler which orders these operations.

Finally, database theory also uses the term *view* under different meanings. Specifically, the two terms *view equivalence* and *view serializability* are used [BHG87]. These two terms are independent of view consistency as defined in this thesis.

So far, only single database systems have been covered. In *distributed databases*, the *virtual partitioning* algorithm exhibits a problem very similar to the view consistency problem presented here: Each transaction on an item operates on a *set of entries*, the set of all database entries for a single item, which is distributed on different sites. A *view* in this context is the set of sites with which a transaction is able to communicate [BHG87]. Ideally, a transaction has a view including all sites, so all updates are "atomic" on a global scale. However, communication delays and failures prevent this from being a practical solution. The virtual partitioning protocol [ASC85] ensures that all transactions have the same view of the copies of data that are functioning and those that are unavailable. Whereas a view in a distributed database corresponds to one data item which should be accessed atomically, a view as described in this thesis encompasses sets of distinct data items. The applicability of ideas from this protocol to the view consistency model in the multi-threading domain looks promising.

10.4 Hardware Concurrency

In hardware design and compiler construction, Lamport has made a major step towards correct shared memory architectures for multiprocessors [Lam79]. He uses *sequential consistency* as a criterion for ensuring correctness of interleaved operations. It requires all data operations to appear to have executed atomically. The order in which these operations execute has to be consistent with the order seen by individual processes.

Herlihy and Wing use a different correctness condition which they call *linearizability* [HW90]. It provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and response. Linearizability is a stronger property than sequential consistency and has the advantage that it preserves real-time ordering of operations. Although the theory is very general, it is geared towards hardware and compiler construction because it allows exploiting special properties of concurrent objects where transactions would be too restrictive. However, it is not directly applicable to multi-valued objects and seems to be incapable of capturing such high-level problems.

Lamport's notion of sequential consistency is rather restrictive and can be relaxed such that processors are allowed to read older copies of data as long as the observed behavior is indistinguishable from a conventional shared memory system [ABM93]. Mittal and Garg extended this work and Herlihy's linearizability [HW90] to multi-object operations, such as double-register compare and swap operations [MG98]. Problems occurring with such multi-object operations are very much alike to high-level data races. Unlike the approach shown in this thesis, which deals with access patterns, their approach is concerned with the interleaving of operations and based on histories as known in database literature.

10.5 Stale Values

The kind of error found by our algorithm corresponds to stale values as defined by Burrows and Leino [BL02] but is an independent approach to this question. Our algorithm compares IDs of monitor blocks to verify whether a register contains stale shared data. It has the advantage that these IDs can be used to track the source of a stale value, which is crucial when showing a failure trace to the programmer. Burrows' algorithm uses two flags *stale* and *from_critical* instead, which must by updated whenever a register changes. Unlike their approach, which is based on source code annotation, we model the semantics of Java bytecode directly. This covers the full semantics of Java, including method calls and arithmetic expressions, and allows us to discover potential non-determinism in program output, when registers are written to an output. Burrows' approach misses such an error as it involves the use of a register in a method call. Furthermore, we have a dedicated checker for this property, which is orders of magnitude faster [ABH04] than Burrows' prototype which uses the ESC/Java [FLL$^+$02] framework that was targeted to "more heavy-weight checking" [BL02].

10.6 Escape Analysis

The data flow analysis used for block-local atomicity is related to pointer escape analysis, see [Bla99, BH99, CGS$^+$99, WR99] and the more recent [Bla03], in the sense that it determines whether some entity escapes a region of interest. In our case entities are values (primitive as well as references), and the regions are synchronization sections.

For example, if the content of a field x is 5 and this value is read inside a synchronized section, and then later used outside this region, then that value has escaped. In traditional escape analysis on the other hand, typically entities are references to heap-allocated objects (not primitive values, such as an integer) and regions are methods or threads. In our case, the analysis is simpler because modeling the effect of each instruction on the stack and local variables is straightforward.

10.7 Serializability

Atomicity is sometimes too strong as a desired property, because it requires that the effect of an atomic sequence appears as a single action to the global system. Serializability is less strict; actions of one thread may be interleaved with actions of another one, if the result always corresponds to serial (atomic) execution. Atomic blocks are always serializable, but the reverse is not true. Correct programs may be serializable but not atomic [Fla04]. Serializability, while weaker than atomicity, still suffices to guarantee the consistency of thread-local and global program states. Code idioms exist where operations are performed on outdated values but still yield the same result as if they had been performed on the current value, because of double-checking.

Figure 10.2 derived from [Fla04] shows a code idiom suitable for long computations: A shared value is read and stored locally. A complex function is then computed using the

```
public void do_transaction() {
  int value, fdata;
  boolean done = false;
  while (!done) {
    synchronized (lock) {
      value = shared.field;
    }

    fdata = f(value); // long computation

    synchronized (lock) {
      if (value == shared.field) {
        shared.field = fdata;
        /* The usage of the locally computed fdata is
         * safe because the shared value is the same as
         * during the computation. Our algorithm and
         * other atomicity-based approaches report an
         * error (false positive). */
        done = true;
      }
    }
  }
}
```

Figure 10.2: *A code idiom that cannot be analyzed with block-local atomicity.*

local copy. When the result is to be written back, the writing thread checks whether the computation was based on the current value. If this was the case, the result is written; otherwise the computation is repeated with a new copy of the shared value. Note that even in a successful computation, the shared value may have been changed in between and re-set to its original value. Thus this operation is non-atomic but still serializable, and therefore correct.

Atomicity-based approaches, including the one presented in this thesis, will report an error for this example [FF04, vPG01, WS03]. Flanagan's definition of atomicity only entails visible effects of an operation; in this sense, the program is atomic but irreducible [Fla04]. On the other hand, the program violates block-local atomicity because its *action* is not atomic. It is merely wrapped in a (potentially endless) loop that creates the impression of atomicity. There is currently no approach other than model checking [VHB$^+$03] to decide whether a program is serializable. This observation does not diminish the value of our algorithm because model checking is still much too expensive to be applied to large programs [ASB$^+$04, Fla04, VHB$^+$03].

10.8 Summary

This work was inspired by low-level data races, which can be discovered without a user-defined specification using the Eraser lock set algorithm. The goal of this thesis was to extend low-level data races with additional fault-finding algorithms. High-level data races were our first such contribution, which extends the idea of unprotected data accesses to sets of fields. Possible faults thus detected in software are comparable to problems arising in parallel data bases and multi-object operations in hardware.

Unlike the notion of high-level data races, which focuses on sets of data accessed, other related work in software focuses on the atomicity of actions. Various kinds of atomicity definitions have been described, denoting actions which are not synchronized during their entire life time. Many approaches verify atomicity on a method level, which is fairly accurate in practice but does not take data flow into account. Our algorithm, block-local atomicity, is data-flow based and therefore more accurate than method-local atomicity. As an alternative to atomicity-based algorithms, the serializability property offers a weaker correctness property and thus an even more precise approach. However, it can so far only be verified through model checking, which is a severe drawback.

11

Future Work

11.1 Dynamic Analysis in JNuke

Future work in dynamic analysis can be categorized into model checking and run-time verification. For model checking, including more partial-order reduction ideas will reduce the number of paths explored. The key is taking advantage of the garbage collector during model checking because garbage collection can yield a canonical state space representation [IS00]. Further reductions can be achieved by including reachability information from the mark phase in partial-order reduction [VHB$^+$03]. Finally, the number of Java classes that can support rollback operations has to be increased. In cases where rollback is not feasible or computationally too complex, automated abstraction could be included in the tool flow to eliminate such functionality prior to model checking [BPR01, CDH$^+$00].

In run-time verification, the segmentation algorithm [Har00] can reduce false positives across all given algorithms. The main potential for optimization, however, lies in the combination with static analysis. Static identification of thread-safe fields will speed up run-time analysis. Furthermore, many method calls occur in the same context or a context similar enough to have no influence on the state of the analysis algorithm. If such equivalent contexts can be identified quickly and soundly at run time, further analysis can be suppressed. One may even give up soundness by only analyzing the first few invocations of each method, thus considerably reducing verification overhead.

The algorithms presented in this thesis, high-level data races and block-local atomicity, have shown that large classes of faults can be covered by fault patterns. Such patterns may exist for other kinds of multi-threading faults, or other widespread system operations. The success of existing fault patterns, towards which memory management checks can be counted as well, is an inspiration for searching for more such algorithms.

Other challenges exist in making the analysis output more readable. When handling data, the current version reports the same conflict for each instance of a given class (object type) that violates a certain property. This may result in a large error log, which contains redundant information. Suppressing such duplicate output would not only reduce the number of warnings issued but also offers an opportunity for analysis optimization.

The applicability of generic analysis algorithms in this context also has to be investigated further. Finally, combinations of on-the-fly and off-line trace generation technologies promise to eliminate weaknesses without compromising on their advantages.

On the implementation side, future work includes a just-in-time compiler, because a lot of execution time is still spent in the main loop of the VM. Futhermore, it has become obvious that building a custom execution environment is an enormous task. Despite its advantages, alternative approaches may achieve the goal of a functional tool more quickly for future projects. Many native methods are not implemented yet in JNuke; a bridge that can include third-party native libraries could help to reduce this implementation effort.

Because of these difficulties, other implementation approaches may be reconsidered. Instrumentation-based approaches or the use of the Java Debugging Interface [Sun04b] can take advantage of the full foundation library without having to support native methods by rewriting or wrapping them. It remains to be seen how fast optimized implementations of such architectures can be.

11.2 Static Analysis in JNuke

On the technical side, static analysis sometimes still generates more reports than necessary. The problem is similar to multiple reports (one per object instance) issued in runtime verification: Several reports may be issued for the same stale local variable, because several paths may lead to the same problem. In such cases, a single warning is usually sufficient.

Furthermore, the summary method for obtaining information when recursion is present is currently implemented in an ad-hoc way. The evaluation logic is not re-used, and the information obtained is not as precise as it could be. The design of the current implementation already has anticipated summary methods, so it will not be very hard to re-use the standard analysis while suppressing recursion. In principle, method call operations will be overridden such that recursive calls are treated as having an unknown result, while inheriting the rest of the analysis logics from the original algorithm.

Other future work includes porting more analysis algorithms to the generic analysis framework. This does not only include other fault pattern algorithms described here, but also general static checks such as static null pointer checks implemented in Jlint [AB01]. In general, static analysis in JNuke is still limited by the lack of a precise pointer analysis [WR99, WL04]. Lock approximations are currently very simple but usually accurate because most lock objects are allocated during class initialization and never changed [AB01].

In this context, the applicability of code idioms to reduce false positives looks to be promising [AH04]. Finally, visualizing potential faults found in static analysis is still an unsolved challenge; experience has shown that a simple textual output can be very hard to understand. Existing work in the field of program visualization covers visualizing event traces of distributed programs [Har00, HE91, LR85]. This is not directly applicable to static analysis, because a result from static analysis may not yet be concretized to a particular (single) trace, which poses additional challenges for visualization. It may or may not be useful to show a set of possible error traces to a developer, who is more accustomed to reasoning about a single trace.

11.3 High-level Data Races

When the view consistency algorithm generates a view within the scope of each lock, it is not yet fully understood how to properly deal with nested locks. The views of the inner locks cause conflicts with the larger views of outer locks. These conflicts are spurious. Moreover, the elevator case study has shown that a control-flow independent definition of view consistency is needed. Finally, there is a need to study the relationship to database concurrency and hardware concurrency theory in more detail.

11.4 Block-local Atomicity

Future work includes investigating the relationship to Burrows' algorithm in more depth. Our algorithm currently issues a warning when a stale register is used even though the use of such a snapshot may be benign. Burrows' more relaxed reporting could be more useful for practical purposes. Extensions to the algorithm include coverage of thread-locality of data and higher-level segmentation [Har00] of events. Such an algorithm would ideally be implemented in a generic way such that it can also be used statically and dynamically.

11.5 Concurrent Programming in the Future

As shown in this thesis, common programming languages today permit large classes of subtle multi-threading failures that can be very hard to detect. Future programming languages should aim at eliminating the potential of such errors without reducing flexibility and performance.

In the Java library, a small step towards this has been taken by including some well-established concurrency utility classes [Lea99] to its run-time library [Sun04a]. However, the use of such classes does not automatically avoid the fault patterns presented here, because the run-time environment allows these high-level components to be mixed with low-level multi-threading constructs.

Instead, the compiler and its run-time environment should be able to strictly enforce lock protection on given data sets and actions. Thus, the programmer could specify what sets of data must be treated atomically, and which actions form atomic blocks. The former would avoid low-level and high-level data races while the latter would prevent stale-value errors. In a first step, the developer may still have to specify locking, but could be supported by an environment that ensures that this locking is always sufficient to avoid concurrency errors. The long-term goal is to have to avoid explicit specification of locking altogether, using a compiler to generate any thread-specific code. This goal has been reached for some special-purpose domains such as parallel computations [BG03, CDK$^+$01, DM98]. It remains to be seen whether it can be expanded to more general programming constructs.

11.6 Summary

Support for more library classes is at the core of future work for the JNuke VM. Work for the model checker includes better partial-order reduction techniques, while run-time verification would benefit from combinations of currently implemented analysis algorithms with other properties, in order to increase speed and precision.

Reporting can still be improved for both static and dynamic analysis. Furthermore, only few verification algorithms have been implemented so far, leaving many opportunities for future work. Such work includes the search for new fault patterns.

The high-level data race property needs to be extended to a version which is independent of control flow. This presents quite a challenge for current run-time verification techniques. For block-local atomicity, no such major extension seems to be necessary.

Finally, this thesis has shown that going beyond low-level data races allows entirely new, large classes of faults to be detected. This raises the question whether current programming languages implement concurrency in the right way, leaving correctness of almost all critical properties up to the developer. It would be preferrable to have a programming language that allows specification of concurrency properties rather than concurrent program code. Parallel program structures would then be synthesized by a compiler. So far, this has only been achieved for limited domains. Future work should extend this idea to more general concurrent programs.

12 Conclusions

Concurrent or multi-threaded programming is very difficult and allows for subtle errors that do not occur in sequential programs and are unlikely to be found through traditional testing. Each alternative has its own weaknesses: Abstract interpretation does not always approximate information with sufficient accuracy. Model checking can produce exact results but does not yet scale to large software applications. Theorem proving is too labor-intensive for most projects. Facing these problems, run-time verification has established itself as an approach that can ameliorate the problem of schedule non-determinism in testing because it can verify many classes of faults on a single execution trace.

Verification of general-purpose properties is highly useful because they do not require a user-defined specification, nor does the failure looked for actually have to occur in a given execution trace. This thesis investigated such general-purpose rules, or fault patterns, that allow detection of synchronization defects. It went beyond date races, which denote access conflicts on shared fields. High-level data races are a new kind of access conflict concerning sets of data where each individual access is protected, but the entire data set is not accessed consistently. In addition to such data races, atomicity violations must be avoided as well, because they allow for further failures which are not covered by the other algorithms. This thesis presented a data-flow-based approach to detect stale values, which are outdated local copies of shared data. Unlike reduction-based approaches, the approach presented here is more precise because it does not require an entire method to be atomic.

The analysis algorithms presented here have been implemented in JNuke, which can analyze Java programs statically and dynamically. The viability of the given fault patterns has been demonstrated on a large set of benchmark applications. The architecture of JNuke is also capable of utilizing generic analysis algorithms, which can be used in both a static and dynamic setting. Differences between the two environments are abstracted into an intermediate layer, and common structures such as analysis logics and context are shared. This allows for novel combinations and interactions between static and dynamic analysis.

Bibliography

[AB01] C. Artho and A. Biere. Applying static analysis to large-scale, multi-threaded Java programs. In *Proc. 13th ASWEC*, pages 68–75, Canberra, Australia, 2001. IEEE Computer Society Press.

[AB05a] C. Artho and A. Biere. Combined static and dynamic analysis. In *Proc. 1st Intl. Workshop on Abstract Interpretation of Object-Oriented Languages (AIOOL 2005)*, ENTCS, Paris, France, 2005. Elsevier Science.

[AB05b] C. Artho and A. Biere. Subroutine inlining and bytecode abstraction to simplify static and dynamic analysis. In *Proc. 1st Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE 2005)*, ENTCS, pages 98–115, Edinburgh, Scotland, 2005. Elsevier Science.

[ABB$^+$00] W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hähnle, W. Menzel, and P. Schmitt. The KeY approach: Integrating object oriented design and formal verification. In *Proc. 7th European Workshop on Logic in Artificial Inteligence (JELIA 2000)*, volume 1919 of *LNCS*, pages 21–36, Málaga, Spain, 2000. Springer.

[ABG$^+$03] C. Artho, H. Barringer, A. Goldberg, K. Havelund, S. Khurshid, M. Lowry, C. Pasareanu, G. Roşu, K. Sen, W. Visser, and R. Washington. Combining Test Case Generation with Run-time Verification. *ASM issue of Theoretical Computer Science*, 2003. To appear.

[ABH04] C. Artho, A. Biere, and K. Havelund. Using block-local atomicity to detect stale-value concurrency errors. In *Proc. 2nd Intl. Symposium on Automated Technology for Verification and Analysis (ATVA 2004)*, volume 3299 of *LNCS*, pages 150–164, Taipei, Taiwan, 2004. Springer.

[ABM93] Y. Afek, G. Brown, and M. Merritt. Lazy Caching. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(1):182–205, 1993.

[Abr96] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[AH04] C. Artho and K. Havelund. Applying Jlint to space exploration software. In *Proc. 5th Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI 2004)*, volume 2937 of *LNCS*, pages 297–308, Venice, Italy, 2004. Springer.

[AHB03] C. Artho, K. Havelund, and A. Biere. High-level data races. *Journal on Software Testing, Verification & Reliability (STVR)*, 13(4):220–227, 2003.

[Art01] C. Artho. Finding faults in multi-threaded programs. Master's thesis, ETH Zürich, 2001.

[ASB$^+$04] C. Artho, V. Schuppan, A. Biere, P. Eugster, M. Baur, and B. Zweimüller. JNuke: Efficient Dynamic Analysis for Java. In *Proc. 16th Intl. Conf. on Computer Aided Verification (CAV 2004)*, volume 3114 of *LNCS*, pages 462–465, Boston, USA, 2004. Springer.

[ASC85] A. Abbadi, D. Skeen, and F. Cristian. An Efficient, Fault-Tolerant Protocol for Replicated Data Management. In *Proc. 4th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS 1985)*, pages 215–229, Portland, USA, 1985. ACM Press.

[BAEF03] Y. Ben-Asher, Y. Eytani, and E. Farchi. Heuristics for finding concurrent bugs. In *Proc. Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD 2003)*, page 288a, Nice, France, 2003.

[Bar97] J. Barnes, editor. *Ada 95 Rationale, The Language, The Standard Libraries*, volume 1247 of *LNCS*. Springer, 1997.

[Bau02] M. Baur. Pretty printing for JNuke. Technical report, ETH Zürich, Zürich, Switzerland, 2002.

[BCC$^+$99] A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proc. 36th ACM/IEEE conference on Design Automation (DAC 1999)*, pages 317–320, New Orleans, USA, 1999. ACM Press.

[BCC$^+$02] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software. In T. Mogensen, D. A. Schmidt, and I. H. Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, volume 2566 of *LNCS*, pages 85–108. Springer, 2002.

[BCC$^+$03] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. *Bounded Model Checking*, volume 58 of *Advances in Computers*. Elsevier, 2003. To appear.

[BCM$^+$90] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proc. 5th Annual IEEE Symposium on Logic in Computer Science (LICS 1990)*, pages 1–33, Washington, D.C., USA, 1990. IEEE Computer Society Press.

[BDG$^+$04] G. Brat, D. Drusinsky, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, W. Visser, and R. Washington. Experimental evaluation of verification and validation tools on Martian rover software. *Formal Methods in System Design*, 25(2):167–198, 2004.

[BG03] D. Bik and M. Girkar. Inside the Intel compiler. *Linux Journal*, 106:91–94, 2003.

[BGHS04] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Proc. 5th Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI 2004)*, volume 2937 of *LNCS*, pages 44–57, Venice, Italy, 2004. Springer.

[BH99] J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java. In *Proc. 14th ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 1999)*, pages 35–46, Denver, USA, 1999. ACM Press.

[BH03] S. Bensalem and K. Havelund. Deadlock Analysis of Multi-Threaded Java Programs. To be published, 2003.

[BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[BL02] M. Burrows and R. Leino. Finding stale-value errors in concurrent programs. Technical Report SRC-TN-2002-004, Compaq SRC, Palo Alto, USA, 2002.

[Bla99] B. Blanchet. Escape analysis for object-oriented languages: application to Java. In *Proc. 14th ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 1999)*, pages 20–34, Denver, USA, 1999. ACM Press.

[Bla03] B. Blanchet. Escape analysis for Java, theory and practice. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(6):713–775, 2003.

[BM04] J. Bohn and F. Mattern. Super-distributed RFID tag infrastructures. In *Proc. 2nd European Symposium on Ambient Intelligence (EUSAI 2004)*, volume 3295 of *LNCS*, pages 1–12, Eindhoven, The Netherlands, 2004. Springer.

[Bot98] P. Bothner. Kawa — compiling dynamic languages to the Java VM. In *Proc. USENIX 1998 Technical Conf., FREENIX Track*, New Orleans, USA, 1998. USENIX Association.

[Bou93] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proc. Formal Methods in Programming and their Applications*, volume 735 of *LNCS*, pages 128–141, Novosibirsk, Russia, 1993. Springer.

[BPR01] T. Ball, A. Podelski, and S. Rajamani. Boolean and Cartesian Abstractions for Model Checking C Programs. In *Proc. 7th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*

(TACAS 2001), volume 2031 of *LNCS*, pages 268–285, Genova, Italy, 2001. Springer.

[Bri99] E. Briot. JGNAT: The GNAT Ada 95 environment for the JVM. In *Ada France*, Brest, France, 1999.

[Bru99] D. Bruening. Systematic testing of multithreaded Java programs. Master's thesis, MIT, 1999.

[Bry86] R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[BS03] E. Börger and R. F. Stärk. *Abstract State Machines — A Method for High-Level System Design and Analysis*. Springer, 2003.

[BSW$^+$99] J. Bull, L. Smith, M. Westhead, D. Henty, and R. Davey. A methodology for benchmarking Java Grande applications. In *Proc. ACM Java Grande Conference*, pages 81–88, San Francisco, USA, 1999.

[Bur00] M. Burrows. Personal communication, 2000.

[But97] D. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.

[CB74] D. Chamberlin and R. Boyce. SEQUEL: A structured English query language. In *Proc. First ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pages 249–264, Ann Arbor, USA, 1974. ACM Press.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM Symposium on Principles of Programming Languages (POPL 1977)*, pages 238–252, Los Angeles, USA, 1977. ACM Press.

[CDH$^+$00] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. 22nd Intl. Conf. on Software Engineering (ICSE 2000)*, pages 439–448, Limerick, Ireland, 2000. ACM Press.

[CDHR00] J. Corbett, M. Dwyer, J. Hatcliff, and Robby. A language framework for expressing checkable properties of dynamic software. In *Proc. 7th Intl. SPIN Workshop (SPIN 2000)*, volume 1885 of *LNCS*, pages 205–223, Stanford, USA, 2000. Springer.

[CDK$^+$01] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers, Inc., 2001.

[CdR04] F. Chen, M. d'Amorim, and G. Roşu. A formal monitoring-based framework for software development and analysis. In *Proc. 6th Intl. Conf. on Formal Engineering Methods (ICFEM 2004)*, volume 3308 of *LNCS*, pages 357–372, Seattle, USA, 2004. Springer.

[CGP99] E. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 1999.

[CGS$^+$99] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proc. 14th ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 1999)*, pages 1–19, Denver, USA, 1999. ACM Press.

[Chu36] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936.

[CL02] Y. Cheon and G. Leavens. A run-time assertion checker for the Java Modeling Language. In *Proc. Intl. Conf. on Software Engineering Research and Practice (SERP 2002)*, pages 322–328, Las Vegas, USA, 2002. CSREA Press.

[Coh01] S. Cohen. JTrek, 2001. Developed by Compaq, now discontinued.

[Cor98] Standard Performance Evaluation Corporation. SPEC benchmarks, 1998. `http://www.spec.org/`.

[CP96] C.-T. Chou and D. Peled. Verifying a model-checking algorithm. In *Proc. 2nd Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 1996)*, volume 1055 of *LNCS*, pages 241–257, Passau, Germany, 1996. Springer.

[Cra57] W. Craig. Linear reasoning. A new form of the Herbrand-Gentzen theorem. *Journal of Symbolic Logic (JSL)*, 22(3):250–268, 1957.

[CTW99] M. Chaudron, J. Tretmans, and K. Wijbrans. Lessons from the Application of Formal Methods to the Design of a Storm Surge Barrier Control System. In *Proc. World Congress on Formal Methods in the Development of Computing Systems II (FM 1999)*, volume 1709 of *LNCS*, pages 1511–1526, Toulouse, France, 1999. Springer.

[Dah03] M. Dahm. BCEL, 2003. `http://jakarta.apache.org/bcel/`.

[Dij68] E. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.

[Dij72] E. Dijkstra. Notes on structured programming. *Structured Programming*, 1972.

[DLNS98] D. Detlefs, R. Leino, G. Nelson, and J. Saxe. Extended Static Checking. Technical Report 159, Compaq Systems Research Center, Palo Alto, USA, 1998.

[DM98] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computation in Science and Engineering Magazine*, 5(1):46–55, 1998.

[Dru03] D. Drusinsky. Monitoring temporal rules combined with time series. In *Proc. 15th Intl. Conf. on Computer Aided Verification (CAV 2003)*, volume 2725 of *LNCS*, pages 114–118, Boulder, USA, 2003. Springer.

[EM04] D. Engler and M. Musuvathi. Static analysis versus software model checking for bug finding. In *Proc. 5th Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI 2004)*, volume 2937 of *LNCS*, pages 191–210, Venice, Italy, 2004. Springer.

[EU04] Y. Eytani and S. Ur. Compiling a benchmark of documented multi-threaded bugs. In *Proc. 18th Intl. Parallel & Distributed Processing Symposium (IPDPS 2004)*, page 266a, Santa Fe, USA, 2004. IEEE Computer Society Press.

[Eug03] P. Eugster. Java Virtual Machine with rollback procedure allowing systematic and exhaustive testing of multithreaded Java programs. Master's thesis, ETH Zürich, 2003.

[Far04] P. Farkas. Garbage Collection for JNuke, a Java Virtual Machine for Run-time Verification and Model Checking. Master's thesis, ETH Zürich, 2004.

[FF04] C. Flanagan and S. Freund. Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. In *Proc. 31st ACM Symposium on Principles of Programming Languages (POPL 2004)*, pages 256–267, Venice, Italy, 2004. ACM Press.

[Fla04] C. Flanagan. Verifying Commit-Atomicity using Model-Checking. In *Proc. 11th Intl. SPIN Workshop (SPIN 2004)*, volume 2989 of *LNCS*, pages 252–266, Barcelona, Spain, 2004. Springer.

[FLL$^+$02] C. Flanagan, R. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *Proc. ACM Intl. Conf. on Programming Language Design and Implementation (PLDI 2002)*, pages 234–245, Berlin, Germany, 2002. ACM Press.

[FNU03] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *Proc. 20th IEEE Intl. Parallel & Distributed Processing Symposium (IPDPS 2003)*, page 286, Nice, France, 2003. IEEE Computer Society Press.

[FQ03] C. Flanagan and S. Qadeer. Types for atomicity. In *Proc. ACM Intl. Workshop on Types in Language Design and Implementation (TLDI 2003)*, pages 1–12, New Orleans, USA, 2003. ACM Press.

[FQ04] S. Freund and S. Qadeer. Checking concise specifications for multi-threaded software. *Journal of Object Technology (JOT)*, 3(6):81–101, 2004.

[Fre98] S. Freund. The costs and benefits of Java bytecode subroutines. In *Formal Underpinnings of Java Workshop at OOPSLA*, Vancouver, Canada, 1998.

[GH03] A. Goldberg and K. Havelund. Instrumentation of Java bytecode for runtime analysis. In *Proc. 5th ECOOP Workshop on Formal Techniques for Java-like Programs (FTfJP 2003)*, volume 408 of *Technical Reports from ETH Zürich*, pages 151–159, Darmstadt, Germany, 2003. ETH Zürich.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, USA, 1995.

[GJSB00] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley, 2000.

[Göd31] K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931. English translation: On formally undecidable propositions of Principia Mathematica and Related Systems I, Oliver & Boyd, London, UK, 1962.

[God97] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proc. 24th ACM Symposium on Principles of Programming Languages (POPL 1997)*, pages 174–186, Paris, France, 1997. ACM Press.

[Har00] J. Harrow. Run-time checking of multithreaded applications with Visual Threads. In *Proc. 7th Intl. SPIN Workshop (SPIN 2000)*, volume 1885 of *LNCS*, pages 331–342, Stanford, USA, 2000. Springer.

[Hav00] K. Havelund. Using run-time analysis to guide model checking of Java programs. In *Proc. 7th Intl. SPIN Workshop (SPIN 2000)*, volume 1885 of *LNCS*, pages 245–264, Stanford, USA, 2000. Springer.

[HDT87] S. Horwitz, A. Demers, and T. Teitebaum. An efficient general iterative algorithm for dataflow analysis. *Acta Informatica*, 24(6):679–694, 1987.

[HE91] M. Heath and J. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, 1991.

[HJ92] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proc. Winter USENIX Conf. (USENIX 1992)*, San Francisco, USA, 1992. USENIX Association.

[HLP01] K. Havelund, M. Lowry, and J. Penix. Formal Analysis of a Space Craft Controller using SPIN. *IEEE Transactions on Software Engineering*, 27(8):749–765, 2001. An earlier version occurred in the *Proc. 4th Intl. SPIN workshop*, 1998, Paris, France.

[Hoa83] C. Hoare. Communicating sequential processes. *Communications of the ACM*, 26(1):100–106, 1983.

[Hol91] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[HP00] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Intl. Journal on Software Tools for Technology Transfer (STTT)*, 2(4):366–381, 2000.

[HR01] K. Havelund and G. Roşu. Monitoring Java programs with Java PathExplorer. In *Proc. 1st Intl. Workshop on Run-time Verification (RV 2001)*, volume 55 of *ENTCS*, pages 97–114, Paris, France, 2001. Elsevier.

[HR02] K. Havelund and G. Roşu. Synthesizing monitors for safety properties. In *Proc. 8th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2002)*, volume 2280 of *LNCS*, pages 342–356, Grenoble, France, 2002.

[HS99] G. Holzmann and M. Smith. A practical method for verifying event-driven software. In *Proc. 21st Intl. Conf. on Software Engineering (ICSE 1999)*, pages 597–607, Los Angeles, USA, 1999. ACM Press.

[HV99] M. Henning and S. Vinoski. *Advanced CORBA programming with C++*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[HW90] M. Herlihy and J. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[IEE83] IEEE. IEEE Standard 729: Glossary of software engineering terminology, 1983.

[IS00] R. Iosif and R. Sisto. Using garbage collection in model checking. In *Proc. 7th Intl. SPIN Workshop (SPIN 2000)*, volume 1885 of *LNCS*, pages 20–33, Stanford, USA, 2000. Springer.

[JK97] R. Jones and P. Kelly. Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs. In *Proc. 3rd Intl. Workshop on Automated and Algorithmic Debugging (AADEBUG 1997)*, pages 13–26, Linköping, Sweden, 1997. Linköping University Electronic Press.

[KHH$^+$01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. *LNCS*, 2072:327–355, 2001.

[KKL$^+$01] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-MaC: a run-time assurance tool for Java programs. In *Proc. 1st Intl. Workshop on Run-time Verification (RV 2001)*, volume 55 of *ENTCS*, pages 115–132, Paris, France, 2001. Elsevier.

[KN03] G. Klein and T. Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 298(3):583–626, 2003.

[Kna87] E. Knapp. Deadlock detection in distributed databases. *ACM Computing Surveys*, 19(4):303–328, 1987.

[KR88] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice-Hall, 1988.

[Lam79] L. Lamport. How to Make a Multiprocessor that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 9:690–691, 1979.

[Lea99] D. Lea. *Concurrent Programming in Java, Second Edition*. Addison-Wesley, 1999.

[Lea00] D. Lea. Personal e-mail communication, 2000.

[LF03] J. Link and P. Fröhlich. *Unit Testing in Java: How Tests Drive the Code*. Morgan Kaufmann Publishers, Inc., 2003.

[Lib91] D. Libes. expect: Scripts for controlling interactive processes. *Computing Systems*, 4(2):99–125, 1991.

[Lip75] R. Lipton. Reduction: a method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.

[LR85] R. LeBlanc and A. Robbins. Event-driven monitoring of distributed programs. In *Proc. 5th Intl. Conf. on Distributed Computing Systems (ICDCS 1985)*, pages 515–522, Denver, USA, 1985. IEEE Computer Society Press.

[LY99] T. Lindholm and A. Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison-Wesley, 1999.

[LYK$^+$00] S. Lee, B. Yang, S. Kim, S. Park, S. Moon, K. Ebciolu, and E. Altman. Efficient Java exception handling in just-in-time compilation. In *Proc. ACM Java Grande Conference*, pages 1–8, San Francisco, USA, 2000. ACM Press.

[Mar01] R. Marlet. Syntax of the JCVM language to be studied in the SecSafe project. Technical Report SECSAFE-TL-005-1.7, Trusted Logic SA, Versailles, France, 2001.

[McM93] K. McMillan. *Symbolic Model Checking*. Springer, 1993.

[McM03] K. McMillan. Interpolation and SAT-based model checking. In *Proc. 15th Intl. Conf. on Computer Aided Verification (CAV 2003)*, volume 2725 of *LNCS*, pages 1–13, Boulder, USA, 2003. Springer.

[MD97] J. Meyer and T. Downing. *Java Virtual Machine*. O'Reilly & Associates, Inc., 1997.

[Mey97] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2nd edition, 1997.

[MG98] N. Mittal and V. Garg. Consistency conditions for multi-object distributed operations. In *Proc. Intl. Conf. on Distributed Computing Systems (ICDCS 1998)*, pages 582–599, Amsterdam, The Netherlands, 1998. IEEE Computer Society Press.

[MH02] J. Miecznikowski and L. Hendren. Decompiling Java bytecode: Problems, traps and pitfalls. In *Proc. 11th Intl. Conf. on Compiler Construction (CC 2002)*, volume 2304, pages 111–127, Grenoble, France, 2002. Springer.

[Moh02] M. Mohnen. A graph-free approach to data-flow analysis. In *Proc. 11th Intl. Conf. on Compiler Construction (CC 2002)*, volume 2304, pages 46–61, Grenoble, France, 2002. Springer.

[MR90] T. Marlowe and B. Ryder. An efficient hybrid algorithm for incremental data flow analysis. In *Proc. 17th ACM Symposium on Principles of Programming Languages (POPL 1990)*, pages 184–196, San Francisco, USA, 1990. ACM Press.

[Mye79] G. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., 1979.

[NBF98] B. Nichols, D. Buttlar, and J. Farrell. *Pthreads Programming*. O'Reilly, 1998.

[NS03] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *Proc. 3rd Intl. Workshop on Run-time Verification (RV 2003)*, volume 89 of *ENTCS*, pages 22–43, Boulder, USA, 2003. Elsevier.

[Pap79] C. Papadimitriou. The Serializability of Concurrent Database Updates. *Journal of the ACM (JACM)*, 26(4):631–653, 1979.

[Pau03] C. Paulson. Mechanizing compositional reasoning for concurrent systems: Some lessons. Technical Report UCAM-CL-TR-573, Computer Laboratory, University of Cambridge, Cambridge, UK, 2003.

[PD00] T. Printezis and D. Detlefs. A generational mostly-concurrent garbage collector. In *Proc. 2nd international symposium on Memory management (ISMM 2000)*, pages 143–154, Minneapolis, USA, 2000. ACM Press.

[PDV03] C. Pasareanu, M. Dwyer, and W. Visser. Finding feasible abstract counter-examples. *Intl. Journal on Software Tools for Technology Transfer (STTT)*, 5(1):34–48, 2003.

[Pel01] D. Peled. *Software Reliability Methods*. Springer, 2001.

[PGK$^+$97] B. Pell, E. Gat, R. Keesing, N. Muscettola, and B. Smith. Plan Execution for Autonomous Spacecrafts. In *Proc. Intl. Joint Conf. on Artificial Intelligence (IJCAI 1997)*, pages 1234–1239, Nagoya, Japan, 1997. Morgan Kaufmann Publishers, Inc.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 17th Annual Symposium on Foundations of Computer Science (FOCS 1977)*, pages 46–57, Rhode Island, USA, 1977. IEEE, IEEE Computer Society Press.

[RJB98] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley Object Technology Series, 1998.

[Roy02] G. Roy. mpatrol — A library for controlling and tracing dynamic memory allocations, 2002. `http://www.cbmamiga.demon.co.uk/mpatrol/`.

[RV04] *1st, 2nd, 3rd and 4th Intl. Workshops on Run-time Verification (RV 2001 - RV 2004)*, volume 55(2), 70(4), 89(2), 113 of *ENTCS*. Elsevier Science, 2001 – 2004.

[SABS02] H. Schuldt, G. Alonso, C. Beeri, and H.-J. Schek. Atomicity and Isolation for Transactional Processes. *ACM Transactions on Database Systems (TODS)*, 27(1):63–116, 2002.

[SBB04] V. Schuppan, M. Baur, and A. Biere. JVM-independent replay in Java. In *Proc. 4th Intl. Workshop on Run-time Verification (RV 2004)*, volume 113 of *ENTCS*, pages 85–104, Málaga, Spain, 2004. Elsevier.

[SBN$^+$97] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[Sch00] J. Schumann. *Automated Theorem Proving in Software Engineering*. Springer, 2000.

[Sin89] M. Singhal. Deadlock detection in distributed systems. *IEEE Computer*, 22(11):37–48, 1989.

[Sit05] Sitraka. JProbe, 2005. `http://www.quest.com/jprobe/`.

[SSB01] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine*. Springer, 2001.

[Sto00] S. Stoller. Model-Checking Multi-threaded Distributed Java Programs. In *Proc. 7th Intl. SPIN Workshop (SPIN 2000)*, volume 1885 of *LNCS*, pages 224–244, Stanford, USA, 2000. Springer.

[Sto02] S. Stoller. Testing concurrent Java programs using randomized scheduling. In *Proc. 2nd Intl. Workshop on Run-time Verification (RV 2002)*, volume 70(4) of *ENTCS*, pages 143–158, Copenhagen, Denmark, 2002. Elsevier.

[Sun97] Sun Microsystems. A simple multithreaded web server, 1997. `http://java.sun.com/developer/technicalArticles/Networking/Webserver/`.

[Sun04a] Sun Microsystems, Santa Clara, USA. *Java 2 Platform Standard Edition (J2SE) 1.5*, 2004. `http://java.sun.com/j2se/1.5.0/`.

[Sun04b] Sun Microsystems, Santa Clara, USA. *Java 2 SDK, Standard Edition Documentation*, 2004. `http://java.sun.com/j2se/1.4/docs/`.

[Sun05] Sun Microsystems, Santa Clara, USA. *Java 2 Platform Enterprise Edition Specification*, 2005. `http://java.sun.com/j2ee/`.

[Tan92] A. Tanenbaum. *Modern operating systems*. Prentice-Hall, 1992.

[Tro94] J. Trono. A new exercise in concurrency. *SIGCSE Bulletin*, 26(3):8–10, 1994.

[TRY$^+$87] A. Tevanian, R. Rashid, M. Young, D. Golub, M. Thompson, W. Bolosky, and R. Sanzi. A UNIX interface for shared memory and memory mapped files under mach. In *Proc. Summer USENIX Conf. (USENIX 1987)*, pages 53–68, Phoenix, USA, 1987. USENIX Association.

[Tur37] A. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Journal of the London Mathematical Society*, 42(2):230–265, 1937.

[vdBJ01] J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In *Proc. 7th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *LNCS*, pages 299–312, Genova, Italy, 2001. Springer.

[vDMvdBK01] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok. Refactoring test code. In *Proc. 2nd Intl. Conf. on eXtreme Programming and Flexible Processes in Software Engineering (XP 2001)*, Cagliari, Italy, 2001. University of Cagliari.

[VHB$^+$03] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, 2003.

[vPG01] C. von Praun and T. Gross. Object-race detection. In *Proc. 16th ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2001)*, pages 70–82, Tampa Bay, USA, 2001. ACM Press.

[vPG03] C. von Praun and T. Gross. Static detection of atomicity violations in object-oriented programs. In *Proc. 5th ECOOP Workshop on Formal Techniques for Java-like Programs (FTfJP 2003)*, volume 408 of *Technical Reports from ETH Zürich*, pages 99–108, Darmstadt, Germany, 2003. ETH Zürich.

[VR00] R. Vallée-Rai. Soot: A Java bytecode optimization framework. Master's thesis, McGill University, Montreal, 2000.

[VRHS$^+$99] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot – a Java optimization framework. In *Proc. CASCON 1999*, pages 125–135, Toronto, Canada, 1999.

[Wil02] M. Wildmoser. Subroutines and Java bytecode verification. Master's thesis, Technical University of Munich, 2002.

[WL04] J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proc. ACM Intl. Conf. on Programming Language Design and Implementation (PLDI 2004)*, pages 131–144, Washington D.C., USA, 2004. ACM Press.

[WR99] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proc. 14th ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 1999)*, pages 187–206, Denver, USA, 1999. ACM Press.

[WS03] L. Wang and S. Stoller. Run-time analysis for atomicity. In *Proc. 3rd Intl. Workshop on Run-time Verification (RV 2003)*, volume 89(2) of *ENTCS*, pages 224–242, Boulder, USA, 2003. Elsevier.

[Zwe03] B. Zweimüller. I/O Erweiterung für die JNuke Virtual-Machine (I/O extension for the JNuke Virtual Machine). Technical report, ETH Zürich, Zürich, Switzerland, 2003.

Curriculum Vitae

Cyrille Artho

June 17, 1976	Born in Zurich, Switzerland
1983 – 1988	Primary School, Zufikon
1988 – 1992	Bezirksschule Bremgarten
1992 – 1996	Gymnasium Wohlen, Type C (Sciences)
1996 – 2001	Studies in Computer Science, ETH Zurich
1998 – 1999	Exchange Studies, University of Strathclyde, Glasgow, UK
1999	Internship at Unitek Engineering AG, Zurich
2001	Completion of Master's Thesis, ETH Zurich and Trilogy Software, Austin, USA
2001 – 2005	Research and Teaching Assistant, Computer Systems Institute, ETH Zurich
2002 and 2003	Research collaboration with Klaus Havelund and the Automated Software Engineering (ASE) Group, NASA Ames Research Center, Moffett Field, USA
2005	Completion of Ph.D. Thesis, ETH Zurich