Java Virtual Machine with Rollback Procedure allowing Systematic and Exhaustive Testing of Multi-threaded Java Programs

Pascal Eugster <pe@student.ethz.ch>

10th April 2003

Abstract

Many Java programs have sources of non-determinism, where a program depends upon factors that cannot be controlled. Such sources include (true) random numbers or the inherent non-determinism in concurrent threads, where the thread schedule is determined by the environment and varies between executions. This makes classical testing and fault-finding very inefficient: due to the non-deterministic nature of concurrent programs, a program run can no longer be reproduced reliably, and a fault may not manifest itself except under rare circumstances. Therefore the wish arises to test a program, under all possible outcomes. Different thread schedules are of primary interest.

The main goal was to create a virtual machine that aims to expose its internal execution environment to tools in a consistent, well-documented way. It also includes a rollback mechanism, that allows establishing checkpoints in program execution, which can be reverted to in order to test another schedule.

On top of our custom virtual machine, a testing algorithm was implemented that systematically finds errors resulting from unintended timing dependencies. This work implements the *ExitBlock* and the *ExitBlock-RW* algorithm presented in Derek Bruening's master thesis [7]. Both algorithms execute a program or parts thereof on a given input multiple times, enumerating meaningful schedules in order to cover all program behaviours. A key challenge is to minimize the number of schedules. *ExitBlock* and the *ExitBlock-RW* achieve this by enumerating possible orders of synchronized regions, provided the target program follows a mutual-exclusion locking discipline. However, the number of resulting schedules is still to high. Real programs therefore cannot be exhaustively tested as a whole. This thesis presents the implementation of *ExitBlock* and *ExitBlock-RW* in our custom virtual machine, and demonstrates how possible assertion violations, and most potential deadlocks are discovered.

Acknowledgements

I would like to thank *Prof. Armin Biere* of the *Swiss Federal Institute of Technology* for giving me the opportunity to participate in such a challenging and interesting project.

This Diploma Work would not have been gotten so far without the guidance and tutelage of my assistant *Cyrille Artho*. He gave me invaluable guidance at every stage of the work – from planning over development to proofreading. He was never tired of giving feedback.

More thanks to *Viktor Schuppan* for taking the time to revise this report meticulously. He gave me many ideas how to improve this report.

Contents

Chapter 1

Introduction

A concurrent program consists of several *threads* or *processes*. A process has its own program counter, its own stack, register set, and address space. Processes have nothing to do with each other, except that they may be able to communicate through the system's interprocess communication primitives, such as semaphores, monitors or messages [3]. In contrast to processes, threads have the same address space which means they share global variables allowing communication. Threads are sometimes called *lightweight processes* [42]. The Java programming language inherently follows the concept of threads. All Java threads share the same memory and thus all variables of objects may be accessed from different threads.

The *operating system* executes threads and processes in parallel. On a uni-processor machine however, threads do not actually execute in parallel. A uni-processor environment simulates parallelism by interleaving processes or threads using time-sharing. Real parallel execution presumes a multi-processing environment.

Concurrency allows to perform multiple computations in parallel and to control multiple external activities which occur at the same time. It can also increase the throughput and responsiveness. Enterprise applications increasingly rely on concurrency in order to process many user requests at the same time [2]. Graphical user interfaces often use concurrency as well, so they remain responsive to user interaction even during a calculation.

The execution order of a concurrent program is nondeterministic due to the apparent randomness in the way threads are scheduled. Different execution orders of threads or processes may result in different program behaviours. Some of them might be erroneous. Errors that underlie the timing of threads are called *timing-dependent errors*. Due to the nondeterministic nature of concurrent programs, classical testing techniques for sequential programs are far less effective. Techniques for testing sequential programs usually consist of test suites where the target program is run on different representative sets of input data. Applied to a concurrent program however only one fraction of the possible schedules is covered. A concurrent program may pass a traditional test suite in spite of timing-dependent errors. Even if a timing-dependent error is found, there is no guarantee that the same erroneous behaviour can be reproduced by rerunning the same test case with the same input data as the outcome of a concurrent program is determined by a pair of input and schedule.

Thus, this thesis describes algorithms for exhaustively enumerating possible behaviours of a Java program for a given input. Our motivation is to discover all erroneous behaviours by systematically exploring a concurrent program's behaviour.

A tester therefore needs to exhaustively explore all possible schedules for each representative input. A naïve approach would consist of interleaving each statement of a thread with each statement of all other threads. The problem with simply enumerating possible schedules of the program is the exponential increase in the number of schedules that has to be considered. Thus, this naïve approach is hardly applicable for real concurrent programs. The *ExitBloc*k algorithm [7] assumes that a program follows a *mutual exclusion locking discipline.* It enumerates orders of synchronized regions, which still covers all behaviours. However, the number of explored schedules by *ExitBlock* still increases exponentially in both the number of threads and the number of lock uses by each thread. The solution is to investigate which schedules lead to the same behaviour and thus execute as few schedules as possible. This idea is followed by *ExitBlock-RW* [7], an extension of *ExitBlock*, which does not reorder regions that do not share data. The number of schedules to be considered is reduced. It results in a polynomial growth in average case. In this thesis a systematic tester implementing *ExitBlock* and *ExitBlock*-RW is described. It provides behaviour-complete testing for multi-threaded Java programs by enumerating possible schedules.

ExitBlock uses a depth-first search. It first executes one complete schedule of the program. Then, *ExitBlock* backs up from the end of the program to the last synchronized region boundary at which a different thread is chosen to schedule. This leads to a new schedule branching off from this point in the program. This process is systematically repeated, continuing to back up to synchronized region boundaries choosing different threads.

The described depth-first search presumes that the state of a program can be stored in order to return to a previous state. Thus, there has to be a mechanism that allows establishing check points in program execution, which can be reverted to. We call this *milestone/rollback mechanism*.

Figure 1.1 shows different approaches that come into question. The implementation of the rollback/milestone mechanism can take place at any layer of the system stack. The according layer is then either replaced as a whole or parts thereof by a custom implementation. Beside replacing a layer, it is also possible to control an existing layer. Table 1.1 gives an overview about possible approaches detailed below.

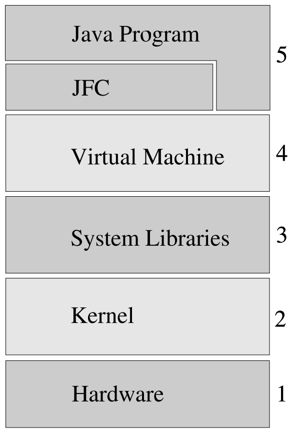

Figure 1.1: The system stack from the Java program to the hardware.

Layer	Description	Flexibility	Effort	Performance
5	Instrumentation of the Java program	$-$	low	$+/-$
4	Replacing the virtual machine as a whole or parts thereof	$+$	low	$+/-$
3	Replacing parts of the POSIX API	$-$	high	$+$
3	Steering execution with strace, ptrace and other system tools	$-$	high	$+$
2	Modification of the OS kernel	$-$	very high	$+$
1	Dedicated hardware supporting rollbacks	$-$	very high	$+$

Table 1.1: Overview about possible ways to implement a milestone/rollback mechanism at different system layers introduced in Figure 1.1.

The first approach is located at the fifth layer where a package of Java classes is provided implementing the rollback mechanism. Maybe *Foundation Classes* need to be rewritten as well. A target Java program is then instrumented using this package and executed by an arbitrary virtual machine. The implementation of the libraries providing the rollback mechanisms and the instrumenting of the Java code is rather difficult to realize as a Java program has hardly any access to the underlying virtual machine. This approach, however, could profit of the optimization of the used virtual machine, such as a just-in-time compiler.

It is also possible to replace the virtual machine or parts thereof. This thesis follows this approach. Our virtual machine is written from scratch, which comes with several considerable advantages. First, the virtual machine is implemented such that it exposes its internal execution environment to tools in a consistent, well-documented way. Its event system easily allows tools to access run-time information and control the virtual machine. Second, a custom virtual machine allows to chose a design well supporting the implementation of the rollback mechanism. However, a custom virtual machine cannot profit of all the optimizations provided by commercial virtual machines. This results in lower performance. The Rivet virtual machine [7] written in Java itself, running on top of any other virtual machine, is a partial virtual machine. Thus, Rivet can make use of lower virtual machine's platform-specific features and inherently profit of optimizations of the underlying virtual machine. Development, however, has been discontinued as Rivet replaces foundation classes, which is disallowed by current virtual machines.

Another approach is to implement a software layer beneath the existing virtual machine which replaces parts of the POSIX API [33] level and allows to control thread switches as well as memory accesses (provided the target virtual machine uses the POSIX API for threading support). Replacing the POSIX API may raise some compatibility issues. Working with strace/ptrace [3] avoids this. These tools allow to monitor and influence system calls. It may be possible steer the system this way. Also, we can go to deeper layers down to the hardware. The effort, however, increasingly grows, and portability becomes a problem. It is also supposable to implement a virtual machine on top of the host operation system that executes another operation system or the Java virtual machine [46, 44].

As mentioned, a virtual machine, written from scratch, exposing its interfaces, easily allows to implement runtime checkers. It provides a good flexibility paired with a reasonable effort. Thus, our group decided to write a custom virtual machine extended with rollback facilities based on the JNuke framework introduced in Section 1.2. Due to the early stage, our custom virtual machine is much slower than Sun's virtual machine. However, it is able to execute any program in reasonable time provided the target program does not perform I/O operations. Thes are not implemented yet. *ExitBlock* and *ExitBlock-RW* are implemented on top of our virtual machine. Experiments in Chapter 5 demonstrate that possible assertion violations and most deadlocks are discovered. Examples like dining philosophers with up to twenty threads are considered. A dining philosophers program with twenty concurrent threads leads to 300'000 schedules explored within five minutes on a recent workstation[1].

1.1 Considered concurrency errors

Various timing-dependent errors can occur in a concurrent program. This thesis considers the following timing-dependent errors:

Race Conditions also called *data races* can be defined as follows. A data race occurs 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 [36]. Race conditions can be detected by the *Eraser* algorithm [36]. It observes what locks are held by each thread on each variable access.

Deadlocks are a cycle of resource dependencies that leads to a state in which threads are blocked from execution for infinite time. Lock cycle analyzes can detect deadlocks. A lock graph without any cycles ensures the absence of deadlocks [7].

Specific Properties are often tested through assertions at run-time and may fail because of an unexpected schedule. For instance, this can happen if conditions are not rechecked when several threads are awakened by `notifyAll`. Violations of properties are hard to track down and can only reliably found by systematically exploring a concurrent program's behaviours.

ExitBlock and *ExitBlock-RW* discover assertion violations and deadlocks. If the Eraser algorithm [36] is run in parallel data races can be found, too. Examples like *dining philosophers*, *producer-consumer problems*, and others show that even difficult timing-dependent errors are located. The described implementation is able to analyze small programs containing a small number of threads and synchronized regions within a couple of minutes.

1.2 The JNuke framework

The virtual machine is based on the JNuke framework. This framework provides an object-oriented concept for C, commonly used container and other utility classes, and, most importantly, a Java classloader. This classloader performs a transformation which translates the stack based bytecode of Java to a *Register Based Bytecode* (*RBC*) by eliminating the stack. The final byte code, the Register Based Bytecode, contains register operations instead of stack operations. Each register index corresponds to the

[1]The test platform was an Intel Pentium IV with a processor of 2 GHz.

current stack height of the original bytecode. For instance, the first value on the stack is represented by `r0`. Moreover, local variables are represented with indices, too. Local variables and registers are therefore treated uniformly. Register Based Bytecode is more suitable for analysis and optimizations and it benefits from the *Abstract Bytecode Transformation* [38].

The `JNuke` framework already provides some basic optimization. The transformation to Register Based Bytecode produces many useless register assignments. Thus, many of them can be eliminated which results in so called *Optimized Register Based Bytecode*. As an example, consider Figure1.2.

```
RBC                            Optimized RBC
0:  "r5 = Get" r1              0:
1:  "r6 = Get" r2              1:
2:  "r5 = Prim" iadd r5 r6     2:  "r1 = Prim" iadd r1 r2
3:  "r1 = Get" r5              3:
```

Figure 1.2: Register byte code produced for the example statement `local1 += local2`.

1.3 Related work

Software verification can be divided into *dynamic* and *static checking*. Dynamic checking consists of verifying properties of a program at run-time. Because of the arbitrary number of possible inputs and possible schedules, it is necessary to test many inputs combined with many schedules hoping that eventually, enough combinations would be tested to uncover most faults. In contrast to that, static checkers do not run the program, but analyze properties based on the source or structure of the program.

Dynamic checkers

Dynamic checkers monitor properties of interest, such as assertions, lock sets, or memory accesses at run-time. Since the number of possible inputs is exponential in the length of the input, exhaustive testing that covers each outcome cannot be implemented. In addition to the enormous number of inputs to be considered, the outcome of concurrent programs depends on the schedule. The number of schedules is exponential in the number of threads and locks. There are approaches that use heuristics in order to reduce complexity or keep track of the program's history and deduce information about other possible outcomes.

A dynamic checker finds a fault if the currently tested input leads to this fault. For a multi-threaded program any possible schedule has to be considered, too. Thus, certain faults cannot be detected unless either the checker is exhaustive or someone has created a test case that leads to this fault. Creating test cases is far from trivial and a time-consuming and tedious task. This strongly limits dynamic checkers.

There are also advantages. A dynamic checker knows the entire and exact program state at any point of execution whereas static checkers work on an approximation of the program states. This accurate view on the program state allows to make assertions without a doubt.

There are some dynamic checkers working on multi-threaded programs which are able to detect common timing-dependent errors. Some of them are listed below:

Rivet [7] is a partial virtual machine implemented by the Software Design Group at the MIT. Equivalent to the virtual machine described in this work, the Rivet virtual machine provides a rollback mechanism. It also provides a debugger that supports execution in the reverse direction of program flow, including stepping, execution to the next breakpoint, and execution until a data value changes. There is an implementation of *ExitBlock* and *ExitBlock-RW* on top of the Rivet virtual machine. The Rivet virtual machine is written in Java. Its design focuses on modularity, extensibility, and sophisticated tool support rather than on maximizing performance. Since Rivet runs on top of another Java virtual machine, it can make use of that *lower* virtual machine's platform-specific features like garbage collection and native method libraries. The work, however, has been discontinued as the security policy of current virtual machines does not allow to replace classes from the standard library.

MaC [24] MaC (Monitoring and Checking) is a framework combining low-level monitoring with high-level requirement specification. MaC instruments and verifies single-threaded as well as multi-threaded programs. It is being developed at the University of Pennsylvania.

Verisoft [45, 14], by Patrice Godefroid from Lucent Technologies, systematically explores the state space including thread interleavings of a program. Verisoft makes use of state-space pruning methods and performs some static analysis to provide information needed by these pruning methods [13]. Verisoft's target programs are small C programs consisting of multiple processes. Since processes run in separate memory spaces and must explicitly declare the variables that they share, the number of global states is significantly smaller than the number of states. Since processes can only communicate through global states, Verisoft need only consider schedules that differ in their sequences of global states. Verisoft further prunes the state space by identifying schedules that lead to identical behaviours. Verisoft discovers errors in C program such as deadlocks, life-locks, assertion violations, and other properties.

Visual Threads [43] is part of the development tools of Compaq's Tru64 Unix. Visual Threads is a diagnostic tool used to analyze and refine multi-threaded applications. It can be used to debug potential thread-related logic problems, such as race conditions and deadlocks that only occur due to slight timing differences. It can also pinpoint bottlenecks and performance problems by using its rule-based analysis and statistics capabilities and visualization techniques.

JPaX (Java PathExplorer) [16, 15] is built at NASA Ames research center. The runtime analysis is based on program instrumentation or transformation, and can include monitoring of a program in its final environment. JPaX implements deadlock and data race analysis. It includes automated test-case generation as well as automated generation of assertions and properties corresponding to test cases.

Static checkers

In contrast to dynamic checkers, static checkers do not execute the program. They examine and check programs against defined properties working on the source or an abstract model thereof. Three techniques have mainly emerged: *model checking*, *abstract interpretation*, and *theorem proving*. Model checking explores the full state space. Abstract interpretation examines the abstract model which represents the *control flow* and *data flow* of a program. Theorem proving translates the program into logic formulas where a prover works on these logic formulas.

Static checkers have advantages over dynamic checkers as they work on a more abstract representation of the target program. In particular, static checkers depend neither on the input nor on the schedule of a program. The complexity rather depends on the size of the model and the algorithms applied to this model.

The key problem is that the actual values of variables are usually known at run-time only. Indeed, static checkers have to work on an approximation of the reachable program states which sometimes leads to states that are unreachable in the real program. Since the model has a limited knowledge of the program state, the model is not accurate at any execution point, resulting in spurious errors.

A range of static checkers working on multi-threaded programs are listed below:

ESC/Java [10] is a theorem prover that detects common programming errors at compile-time. It has been developed by Digital Equipment Inc. (now part of Hewlett Packard). ESC/Java statically checks a program for `null` reference errors, array bounds errors, potentially incorrect type casts and race conditions. The source code needs annotations in an annotation language.

Spin [37] is a model checker serving among other things as back-end for other static checkers, such as Bandera or JPF1. The tool was developed at Bell Labs in the original Unix group of the Computing Sciences Research Center. The software has been available freely since 1991.

Bandera [4] comes from the Kansas State University. It tries to simplify the program by omitting statements that are not relevant to the later analysis and reducing the state space of variables. This technique is called *slicing* [9]. It can drastically cut down the complexity of the model. The model created is processed by Spin.

JPF [23] is the second Java Model Checker developed by the Automated Software Engineering group at NASA Ames. JPF works on Java bytecode and discovers assertion violations and deadlocks statically. JPF1 [17] used a translation from Java to PROMELA, SPIN's input language, in order to do model checking with the SPIN model checker [37].

Jlint [22, 21] has been developed at the Moscow State University. Jlint detects certain deadlocks, race conditions and a few other faults. It performs a global control flow and a local data flow analysis that allows to check many properties in Java bytecode.

Java virtual machines

There are various Java virtual machines available. The most popular virtual machines come from Sun [41] and IBM [18]. Both virtual machines can be downloaded for free. They are closed-source, however, which makes it very difficult to implement a milestone/rollback mechanism on top of them. There are also several open source virtual machine projects such as *Kaffe* [29], *LaTTe* [30], *kissme* [25], *SableVM* [31], *Japhar* [28], or *CACAO* [27]. A rollback/milestone mechanism could be implemented on top of these virtual machines. It mainly depends on whether the design of the virtual machine easily allows this. Since these virtual machines do not profit of advantages of the register based byte code transformation, our group decided to write a custom virtual machine.

JNuke already provides a basic interpreter for register based byte code (JNuke Interpreter). Development, however, has been discontinued. The JNuke Interpreter is too slow and many things are not standards compliant. It does not support threading and Java strings. Moreover, things like interface calls, cast checks, exception and many other things do not work properly. Though the prime reason why development of the JNuke Interpreter has been discontinued, was that a milestone/rollback mechanism was to hard to add due to the non-modular design of the interpreter.

1.4 Overview

The next chapter introduces the theory of systematic testing and presents the two algorithms *ExitBlock* and *ExitBlock-RW*. Chapter 3 focuses on the design and the interfaces of the virtual machine. Chapter 4 explains the implementation of the rollback mechanism. Chapter 5 then shows experiments demonstrating the capability of the virtual machine and the implementation of *ExitBlock* and *ExitBlock-RW*. Chapter 6 presents future work and finally Chapter 7 concludes.

Chapter 2

Algorithms for Systematic Testing

This chapter describes the algorithms which execute a program or parts thereof multiple times, enumerating possible schedules in order to cover all program behaviours on a single input. The theory of systematic testing has been taken from Bruening's master thesis [7]. Since many things are unclear or not mentioned in Bruening's master thesis, this chapter explains the theory more accurately. In particular, the description of *ExitBlock* and *ExitBlock-RW* is vague. It was not possible to implement these algorithms without making own assumptions.

In [7], Bruening says that these algorithms guarantee to find all possible assertion violations in a program if the tested program meets some requirements[1]. Assertions in a program can detect any program condition. The algorithms for systematic testing will be called *systematic scheduler*. Thus, the systematic scheduler guarantee to enumerate all possible behaviours.

This chapter is organized as follows. Section 2.1 describes the requirements that a target program has to meet. Section 2.2 introduces the idea of enumerating all meaningful schedules and Section 2.3 describes the first testing algorithm called *ExitBlock*. Section 2.4 modifies *ExitBlock* in order to reduce the number of schedules whereby all program behaviours are still covered. This algorithm is called *ExitBlock-RW*. Finally, Section 2.5 shows detecting deadlocks using *ExitBlock* and *ExitBlock-RW*.

2.1 Target program requirements

The algorithms for systematic testing presented in this work require that a target program meets three criteria:

[1]However, Scott Stoller criticises in [39] this statement. He says that Bruening's proof is incomplete, because the proof implicitly assumes that all accesses satisfy Mutex Locking Discipline (MLD) [36, 39]. The MLD allows objects to be initialized without locking. Initialization is assumed to be completed before the object becomes shared. According Scott Stoller, Bruening does not prove that *ExitBlock* is guaranteed to find a violation of MLD for systems that violate MLD. Even if violations of MLD are manifested as assertion violations, the (incomplete) proof does not imply that *ExitBlock* finds all violations of MLD, because that proof presupposes that the system satisfies MLD.

Mutual-exclusion locking discipline A target program has to follow a mutual-exclusion locking discipline which dictates that each shared variable is associated to at least one mutual-exclusion lock. Furthermore, this lock or these locks are always held by the current thread which accesses that variable. The Eraser algorithm by Savage et al. [36] can be used to verify that a program follows this discipline. It is even possible to run Eraser in parallel to the systematic scheduler.

By limiting the scope to those programs that follows this locking discipline, some valid programs are ruled out for the algorithms described in this chapter. For instance, `wait` and `notify` can be used to build a *barrier*, which is a point that each thread has to reach before any thread can cross this barrier. Among other things, barriers can be used to protect shared variables. Barriers, however, are hardly used for these purposes. Programmers tend to use a simple mutual-exclusion locking discipline.

Thread-safe finalizers Finalizer methods are mainly used to release system resources like open file descriptors. As such, finalizers do not affect the rest of the program. Since finalizers are called by the garbage collector when instances of those classes are reclaimed, finalizers can be invoked at any time and in any order [26]. A systematic testing algorithm would need to execute every possible schedule of finalizers with the rest of the program in order to find possible assertion violations. This can be a very large number of schedules to run. The systematic scheduler thus assume that the order of finalizers has no bearing on weather assertions will be violated or whether a deadlock can occur. This reduces the number of explored schedules.

In order to avoid timing-dependent errors in finalizers, the target program needs to meet the following criteria: A finalizer should only access fields of the current instance and, in order to prevent deadlocks, a finalizer should not contain nested synchronized sections or perform `wait` or `notify`. As a result, the garbage collector can be completely ignored, as a garbage collector cannot affect assertions under this assumption.

As a matter of fact, finalizers rarely occur in Java applications. Consider Table 2.1 which illustrates that the Java Foundation Classes (JFC) of Sun's JDK 1.3 contain only 26 finalizers in about 4'000 classes. Static analysis can ensure the thread safety of finalizers.

Package	Number of finalizers
java/awt	10
java/io	2
java/lang	2
java/net	4
java/util	4
javax/swing	4
TOTAL	**26**

Table 2.1: Number of finalizers in the foundation classes of Sun's JDK 1.3

Terminating threads The systematic scheduler requires that each thread terminates. This is due to the fact that these algorithms explore the tree of schedules in a depth-first search executing a schedule from start to finish. If a thread run for an infinite time, no other schedule would be explored anymore. There are two approaches to avoid this: either the user modifies the target program (for instance, changing an infinite loop to a loop with a finite number of iterations) or the testing algorithm limits the number of byte codes for each thread, terminating threads exhausting this limit.

The systematic scheduler discovers deadlocks and assertions violations in a target program; data races are not discovered. However, Eraser [36] which discovers data races can be applied to a target program to ensure that the mutual-exclusion locking discipline is met. Eraser should be either run in parallel or prior to the systematic testing algorithms.

2.2 Overview

The order of two instructions in different threads can only affect the behaviour of the program if both instructions perform read or write accesses on the same field. As postulated before, the target program follows a mutual-exclusion locking discipline where each shared field is protected by a lock. This discipline ensures that fields cannot be accessed outside of a body of a `synchronized` statement. Multiple accesses to a shared field at the same time are strictly serialized. This means that only one thread is allowed to access a field at a time. Thus, in order to cover all possible behaviours of a program it suffices to enumerate possible orders of synchronized regions.

For these purposes a program is divided into *atomic blocks* based on blocks of code being protected by synchronized regions. An atomic block consists of code in-between the ends of synchronized blocks (so called *lock exit*). Thread creations and terminations also define borders of *atomic blocks*. Atomic blocks of a program may change at run-time due to varying data flow. Atomic blocks are therefore enumerated during execution using depth-first search.

The testing algorithm performs a depth-first search on a program executing one complete schedule of the target program first. After this, execution reverts from the end of the schedule to the last atomic block boundary. At this point a new branch is created choosing another thread to run. This schedule is also executed until program completion. As long as there is still a thread that has not been chosen yet, execution reverts to the last atomic block boundary. Otherwise, a further rollback to the previous atomic block boundary is performed. The same procedure is repeated until any thread at each atomic block boundary has been selected for execution once. At the end, the depth-first search results in a tree of schedules.

T1:	T2:	T3:
1: synchronized (x) {}	2: synchronized (x) {}	3: synchronized (x) {}

Figure 2.1: Three threads each containing a single synchronized region.

Figure 2.2 illustrates a sample tree for three threads shown in Figure 2.1. Arrows show the execution flow. A horizontal line indicates a new branch. Six different schedules are explored where each of the three atomic blocks was selected in any combination with the other two blocks. The depth-first search finds the following schedules from left to right: {1,2,3}, {1,3,2}, {2,3,1}, {2,1,3}, {3,1,2}, and {3,2,1}. This covers any possible schedule and, thus, any behaviour of the program is covered.

The previous program was a simple example where each thread consists of exactly one atomic block at whose end the lock is released. Thus, each thread is able to obtain the lock at any possible schedule. This example does not illustrate how the depth-first

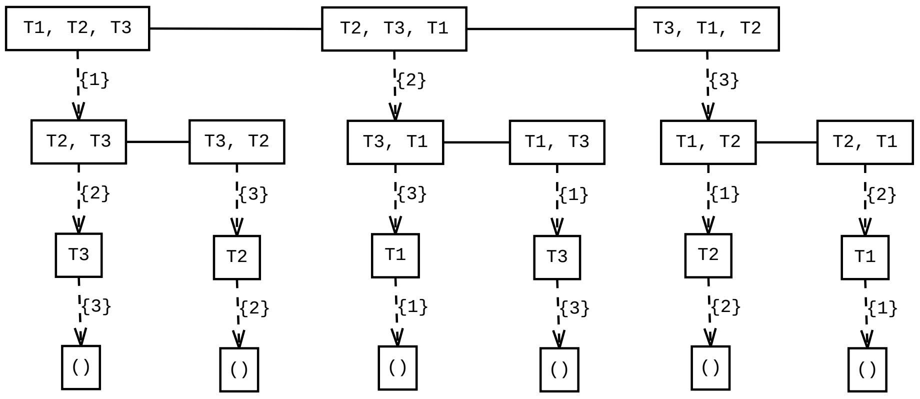

Figure 2.2: Tree of schedules for the three thread of Figure 2.1.

search reacts to a situation where a lock cannot be obtained. The program shown in Figure 2.3 whose tree is presented in Figure 2.4 illustrates such an example. There are two schedules whose execution is blocked as a lock cannot be obtained: {1,2,3,4,5,7} and {1,2,7,8,9,3}. Apparently, the execution cannot be continued and there are still schedules to explore. Therefore the algorithm aborts this path and backs up to the last atomic block boundary where another thread is scheduled instead.

```
T0:                          T1:
1: t1 = new LockAB (A, B);   7:  synchronized (A) {
2: t1.start();               8:    synchronized(B) {
3: synchronized (B) {        9:    }
4:   synchronized (A) {      10: }
5:   }
6: }
```

Figure 2.3: Two Threads, both containing nested synchronized regions. We divide `Thread 0` into atomic blocks {1, 2}, {3, 4, 5}, and {6}. `Thread 1` is divided into atomic blocks {7, 8, 9} and {10}

Since the depth-first search immediately aborts any blocked path, it cannot run into a deadlock. It is however possible to perform lock-cycle deadlock detection when a lock could not be obtained. This is exactly the approach followed by the lock-cycle deadlock detection described in Section 2.5.1. This algorithm is notified when a lock could not be obtained. Then deadlock detection takes place deciding whether the current incident is a deadlock or not. In the program from Figure 2.3 two paths are aborted where both paths would be discovered as deadlock by a lock cycle detection analysis.

The *ExitBlock* algorithm, presented in the next section, applies the depth-first search. *ExitBlock* finds each possible schedule of atomic blocks. The next section will also show that exhaustive testing is applicable for small programs only, as *ExitBlock* interleaves any atomic block with each other, regardless of whether the order of two atomic blocks is relevant for the behaviour. Due to the poor scalability of *ExitBlock* the branching behaviour is modified such that only atomic blocks with data dependencies are interleaved. This modified algorithm called *ExitBlock-RW* is described in Section 2.4.

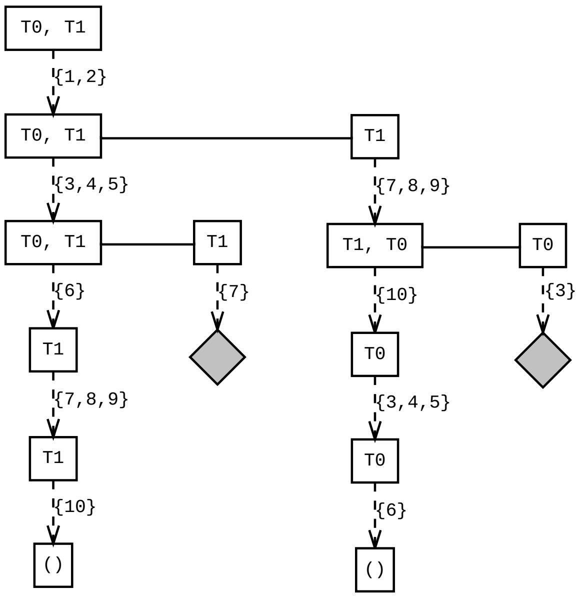

Figure 2.4: Tree of schedules explored by *ExitBlock* for the threads in Figure 2.3. Threads in a rectangle are enabled. Arrows indicate the execution of the sections of code on their right. A gray box means that the execution is blocked because a lock could not be obtained.

2.3 The ExitBlock algorithm

The *ExitBlock* algorithm is implemented as a scheduler placed on top of the virtual machine, controlling execution flow and monitoring the current state. *ExitBlock* initializes the runtime environment, installs listeners at the virtual machine and executes the target program. When the target program is started *ExitBlock* is triggered on certain events as shown at the flow chart in Figure 2.5 on the following page. The according handler methods are considered below.

Handler for lock exits

When a lock exit occurs, *ExitBlock* is notified by the lock manager. *ExitBlock* handles this as shown in Algorithm 1. First, all enabled threads are collected. Then a milestone is created. A milestone saves the whole state of the virtual machine including states of objects, locks, wait sets, threads and the enabled set. Finally, the milestone is pushed on a stack.

The enabled set contains threads that are ready to run and have not been scheduled yet from the current milestone (since the current thread continues running, it is not member of the enabled set). This set of enabled threads is used by the rollback mechanism. On a rollback the state of the whole virtual machine is restored and a new branch is created scheduling one thread from the enabled set.

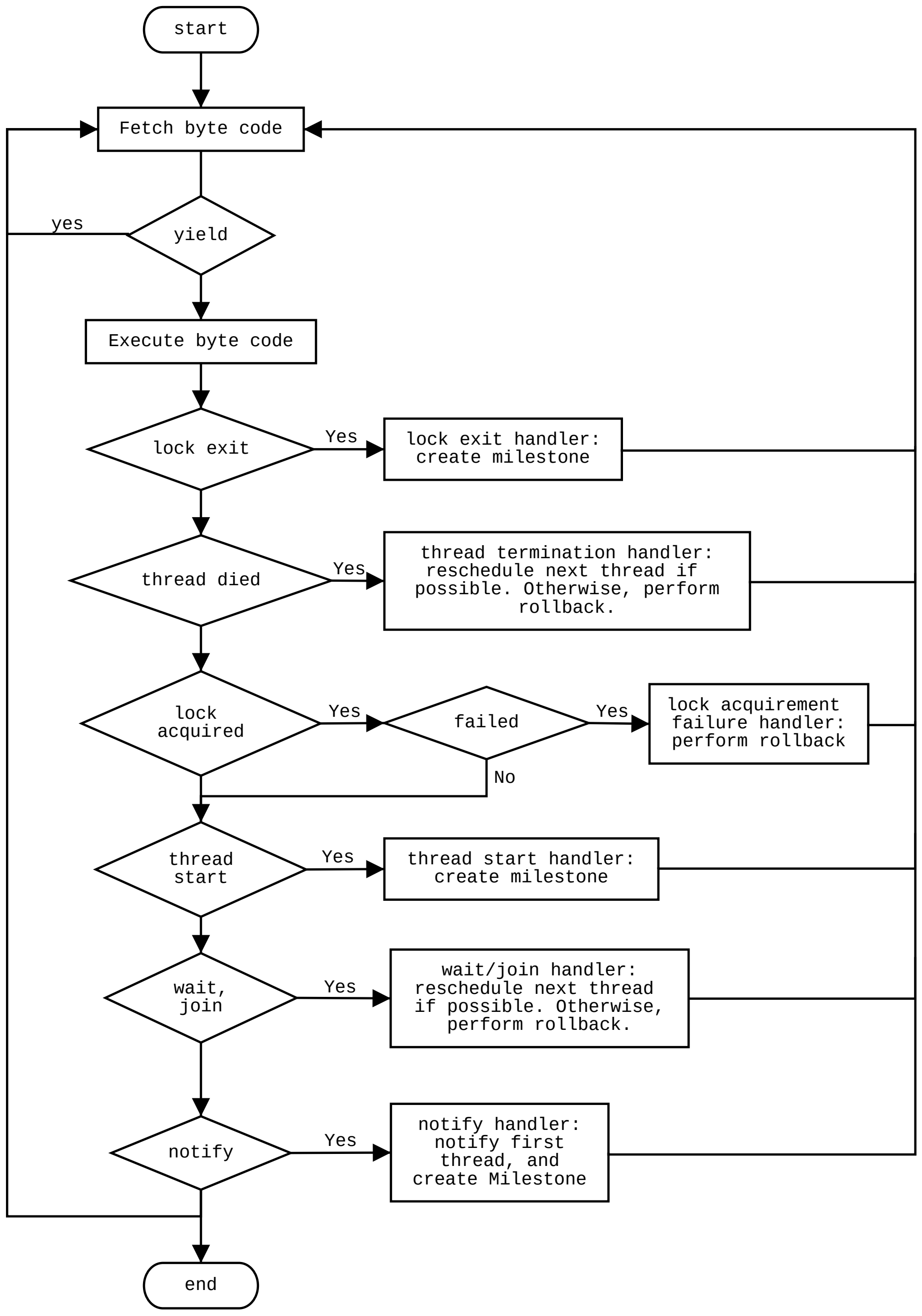
start
Fetch byte code
yes
yield
Execute byte code
lock exit
Yes
lock exit handler: create milestone
thread died
Yes
thread termination handler: reschedule next thread if possible. Otherwise, perform rollback.
lock acquired
Yes
failed
Yes
lock acquirement failure handler: perform rollback
No
thread start
Yes
thread start handler: create milestone
wait, join
Yes
wait/join handler: reschedule next thread if possible. Otherwise, perform rollback.
notify
Yes
notify handler: notify first thread, and create Milestone
end

Figure 2.5: Flow chart showing event handling for *ExitBlock*

Algorithm 1 Handler triggered on a lock exit

```
handleLockExit() {
  enabled_set = collectEnabledThreads();
  enabled_set = enabled_set \ cur_thread;
  if (number_enabled_threads >= 1) /* avoids empty milestones */
    setMilestone( enabled_set );
}
```

Handler for thread termination

When the current thread terminates, an enabled thread to be scheduled next must be selected (see Algorithm 2). If there is no enabled thread left, the end of the current schedule is reached; all threads have terminated. Thus, a rollback is performed in order to explore the next schedule in a new branch.

Algorithm 2 Handler triggered on death of current thread

```
milestone = getCurrentMilestone();
enabled_set = enable_set \ {cur_thread};
next_thread = reschedule();
if (next_thread == NULL )
  /** rollback continues down the stack,
      selects a next thread there. Returns
      NULL, if stack becomes empty.*/
  next_thread = rollback();
if (next_thread == NULL )
  /** empty stack of milestones */
  terminate();
else
  switchThread (next_thread);
```

The rollback mechanism (illustrated in Algorithm 3) reverts to the last milestone, restoring the state of the virtual machine and trying to select a new thread from the enabled set of the last milestone. If all threads of the enabled set have already been scheduled from the last milestone (which is indicated by an empty enabled set), the last milestone is removed and the previous milestone is retrieved from the stack. The thread election starts anew. This procedure stops if a milestone was found that still has enabled threads to schedule (then a new branch is created) or if the stack of milestones becomes empty (*ExitBlock* terminates).

Handler for lock acquirement failure

The lock manager notifies *ExitBlock* when a lock could not be obtained by the current thread. In consequence of this *ExitBlock* aborts the current branch and performs a rollback as shown in Algorithm 4.

Algorithm 3 Implementation of `rollback`

```
thread rollback() {
  rollbackVM();

  /** elect thread */
  milestone = getCurrentMilestone();
  next_thread = first( milestone->enabled_set );
  milestone->enabled_set = milestone->enabled_set \ next_thread;
  if (next_thread == NULL)
  {
    /** no further branch from this
        milestone. So remove this milestone
        and rollback again, continuous down
        the stack.*/
    removeMilestone( milestone );
    next_thread = rollback();
  }

  return next_thread;
}
```

Algorithm 4 Handler triggered when a lock could not be obtained

```
next_thread = rollback();
switchThread (next_thread);
```

Handler for thread start

The situation when a thread was started through `java/lang/Thread.start()` is handled like a lock exit as previously presented in Algorithm 1. A milestone is created where the enabled set contains, among other things, the newly started thread, which is scheduled once when *ExitBlock* reverts to this position.

Consider the program in Figure 2.6 and the according tree of schedules in Figure 2.7. The sample program contains a condition deadlock which occurs when `notify` is performed prior to `wait`. The deadlock occurs in the schedule (1,5,6,7,2) which is discovered only if a milestone is created after line 1. Otherwise, *ExitBlock* explores only schedule (1,2,3,4,5,6,7) and `wait` and `notify` are executed in the right order. The deadlock does not occur. Thus, *ExitBlock* always creates a milestone when a thread has been started which ensures behaviour-complete testing.

```
T0:
1:  t1.start();
2:  synchronized(a) {
3:    a.wait();
4:  }
```

```
T1:
5:  synchronized(a) {
6:    a.notify();
7:  }
```

Figure 2.6: Two threads with a condition deadlock occurring when `notify` is performed prior to `wait`

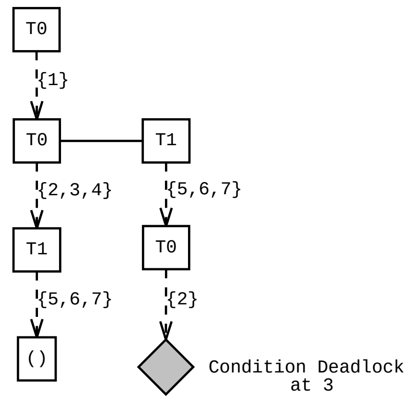

Figure 2.7: Tree of schedules produced by the program in Figure 2.6.

Handler for invocation of wait or join

So far thread communication other than with shared variables protected by synchronized regions has been ignored. In particular, `wait` and `notify` have been omitted. The base class `java/lang/Object` provides the three methods `wait`, `notify`, and `notifyAll` to any sub-class. Method `wait` allows the current thread to wait on an object which sets the current thread sleeping until another thread performs either `notify` (awakening a random thread waiting on the object) or `notifyAll` (awakening all threads waiting on the object). Another thread operation that needs to be considered is `join`. The current thread can join another thread which sets the current thread sleeping until the joined thread has been terminated. Method `join` and `wait` have in common that the current thread is disabled such that the scheduler has to find another thread to schedule. Thus, `join` and `wait` are treated the same way as illustrated in Algorithm 5. First it is tried to find a thread that is still enabled. If this succeeds this thread is scheduled. Otherwise, *ExitBlock* has detected a condition deadlock where one thread is waiting on a notification for an infinite time as no other thread is enabled. Since the current schedule is blocked, a rollback is performed. Method `rollback` reverts to the last milestone, creating a branch and returns a thread scheduled next. If the depth-first search has been completed which is indicated by an empty stack of milestones, `rollback` returns `null`.

Algorithm 5 Handler triggered for *join* and *wait*

```
next_thread = reschedule();
if (next_thread == NULL )
{
  /** condition deadlock detected !!!*/
  next_thread = rollback();
}
if (next_thread == NULL )
  terminate_depth_search();
else
  switchThread(next_thread);
```

Handler for invocation of notify

In contrast to `notifyAll`, which awakens any thread waiting on a certain object, `notify` does the same for one arbitrary thread. *ExitBlock* needs to explore the schedules resulting from each possible thread being woken up by a `notify`, so it needs to make new branches for each. The handler for `notify` is shown in Algorithm 6. The first thread in the wait set is notified which is the usual way to handle `notify`. Furthermore, *ExitBlock* creates a milestone that additionally contains a list of threads that can also be notified (called *notify set*). On each rollback the virtual machine reverts to the point where `notify` was performed and notifies another thread from the notify set. *ExitBlock* reverts to this milestone as often until each thread initially waiting on the notified object has been notified. Finally, *ExitBlock* has created a branch for each thread waiting on the notified object.

The rollback mechanism needs some additions as shown in Algorithm 7. They are printed in bold. When *ExitBlock* reverts to a milestone that belongs to an invocation of `notify`, the enabled set is restored and one thread of the notify set is notified at a time. *ExitBlock* creates for each thread in the notify set one branch.

Handler for invocation of yield

Method `yield` causes the currently executing thread to temporarily pause and allow other threads to execute. Since *ExitBlock* allows preemption on lock exits only, invocations of `yield` are ignored.

Algorithm 6 Handler triggered on invocation of notify

```
/** retrieve the wait set of the object on which notify was
    performed */
wait_set = getWaitSet( object );
/** notify first thread which removes notified thread
    from wait set */
notifyNext (wait_set);
/** create a milestone if there are still threads
    waiting on this object that could also be notified */
if (count (wait_set) > 0)
  {
    handleLockExit() /* see Algorithm 1 */

    milestone = getCurrentMilestone();
    /** threads from the wait set needs to be notified later. */
    milestone->notify_set = copy(wait_set);
    /** remember enabled set */
    milestone->saved_enabled_set = copy(milestone->enabled_set);
  }
```

Algorithm 7 Implementation of `rollback` with additions for notify handling.

```
thread rollback() {
  milestone = getCurrentMilestone();
  enabled_set = getEnabledSet( milestone );
  next_thread = pop( enabled_set );

  if (isNotifyMilestone( milestone ))
  {
    milestone->enabled_set =
      copy (milestone->saved_enabled_set);
    thread = first(milestone->notify_set);
    milestone->notify_set = milestone->notify_set \ {thread};
    if (thread)
     notify( thread );
      next_thread = cur_thread;
    else
      next_thread = NULL;
  }
  if (next_thread == NULL)
  {
    removeMilestone( milestone );
    next_thread = rollback();
  }

  return next_thread;
}
```

Handler for stop, resume, and suspend

There is no handler implemented for `stop`, `resume`, and `suspend` yet due to lack of time. Furthermore, `stop`, `resume`, and `suspend` have been deprecated. Method `suspend` and `resume` are inherently deadlock-prone. If the target thread holds a lock on the monitor protecting a critical system resource when it is suspended, no thread can access this resource until the target thread is resumed. If the thread that would resume the target thread attempts to lock this monitor prior to calling resume, a deadlock results.

Method `stop` is inherently unsafe. Stopping a thread causes it to unlock all the monitors that it has locked. If any of the objects previously protected by these monitors were in an inconsistent state, other threads may now view these objects in an inconsistent state.

2.3.1 Number of schedules executed by ExitBlock

Assume a program with k threads each with n lock exits, the total number of lock exits is $k \cdot n$. The n lock exits of the first thread can be placed $\binom{kn}{n}$ times in between all of the other lock exits. The second thread can place its atomic blocks $\binom{(k-1)n}{n}$ times in between the others. This continues as follows: $\binom{kn}{n} \cdot \binom{(k-1)\cdot}{n} \cdot \binom{(k-2)n}{n} \cdots \binom{n}{n}$. Applying Stirling's approximation results in a product of terms exponential in both the number of threads and the number of locks uses by each thread [7].

The exponential growth of the number of schedules limits the applicability to small and not representative concurrent programs. *ExitBlock* explores too many schedules in a real concurrent program. It does not terminate in a reasonable amount of time. *ExitBlock* could be restricted to explore portions of a program. The user would define critical sections to explore. This feature is not implemented yet. It is, however, discussed in Chapter 6.

2.4 The ExitBlock-RW algorithm

The *ExitBlock* algorithm executes all schedules interleaving each atomic block with each other. However, not all schedules need to be executed in order to find all possible assertion violations. If two atomic blocks have no data dependencies between them, the order of their execution has no effect on a potential assertion violation. A data dependency between two atomic blocks exists if one atomic block writes a field whereas the other atomic block either reads or writes the same field. Two atomic blocks reading the same field or accessing different fields do not have data dependencies.

ExitBlock-RW records a read/write log for each thread. The log for a thread n is defined as pair of two sets (r_n, w_n). A log k intersects with another log l iff $(w_k \cap w_l \neq \emptyset \vee w_k \cap r_l \neq \emptyset \vee r_k \cap w_l \neq \emptyset)$.

A thread that has been scheduled in a branch is disabled for further branches. *ExitBlock*-RW re-enables a disabled thread when the currently executing thread's log intersects with the disabled thread's log. If no such intersection occurs, then none of the later threads interacts with the disabled thread and there is no reason to execute schedules in which the disabled thread follows them. As a result, disabled threads which are not re-enabled due to lack of data dependencies decrease the number of branches and cause aborts of branches prior to maturity.

As an example, consider again the sample program shown in Figure 2.3, and the according tree explored by *ExitBlock-RW* in Figure 2.8. Consider path {1,2,7,8,9} which leads to a milestone where no branch is created in contrast to the tree explored by *ExitBlock*. `T0` whose log consists of entries from {3,4,5} does not intersect with the log of the current Thread `T1` whose log consists of entries from {7,8,9}. In consequence `T0` is not re-enabled and no branch is created as the order of the two atomic blocks {3,4,5} and {7,8,9} is not relevant. Consider also path {1,2,7,8,9,10} which is aborted since `T0` still cannot be re-enabled. The log of `T0` tracked during execution of {3,4,5} does not intersect with the current thread's log. As a result, {3,4,5,6} is not executed anymore as this sequence was executed in the path {1,2,3,4,5,6,7,8,9,10} where the thread interleaving is not relevant due to lack of data dependencies.

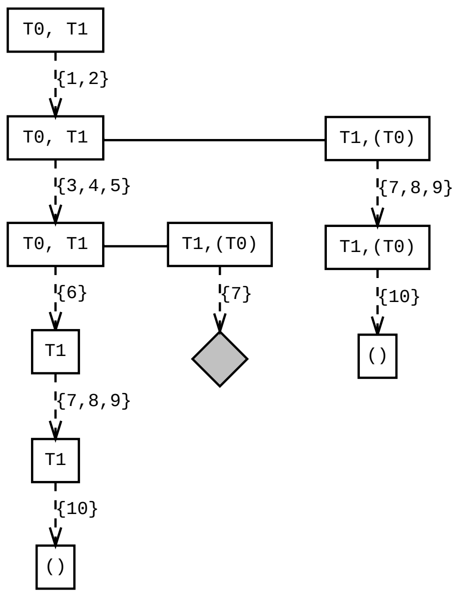

Figure 2.8: Tree of schedules explored by *ExitBlock-RW* for the thread in Figure 2.3. Threads in parentheses are disabled. Compared to the tree for *ExitBlock* shown in Figure 2.4 this tree does not bother finishing the schedule {1, 2, 7, 8, 9, 10, ...} and does not execute at all tree {1, 2, 7, 8, 9, 3, ...}.

2.4.1 Number of schedules executed by ExitBlock-RW

The number of executed schedules depends on the number of atomic blocks with data dependencies. Assume k threads each obtaining locks n times. If no atomic blocks interact, which is the best case, every thread is disabled once and never becomes re-enabled. Thus we can consider the problem creating each schedule simply that of deciding where to disable each thread. Each thread can be disabled at $n+1$ places (at n lock exits and also at the end of the program). Providing no data dependencies exist, there are $(n+1)^{k-1}$ schedules. This is polynomial in the number of locks per thread and exponential in the number of threads. It is much better than the growth of *ExitBlock* which is exponential in the number of locks per thread. Since the number of threads in a program is typically not very high, while the code each thread executes can grow, *ExitBlock-RW* achieves polynomial growth in the best case.

The worst case exists if each atomic block has data dependencies with every other one. In this case *ExitBlock-RW* has exponential growth like *ExitBlock*. Thread interactions between atomic blocks of different threads are usually kept to a minimum. So it appears likely that the typical number of paths explored by *ExitBlock-RW* is closer to the best case result than the worst case result. Experiments in Chapter 5 support this assumption.

2.5 Deadlock detection

A deadlock is a cycle of resource dependencies that leads to a state in which threads are blocked from execution. Two kinds of cycles are possible in Java programs. Either the deadlock occurs because of a cycle lock chain or threads wait on an object and other threads are blocked on locks. The first kind of deadlocks is called *lock-cycle deadlock* considered in Section 2.5.1. The second kind of deadlocks is called *condition deadlock*, discussed in Section 2.5.2.

2.5.1 Lock-Cycle deadlocks

Consider the program in Figure 2.9 and the schedules produced by *ExitBlock* for this program, shown in Figure 2.10. The deadlock occurs if `T0` holds lock `a` but not lock `b` and `T1` holds lock `b` but not lock `a`. However, *ExitBlock* does not run into this deadlock since the acquisitions of both locks in each thread are in the same atomic block.

```
T0:                              T1:
1: synchronized(a) {             5: synchronized(b) {
2:   synchronized(b) {           6:   synchronized(a) {
3:   }                           7:   }
4: }                             8: }
```

Figure 2.9: Two threads with a potential deadlock.

Nevertheless deadlocks can be detected based on the explored schedules by *ExitBlock*. The key observation is that a thread in a lock-cycle deadlock blocks when acquiring a nested lock, since it must already be holding a lock. Also, the lock that it blocks on cannot be the nested lock that another thread in the cycle is blocked on, since two threads locked on the same lock cannot be in a lock cycle. The cycle must be from a nested lock of each thread to an already held lock of another thread. For example, when the threads of Figure 2.9 deadlock, `T0` is holding its outer lock `a` and blocks on its inner lock `b` while `T1` is holding its outer lock `b` and blocks on its inner lock `a`.

These observations suggest the following approach. We track not only the current locks held by any thread but also the last lock held by each thread. Then, after we execute a synchronized region nested inside some other synchronized region, the last lock held is the lock of the inner synchronized region. When a thread cannot obtain a lock, the *reverse lock chain analyzer* looks at the last locks held by the other threads and sees what would have happened if those threads had not yet acquired their last locks. It is looking for a cycle of matching outer and inner locks. The outer locks are currently held by the threads. The inner locks are the thread's last locks held. If the

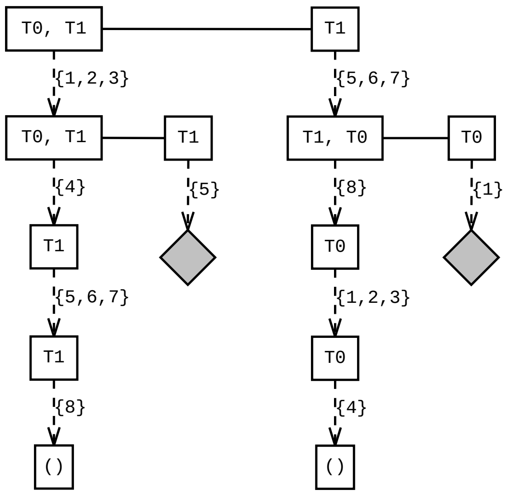

Figure 2.10: Tree of schedules explored by *ExitBlock* for the threads in Figure 2.9.

current thread cannot obtain a lock and the analyzer can follow a cycle of owner and last lock relationships back to the current thread, then a lock-cycle has been detected.

Figure 2.11 shows the situation at the time when a deadlock occurs in schedule {1,2,3,5}. The first thread holds lock `a` and has lock `b` released which becomes the last lock held. The current thread has obtained lock `b`; it blocks on lock `a`. Figure 2.12 presents how the reverse lock chain algorithm works. As mentioned, the current thread was not able to obtain lock `a` which belongs to the first thread whose last hold lock is `b`. Lock b is owned by the current thread which closes the cycle. The deadlock is discovered.

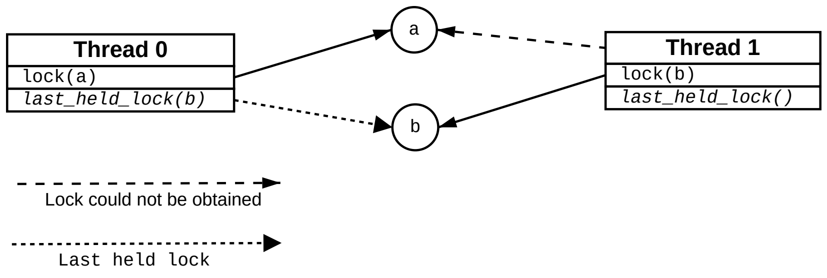

Figure 2.11: Lock dependency graph illustrating the situation when `T0` has executed {1,2,3} and `T1` cannot obtain lock a at line 6 of the program presented in Figure 2.9

The implementation of the reverse lock chain analyzer, presented in Algorithm 8, is straightforward. It does not cost much in performance, as failures to acquire locks hardly happen. The reverse lock chain analyzer is implemented as a listener of the *ExitBlock*. The lock manager notifies *ExitBlock* on failures to acquire locks, *ExitBlock* forwards these events to the reverse lock chain analyzer.

The performance can be improved by combining reverse lock chain analysis with *ExitBlock-RW* instead of *ExitBlock*. This optimization, however, comes at a cost.

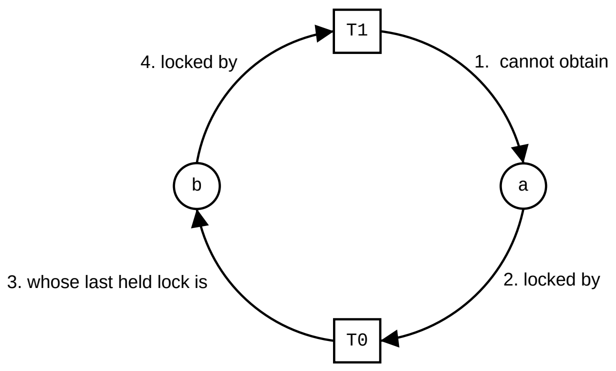

Figure 2.12: Reverse lock chain analysis for the situation illustrated in Figure 2.11.

Algorithm 8 Reverse lock chain analyzer

```
/* in: lock - The lock that could not be obtained */
while (owner_thread = JNukeLock_getOwner (lock)) {
  lock = JNukeThread_getLastHeldLock (owner_thread);
  if (getOwner (lock) == cur_thread) {
    /** Deadlock detected !!! */
    break;
  }
}
```

ExitBlock-RW in connection with reverse lock chain analysis does not always detect deadlocks in a program. *ExitBlock-RW* does not consider a lock enter to be a write to the lock object. Thus, *ExitBlock-RW* may prune paths on which deadlocks can occur.

2.5.2 Condition deadlocks

Condition deadlocks occur when threads are waiting and the rest are blocked on locks disallowing a thread to wake up the waiting thread. *ExitBlock* detects condition deadlocks by checking if there are threads waiting or blocked on locks whenever the scheduler detects that no enabled threads are left. Consider the example in Figure 2.13 showing two threads manipulating a queue. Thread `T0` holds both locks `lock` and `queue` when calling `wait`. It only releases `queue` but not `lock`. If Thread `T1` executes after `T0`, it can never obtain `lock` and so never proceed to `notify`. This results in a condition deadlock.

Condition deadlocks can also be detected by *ExitBlock-RW*. However, it does not find condition deadlock on paths that were pruned. Also, when the scheduler runs out of enabled threads and there are disabled threads, *ExitBlock-RW* cannot determine if it is a condition deadlock. Since a disabled thread, which is not scheduled due to lack of data dependencies, could break the condition deadlock, it cannot accurately be declared whether a condition deadlock has occurred.

```
T0:                          T1:
1: synchronized (lock) {     8:  synchronized (lock) {
2:   synchronized (queue) {  9:    synchronized (queue) {
3:     while(queue.empty()) {10:     queue.add(element);
4:       queue.wait();       11:     queue.notify();
5:     }                     12:   }
6:   }                       13: }
7: }
```

Figure 2.13: Queue example with an apparent condition deadlock

2.6 Summary

This chapter has introduced systematic and behaviour-complete testing by enumerating possible schedules. Behaviour-complete testing allows to discover any assertion violation in a multi-threaded program. It has shown that it is possible to build a systematic scheduler for multi-threaded Java programs by making three assumptions about the nature of the target programs: First, each thread of the target program has to terminate. Otherwise, depth-first search never terminates. Second, finalizers have to be thread-safe, as they are not tested by the systematic scheduler. And third, the target program has to follow a mutual-exclusion locking discipline. Given these assumptions, only schedules of interleaving synchronized sections needs to be considered in order to cover all behaviours of the target program.

Based on this idea, two algorithms has been presented: *ExitBlock* and *ExitBlock-RW*. *ExitBlock* enumerates all possible schedules of synchronized regions regardless of whether the order of two synchronized regions is relevant for the outcome. *ExitBlock-RW* avails this observation and uses data dependency analysis to prune the tree of schedules explored by *ExitBlock*. As a result, *ExitBlock-RW* is able to reduce the number of schedules to be considered. In the average case, the number of schedules grows polynomially in the number of locks per thread instead of exponentially which is the case for *ExitBlock*.

This chapter has also shown how deadlock detection works based on *ExitBlock* and *ExitBlock-RW*. Condition deadlocks and lock-cycle deadlocks are discovered where *ExitBlock-RW* sometimes does not discover deadlocks as it prunes the tree of schedules. However, *ExitBlock-RW* is still able to find all possible assertion violations as they depend on data dependencies.

Chapter 3

The Virtual Machine

This chapter describes how the Java Virtual Machine is implemented and how it can be used by a client. It is organized as follows: Section 3.1 appoints the objectives of the Virtual Machine. Section 3.2 gives an overview of the design. All sub-systems and the relationship between them will be shown. As a result of this section, the following sections 3.3, 3.4, 3.5, 3.6, and 3.7 give full particulars the sub-systems.

3.1 Objectives

Functional objectives

The goal was to implement a Java virtual machine (*JVM*) that is able to execute multi-threaded Java programs. Unlike other virtual machines the desired virtual machine provides well-documented interfaces that allow external tools to monitor and modify the virtual machine state at run-time. In particular, the virtual machine should allow to monitor and modify thread scheduling, locks, and field accesses. The intended virtual machine also allows external tools to provide their own thread scheduler. Thus, thread scheduling takes place at user level instead of kernel level as current JVMs do. This allows tools to achieve full control over the virtual machine and its threads.

Non-functional objectives

Almost any common JVM today comes with a *Just-in-time Compiler* (*JIT*) that translates Java byte code into native machine code of the host system. A JIT enormously boosts the execution performance. However, a JIT compiler does not come into question for this virtual machine, as full control over the virtual machine, as postulated before, cannot be achieved anymore. In spite of this, the virtual machine has to run even complex programs in a reasonable time. In particular, the execution time for algorithms which exhaustively test schedules should remain as short as possible. Although an interpreter can never reach the performance of native machine code, performance was always kept in mind during the implementation.

It is also important to keep track of the memory footprint. Although recent computers are provided with hundreds of megabytes of memory, memory footprint becomes an issue in terms of runtime verification algorithms that save the whole state of the virtual machine many times in order to revert to previous states. It is therefore important that the layout of Java objects is as compact as possible and data structures used by the Virtual Machine during execution are held as small as possible.

3.2 The design at a glance

The entire virtual machine consists of several self-contained subsystems as shown in Figure 3.1. The core subsystem is the run-time environment (described in Section 3.6). The runtime environment drives the whole execution of byte codes. It starts at the static `main` method, assumed that a class with a valid `main` method is present, and stops running when the program exits. The run-time environment delegates tasks like object locking, heap operations, and managing of wait sets to subsystems: the heap manager (class `JNukeHeapManager`) shown in section 3.3, the lock manager (class `JNukeLockManager`) shown in section 3.4, and the waitset manager (class `JNukeWaitSetManager`) presented in section 3.5.

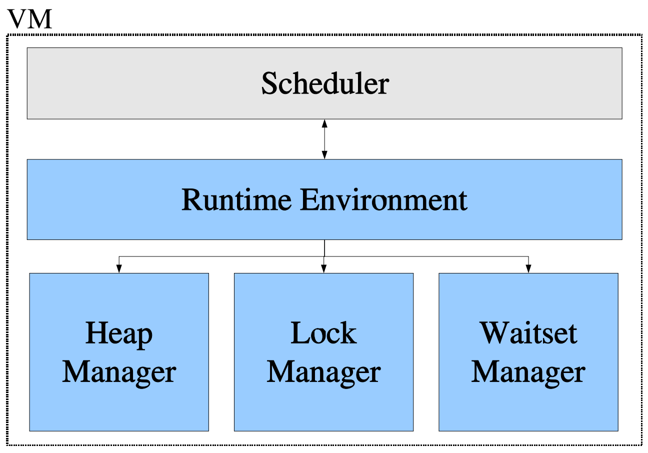

Figure 3.1: The virtual machine consists of several subsystems: a scheduler on top, that controls the execution flow, the runtime environment, a heap manager, a lock manager, and a waitset manager.

The run-time environment as such is not able to execute multi-threaded programs as it does not provide scheduling facilities. However, it provides a well-documented interface allowing to write pluggable schedulers. A scheduler acts as a listener of the run-time environment, reacting on events issued by the virtual machine. A scheduler can register itself to various events. As a result, a scheduler is able to monitor and control the virtual machine at any point of execution. Thus, a scheduler can preempt threads and control the execution order of threads.

The virtual machine currently implements a simple round-robin scheduler (class `JNukeRRScheduler`) that grants each thread a fixed time slice in a round-robin order. As an example, the *ExitBlock* and *ExitBlock-RW* algorithm provide their own schedulers, since they allow predetermined thread switches at lock exits only.

3.3 The heap manager

A Java Virtual Machine needs a heap memory management unit where all instances are managed. Unlike other programming languages, Java creates all instances on the heap. Even temporary instances with local scope are heap objects. The heap management unit provides methods for creating instances and, load and store operations. These methods are implemented in class `JNukeHeapManager,` which is the only class used by a client of the heap manager. In contrast to a usual heap manager the heap management unit does not provide any methods yet for relinquishing instances. This is because there is no garbage collector neither implemented nor planned[1].

The design goals of the heap management unit are to achieve both fast operation and small memory footprint. Fast operations mean in this context:

1. fast creation of instances,
2. fast access to fields, and
3. a fast rollback mechanism (explained in Section 4).

A small memory footprint enforces that data should be stored as compact as possible. In particular, this is important in terms of the milestone/rollback mechanism where each modification of an instance is recorded.

Java instances

Java Classes contain field declarations either describing *class variables*, which are incarnated once, or *instance variables*, which are incarnated for each instance of the class. Any Java class has exactly one *class instance* and an arbitrary number of *object instances*. The unique class instance is dedicated to the class variables and created by the virtual machine. The operator `new` applied to a class name creates an object instance. An object instance holds the instance variables for one incarnation. Java also supports arrays. *Array instances* are objects and dynamically created as object instances [26].

Instance descriptors

Any Java instance consists of two parts. The *descriptor* and the actual instance with the values of the instance. This applies to class instances, object instances, and array instances. The descriptor is either an instance of the class `JNukeInstanceDesc` (for class instances and object instances) or `JNukeArrayInstanceDesc` (for array instances).

There are exactly two separate descriptors for each Java class. The first descriptor stores names and offsets of the class variables. The second stores the same for instance variables. Since all object instances of the same class have the same variables, they all share the same descriptor. Consider the example in Figure 3.2.

[1]The reason why no garbage collector is planned is that systematic testing of programs with a garbage collector working in parallel becomes very complicated. Classes in Java may have finalizers that are called by the garbage collector when instances of those classes are reclaimed. The main problem is that the order of execution of finalizers is not determined and finalizers can be executed at any time. This means that systematic testing would need to execute every possible schedule of finalizers with the rest of the program in order to find possible assertion violations. This can be a very large number of schedules. However, most finalizers are used only to deallocate resources such as file descriptors. Usually, this kind of activity does not interact with the rest of the program and so it is a considerable approach not to execute any finalizer. They

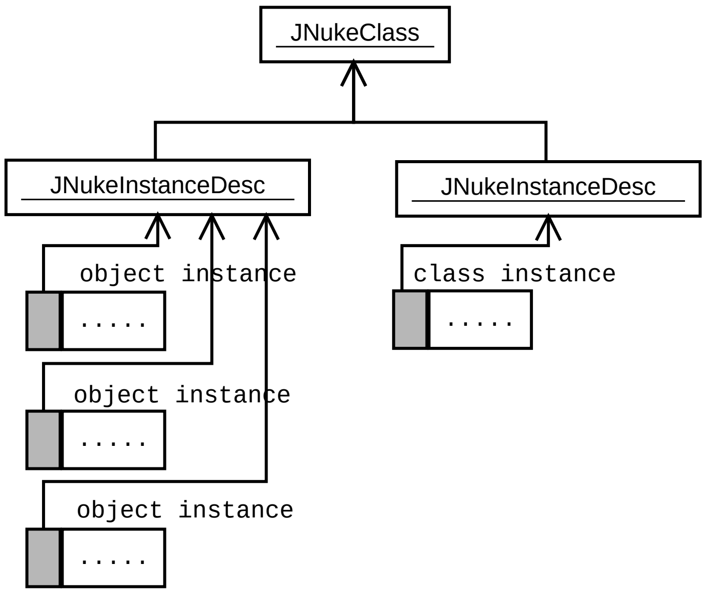

Figure 3.2: A class having two descriptors: one for the unique class instance that contains class variables and one for all object instances of this class. There are four instances present on the heap: three object instances on the left and the fourth instance, the unique class instance, on the right.

An array instance does not have named fields as opposed to class and object instances. Therefore, array instances have their own descriptors. This descriptor stores the component type and the size for one component. This suffices to calculate the offset of a component. Two arrays share the same descriptor iff they have the same component type and the same dimension (see Figure 3.3 for an example).

The separation of the actual instance and its descriptor allows to share many descriptors. This results in a small memory footprint. In particular, when a program creates many instances of the same type, the same descriptor is used each time.

The memory layout of instances

In contrast to a descriptor, the actual instance is designed as a contiguous block of memory. This approach is similar to *C structs* which comes along with several considerable advantages:

- Low memory consumption.
- Fast creation of instances. It simply consists of allocating a contiguous block of memory according the size provided by the descriptor.
- Fast field access. A `JNukeInstanceDesc` stores pairs of variable names and the according offsets. A `JNukeArrayInstanceDesc` provides the size of each

are considered thread safe. In order to ensure that no deadlocks can occur due to the timing of a finalizer, finalizers should not contain nested synchronized regions or perform `wait` or `notify` operations.

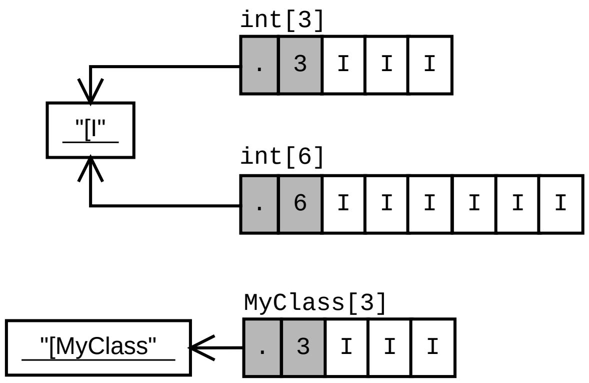

Figure 3.3: Three different array instances: Two integer arrays with different size but equal type, sharing one array descriptor and a third array with its own descriptor.

component. The offset results if the component size is multiplied by the desired index.

- Contiguous blocks of memory are ideal in terms of the milestone/rollback mechanism. This mechanism uses tracking of heap modifications which is easy to implement for contiguous blocks of memory.

Each contiguous block has a header as shown in Figure 3.4. The first field of the header points to the instance descriptor. This descriptor is either an instance of `JNukeInstanceDesc` or `JNukeArrayInstanceDesc` which allows to identify each instance on the heap.

```
struct JNukeInstanceHeader {
  JNukeObj *descriptor;
  JNukeObj *lock;
  JNukeInt4 arrayLength; /* for arrays only */
  JNukeObj *waitset;
};
```

Figure 3.4: The header of an instance consists of a descriptor pointing to either an instance of `JNukeInstanceDesc` or `JNukeArrayInstanceDesc`, a reference to a lock, the array length (used for arrays only), and a reference to a list containing threads currently waiting on that object.

The alignment of fields or components depends heavily on the underlying architecture, the same way as C structs do. The i386 architecture is a 32-bit architecture where each 4-byte field is aligned to 4-byte boundaries. A field that is smaller than 4 bytes is also aligned to 4-byte boundaries. 8-byte fields such as double and long fields can be aligned either to 4-byte or 8-byte boundaries. The Intel architecture accepts both. However, an 8-byte alignment is preferred [19]. The same alignment is used for the Power-PC architecture. On 64-bit platforms such as Alpha or SPARC v9 a strict 8-byte alignment was selected in spite of the fact that this wastes memory for fields smaller than 8 bytes. At the moment the algorithm that calculates offsets is not smart enough to compact two 4 byte fields into one 8-byte field. As the SPARC architecture is not able to read 8-byte values aligned to 4-byte addresses, even SPARC v8, a 32-bit architecture, uses a strict 8-byte alignment. Figure 3.6 shows an example where offsets are calculated for the sample class shown in Figure 3.5.

```
public class MyClass {
  int a;
  int b;
  long c;
}
```

Figure 3.5: An example class with a couple of fields

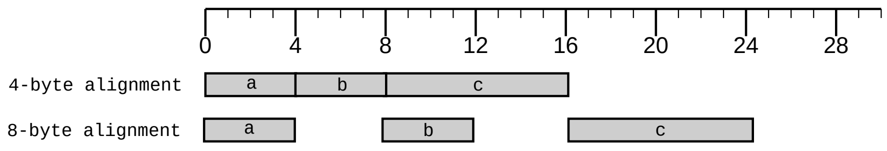

Figure 3.6: The calculated offsets of the class from Listing 3.5. The 8-byte alignment wastes 16 bytes in this case although a smart algorithm could place the fields such that memory usage is the same as for architectures with 4-byte alignment.

Multi-dimensional arrays

As opposed to array, object or class instances, multi-dimensional arrays are not stored at a single contiguous block of memory. Some programming languages such as C implement multi-dimensional arrays as one linear memory block. The offset of an element is calculated with a few multiplications and additions. The overhead of the additional multiplications and additions is negligible. Thus, the overall performance is nearly as fast as for arrays. The complexity is therefore $O(1)$. Java, however, presumes that a multi-dimensional array consists of a composition of several usual array instances [26]. Consider the example in Figure 3.7 which shows a multi-dimensional array `int[3][2][3]`. The first level consists of one array with three components. The second level consists of three arrays with two components each. Finally, the third level has six arrays with three components each. In order to read or write an element at the third level, the virtual machine needs to walk top down the tree. In consequence of that, an array access is $O(n)$ (n is the number of array levels). It is almost impossible to allocate a contiguous block of memory for multi-dimensional arrays, as Java allows to create arrays dynamically at run-time. An array can be incomplete and may change its size during execution, as shown in Figure 3.8.

Listeners and events

The heap manager provides an interface to allow a client to create, read, and write instances. It hides internal details from the client. Communication is strictly unidirectional: the client calls the heap manager. However, the heap manager provides another mechanism that implements notification callbacks similar to the Java event model (also

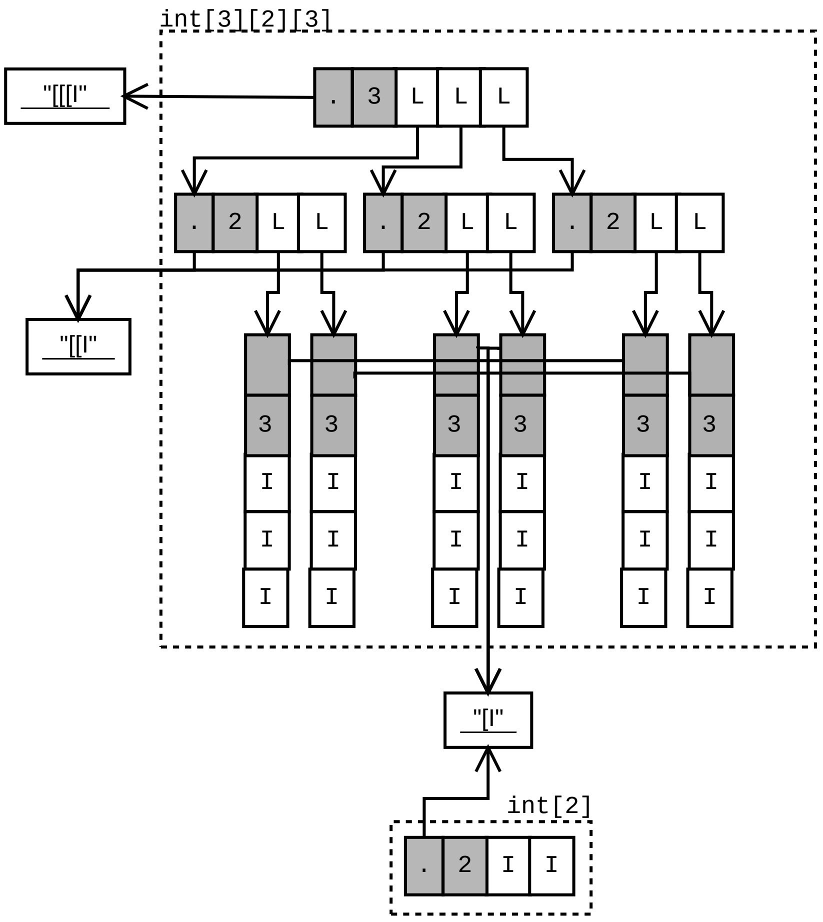

Figure 3.7: Two arrays, `int[3][2][3]` and `int[2]`. Each array, even the sub-level arrays, have an array descriptor. Sub-level arrays of the same level share the same array descriptor. Each type of a sub-level array is the dereferenced type of the super-level array.

```
public class MyArrays {
  static public void main(String[] args) {
    int[][] i = new int[4][];
    i[3] = new int[10];
    int[][] a = new int[4][5];
    a[3] = new int[10];
  }
}
```

Figure 3.8: A Java example where two arrays are changed at run-time.

called *observer pattern* [12]). This enables a client to get notified if an object on the heap has been read or written, which is often of interest for verification tools.

There is a struct called `JNukeHeapManagerActionEvent` used as an event (detailed in Section B.1). The heap manager provides two registration methods, `addReadAccessListener` and `addWriteAccessListener`, shown in Figure 3.9. The first registration method is used if a client wants to be notified when an object is read. The second one does the same, however it is used in order to listen to write accesses of objects. Each registration method allows to add a listener function pointer that is used as a callback when an according event occurs[2].

```
void JNukeHeapManager_addReadAccessListener (
  JNukeObj * this,
  JNukeObj * listenerObj,
  JNukeHeapManagerActionListener (l));
void JNukeHeapManager_addWriteAccessListener (
  JNukeObj * this,
  JNukeObj * listenerObj,
  JNukeHeapManagerActionListener (l));
```

Figure 3.9: Methods addReadAccessListener and addWriteAccessListener register a listener. A listener consists of a listener object and a listener function pointer. If an event occurs, the listener function is called. It takes two arguments: the listener object used as this pointer and the event.

3.4 The lock manager

The lock manager (class `JNukeLockManager`) is a subsystem of the runtime environment. The Java programming language does not provide a way to perform separate *lock* and *unlock* operations; instead, they are implicitly performed by high-level constructs that always pair such operations correctly. There is a lock associated with every object. The `synchronized` statement attempts to perform a lock operation on the object and does not proceed further until the lock operation has successfully completed. After the lock operation has been performed, the body of the `synchronized` statement is executed. When the body has been completed an unlock operation is performed on that object. The same applies to synchronized methods [26].

The lock manager provides lock and unlock operations, called `acquireObjectLock` and `releaseObjectLock` (see Figure 3.10). If the object assigned to the `synchronized` statement is already locked by another thread, the current thread is suspended by the Lock Manager. For this purpose, each lock, represented by the class `JNukeLock`, implements a list containing threads that were not able to obtain this lock. As soon as the according lock is completely released, all waiting threads from this list are re-enabled for scheduling. The scheduler may reschedule one of these re-enabled threads that immediately retries to acquire the lock. If this succeeds, it enters the `synchronized` body. Otherwise, the lock manager suspends the thread and adds the thread to the wait list of the corresponding lock again.

[2]Two test cases show the usage of this mechanism: vm/heapmgr/13 and vm/heapmgr/14.

```
int JNukeLockManager_acquireObjectLock (JNukeObj * this,
  void * object,
  JNukeObj * thread)
void JNukeLockManager_releaseObjectLock (JNukeObj * this,
  void * object)
```

Figure 3.10: Method `JNukeLockManager_acquireObjectLock` and method `JNukeLockManager_releaseObjectLock`. Both methods take the pointer to the instance as the second argument. Method `JNukeLockManager_acquireObjectLock` additionally takes a third argument that determines the thread that wishes to obtain the object lock. The method returns 1 if the lock could be obtained. Otherwise, the method returns 0 and the thread is suspended and added to the wait set for this lock.

Consider Figure 3.11 illustrating the relationship between instances, locks and the lock manager. Thread `owner` is the current owner of the lock. Vector `waitList` contains threads that were not able obtain the lock. Each lock is assigned to exactly one instance and vice versa.

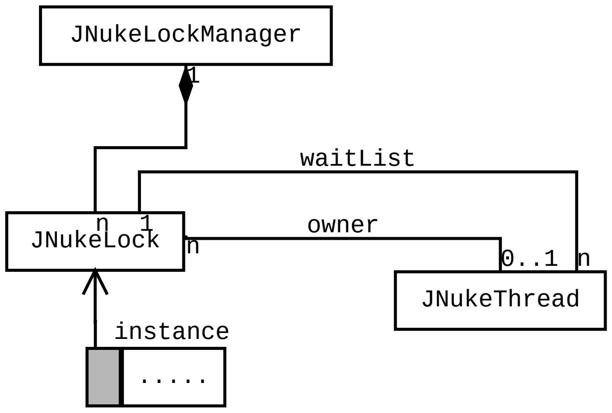

Figure 3.11: Relationship between the Lock Manager, locks, instances and threads.

Listeners and events

As locking activities are often of interest for runtime-verification algorithms, the lock manager also provides a listener registration interface. There are three different events:

OnLockReleased occurs when the current thread has performed an unlock operation, which happens when a thread either invokes `wait` or leaves a synchronized region. Method `JNukeLockManager_addOnLockReleasedListener` registers a listener at the lock manager. The listener is notified by `JNukeLockManagerActionEvents` (shown in Figure 3.12)

OnLockAcquirementSucceed occurs when the lock manager has granted the current thread to obtain a lock. A thread triggers this event when it enters a synchro-

```
struct JNukeLockManagerActionEvent {
  JNukeObj *issuer;
  void *object;
  JNukeObj *lock;
};
```

Figure 3.12: A `JNukeLockManagerActionEvent` provides the pointer to the object and the pointer to the according lock.

nized region or it is awaken by `notify` or `notifyAll`. The registration method is `JNukeLockManager_addOnLockAcquirementSucceedListener` and the corresponding event is `JNukeLockManagerActionEvent` (see Figure 3.12).

OnLockAcquirementFailed occurs when the lock manager could not grant the current thread to obtain a lock. The current thread is suspended. A client that has registered itself with `JNukeLockManager_addOnLockAcquirementFailedListener` are notified. As before, an instance of `JNukeLockManagerActionEvent` represents the event.

3.5 The waitset manager

Beside the `synchronized` statement, Java supports another mechanism for synchronizing threads. The class `java.lang.Object`, contains the methods `wait`, `notify`, and `notifyAll`. Every instance, in addition to having an associated lock, has an associated wait list as illustrated in Figure 3.13. When an instance is first created, its wait list is empty. The `wait` method of the wait set manager adds the current thread to the wait set of the instance, disables the current thread for thread scheduling purposes, and performs a complete unlock on the instance to relinquish the lock on it. Since a thread may obtain an object lock several times (so called *recursive lock*), the lock is unlocked the same number of times (so called complete unlock). The thread then lies dormant until one of two things happen:

- Another thread invokes `notify` on the same instance. The thread is arbitrarily chosen from the set of waiting threads.
- Another thread invokes `notifyAll` on the same instance.

The thread is removed from the wait list and re-enabled for thread scheduling. Once being elected by the scheduler, it will attempt to obtain the locks again.

The waitset manager does not provide an event/listener interface yet, as *ExitBlock* and *ExitBlock-RW* do not need it. Other runtime verification algorithms may need an event/listener interface for notification on `notify`, `notifyAll`, and `wait.`

3.6 The runtime environment

The runtime environment is the heart of the virtual machine and drives the execution of register byte codes in an execution loop. It consists of the class `JNukeRuntime-Environment` and is in relationship to other classes as illustrated in Figure 3.14. The

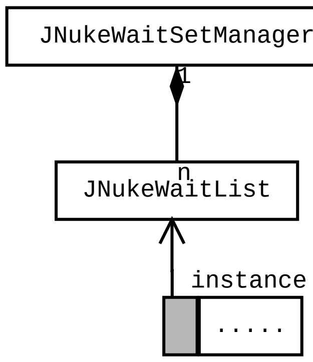

Figure 3.13: `JNukeWaitSetManager` and `JNukeWaitList`

runtime environment reads byte codes and triggers according actions in the static class `JNukeRBCInstruction`. Trivial byte codes are executed in `JNukeRBCInstruction`; others are delegated to the according manager (heap, waitset or lock manager). Creation of instances, reading and writing of fields is managed by the heap manager. Lock and unlock operations are forwarded to the lock manager. The wait set manager handles invocations of `wait`, `notify`, and `notifyAll`.

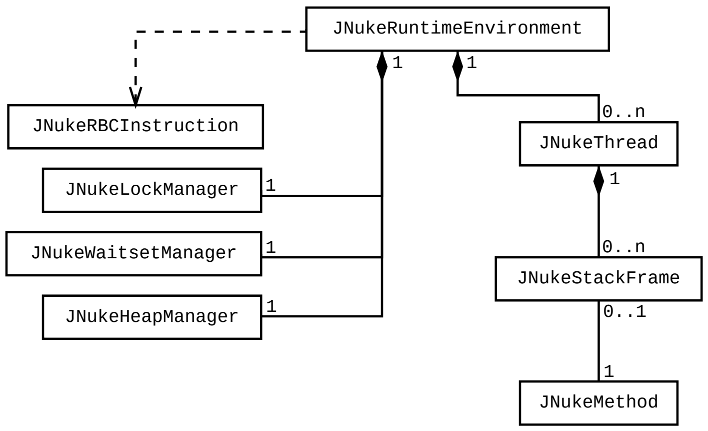

Figure 3.14: The runtime environment with related classes.

Threads are managed by the runtime environment. A thread holds a stack of stack frames. This stack corresponds to the call stack. Each stack frame is associated to a method and contains its own register set. The current method in execution is associated to the top stack frame of the current thread. The current register set is also stored in the top stack frame of the current thread.

The register set

Most register byte codes in a program, such as mathematic, comparison, or branch operations, are simple operations. These operations mainly consist of a calculation and need a few input and output registers or locals. Since Java programs have a high number of these operations, execution of such simple operations should be as fast as possible.

Since the number of locals and registers used in a method is predetermined [26], a register set can be designed as a static array. The width of a register is either 4 bytes on a 32-bit architecture or 8 bytes on a 64-bit architecture. Since 64-bit architectures use 8-byte addresses for objects in memory, a Java register also needs 8-byte on these platforms.

On a 32-bit architecture a 32-bit value consumes one register and a 64-bit value therefore consumes two registers. On 64-bit architectures, both 32-bit and 64-bit values fit into one register.

The register byte code does not distinguish between locals and registers. Both, locals and registers are in the same register set. The enumeration starts with index 0 and ends at $locals + registers - 1$.

Consider Figure 3.15 showing two additions, one with integer values, the other with long values. The integer addition is implemented in the virtual machine as follows: `cur_regs[res_reg] = cur_regs[arg1] + cur_regs[arg2]`.

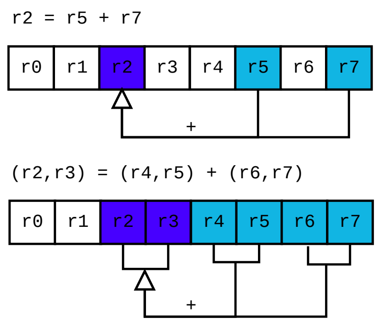

Figure 3.15: Two example additions: one with two integer values, the other with two long values.

Events and Listeners

The runtime environment provides three listeners. The according events are as follows:

onThreadStateChanged is used to notify a listener when the current thread has changed it state which happens if a thread performs `wait`, `join` or it dies. The listener should schedule another thread to execute. The listener is registered by `JNukeRuntime-Environment_addThreadStateListener`.

onExecute is used to notify a listener prior to execution of a byte code. This enables a listener to monitor execution flow, but also to perform thread switches prior to the execution phase. The registration method accepts a bit-mask which determines on which kind of byte-codes the listener is notified. The registration method is `JNukeRuntime-Environment_addOnExecuteListener`.

onExecuted is also used to notify a listener about execution flow. The listener, however, is triggered after the execution phase. As before, the registration method accepts a bit-mask in order to limit notification on a subset of all possible byte codes. A Listener is registered by `JNukeRuntimeEnvironment_addOnExecutedListener`.

The execution loop

The execution loop is presented as a flow chart in Figure 3.16 on the next page. First, a byte code is fetched. The `onExecuteListener` is notified if such a listener is registered. When the listener has finished, the execution loop tests whether the listener has changed the control flow; for instance, a thread switch might happen in the meanwhile. If so, the current byte code is omitted and the loop starts anew. Otherwise, the byte code is executed. After execution, it is tested whether an internal exception such as `ArrayIndexOutOfBounds, IllegalThreadStateException,` etc. has been detected during execution. If so, an instance of the according exception is created and thrown whereupon the virtual machine tries to find a matching exception handler. After this, it is checked whether the state of the current thread has been changed during execution. If necessary, the `onThreadStateChangedListener` is invoked. Finally, the `onExecutedListener` is also called provided there is one defined. The execution loop continues if further byte codes are left. Otherwise, the execution loop stops which also terminates the virtual machine.

Method invocation

The classloader creates for each class a class descriptor (`JNukeClass`) describing members (methods and fields). The class descriptor is not laid out for fast method resolution. Methods are stored in a vector which is iterated for resolution. If the implementation of a method has been derived from a super class the same search procedure is applied to each super class until the method either is found or the search stops at `java/lang/Object`. Since Java uses late binding, method invocations are a time-critical task. The virtual machine consequently implements *vtables* [32]. A vtable for a class `B` contains pairs of method identifiers and references to the methods extracted from the class descriptor of class `B`. The vtable of class `B` also contains methods that are derived from super classes. As an example consider Figure 3.17 on page 40. The vtable for class `B` contains entries for `bar()` and `foo()` since class B implements these methods. Furthermore, there is an entry for `foo(int a)`. The implementation is derived from the super class `A`. The vtable entries are hashed which results in fast method finding.

Entries of the vtable can also refer to native methods. A flag declares whether a vtable entry is a Java or a native method. Native methods can be registered in `vm/native.h` and are statically linked into the virtual machine at compile-time. This approach is not standards compliant [20]; it however provides a basic support for native methods.

3.7 Pluggable schedulers

The runtime environment does not include a scheduler; instead, the virtual machine provides mechanisms that allow to plug in a custom scheduler. Since verification tools sometimes cannot use the built-in scheduler, they have to provide a custom scheduler.

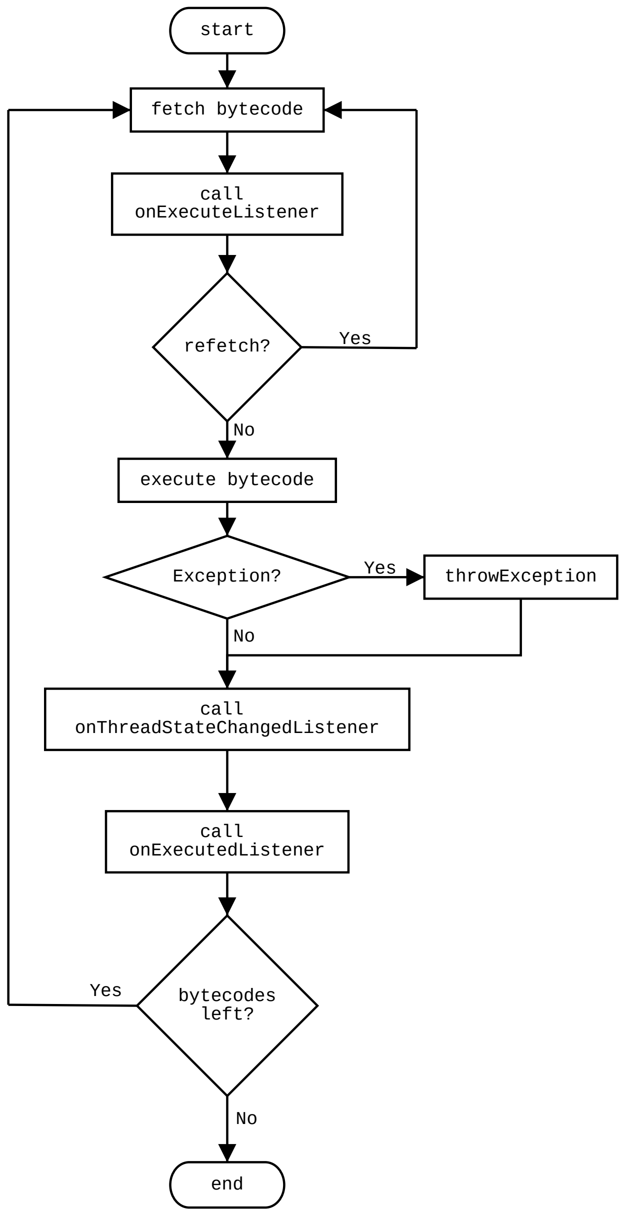

Figure 3.16: Flowchart of the execution loop.

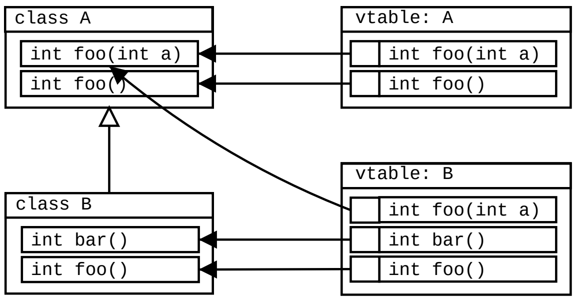

Figure 3.17: Two classes A and B, each with a vtable for fast method resolution

A custom scheduler needs to be able to perform thread switches at arbitrary points of execution and monitor internal execution. Therefore, a scheduler is placed on top of the virtual machine acting as an observer (consider Figure 3.18). Sub-systems of the virtual machine such as the runtime environment, the heap manager, and the lock manager provide registration methods (see Table 3.1), allow a scheduler to install its own listeners. If an event occurs and the scheduler is registered to this event, the scheduler is notified.

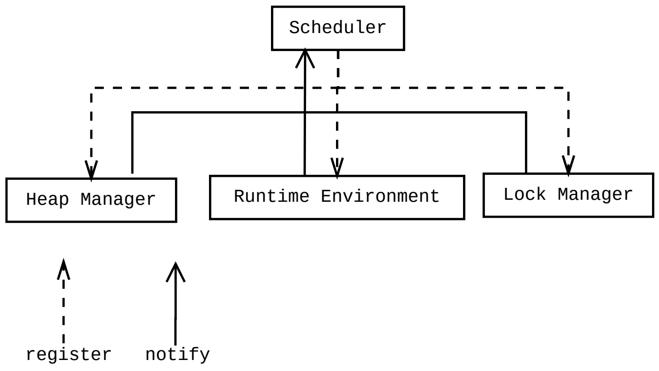

Figure 3.18: The scheduler placed on top of the virtual machine registering own listeners at sub-systems allowing to monitor and steer the execution environment.

A scheduler has to register at least one listener for thread state changes. The virtual machine calls the listener if the current thread has been disabled. This happens if the current thread terminates, performs `wait` or `join`, or could not obtain a lock. The scheduler needs to find the next enabled thread to schedule with `JNukeRuntime-Environment_switchThread`. Such a simple, but complete scheduler performs thread switch only if necessary.

When a scheduler should be notified prior or after execution of certain byte codes, it can install execution listeners with `JNukeRuntimeEnvironment_addOnExecute-Listener` or `JNukeRuntimeEnvironment_addOnExecutedListener`. This enables a scheduler to intervene prior to and after execution of a bytecode. In order to prevent that execution listeners are notified on any byte code execution, both registration methods take a bit-mask for limiting notification to a subset of byte codes.

Name of method
JNukeRuntimeEnvironment_addOnExecuteListener
JNukeRuntimeEnvironment_addOnExecutedListener
JNukeRuntimeEnvironment_addThreadStateListener
JNukeLockManager_addOnLockReleasedListener
JNukeLockManager_addOnLockAcquirementFailedListener
JNukeLockManager_addOnLockAcquirementSucceedListener
JNukeHeapManager_addReadAccessListener
JNukeHeapManager_addWriteAccessListener

Table 3.1: Registration methods of the virtual machine

Tools detecting deadlocks are often interested in locking activities of the target program. For these purposes, the lock manager provides three registration methods. The lock manager can notify listeners on lock and unlock operations. Some tools are also interested in objects accesses on the heap. Thus, the heap manager provides notification on read access and write access, too.

The round-robin scheduler

The virtual machine provides a default round robin scheduler (implemented in the class `JNukeRRScheduler`) granting each thread a fixed time slice in round-robin order [42]. It allows to run multi-threaded programs without any runtime verification. Tools that do not need a custom scheduler can rely on this scheduler. However, the round robin scheduler is not very efficient, as it is notified on any byte code by the runtime environment. The scheduler counts the number of notifications and performs a thread switch when the current thread exhausts its execution slice. As a result, this is rather time consuming. Since the round robin scheduler is that trivial, tools may provide their own scheduler all the same.

3.8 Limitations

Development of the virtual machine has not completed yet. The most important limitation is that I/O operations has not been implemented yet. This is not that trivial as blocking I/O operations need to be treated specially. A thread performing a blocking I/O operation may block the whole virtual machine as scheduling takes place at user-level. Moreover, the Java virtual machine does not provide a standards compliant Java native interface [20].

3.9 Summary

This chapter has shown the design of the virtual machine. The virtual machine is designed as a platform for tools. As such, it provides an event/listener model that allows to expose the execution environment in a well documented way. Tools can listen to the execution flow, locking activities and field accesses. Beside this, the virtual machine also allows tools to control the execution flow.

Since some verification tools have to determine the schedule of a program, they can provide their own scheduler. Schedulers are pluggable using the event/listener model of our custom virtual machine. Tools that do not need a custom scheduler can use the default scheduler (`JNukeRRScheduler`).

During development performance and memory footprint was always kept in mind. The heap manager stores instances in a contiguous block of memory. This is both fast and saves memory. Method invocations are another time critical point in a Java program due to late-binding. Our virtual machine adds vtables providing fast method resolution. Furthermore, a register set was design as an array, which is best in terms of performance and memory footprint.

Chapter 4

The Milestone and Rollback Mechanism

The milestone/rollback mechanism allows establishing milestones in program execution. These milestones can be reverted to at a later point of execution. A milestone records the whole state of the virtual machine such that after a rollback, the state can be accurately restored. This chapter explains how this mechanism is implemented and how it can be used. The main goal was to achieve good performance as well as small memory footprint. Moreover, the mechanism should be easy to use. The next section shows the usage of the mechanism, followed by Section 4.2, focusing on the implementation.

4.1 Usage

The interface of the milestone/rollback mechanism consists of three methods provided by the runtime environment: `setMilestone`, `rollback` and `removeMilestone`. Method `setMilestone` establishes a milestone at the current execution point. It is possible to establish an arbitrary number of milestones which can be reverted to in the opposite order of their creations. A rollback always reverts to the last milestone. Method `removeMilestone` destroys the last milestone allowing to revert to the next milestone.

As an example, consider Figure 4.1. First, two milestones are established at two different points of execution. Each milestone is reverted to twice, which results, in three different paths being explored.

4.2 Implementation

The runtime environment provides an easy-to-use interface for the milestone/rollback mechanism, hiding internal details. The actual work, however, is delegated to the subsystems of the runtime environment. This pattern is also called *facade* [12]. Figure 4.2 illustrates how invocations of `setMilestone`, `rollback`, and `removeMilestone` are delegated to the numerous subsystems. Each of them provides the same three methods (`setMilestone`, `rollback`, and `removeMilestone`) again. The runtime environment

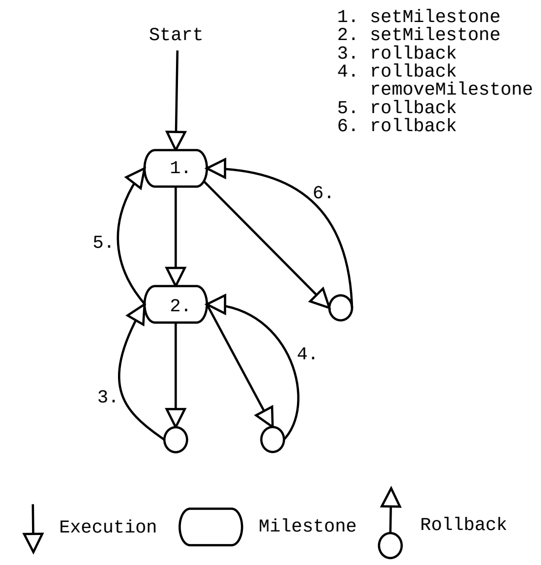

Figure 4.1: Sample sequence of `setMilstone`, `rollback` and `removeMilestone`.

forwards invocation of any one of these three methods to its sub-systems whereupon each sub-system also forwards the invocation to its sub-systems.

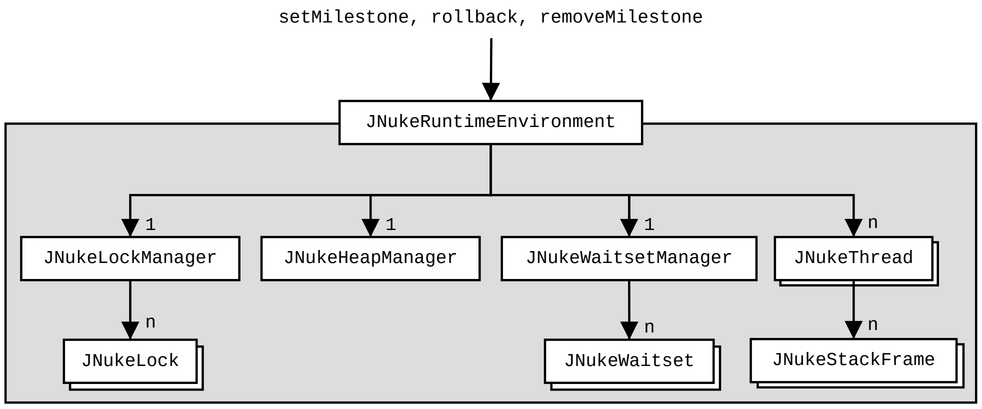

Figure 4.2: Illustrates the delegation of the milestone/rollback mechanism

When `setMilestone` is applied to an instance, the state of the instance is copied on to a stack. Each instance holds its own stack. On a rollback, the copy on top of the stack is written back to its original place. As a result, the rollback mechanism does not change any references. An instance is located at the same address prior to and after a rollback. Method `removeMilestone` applied to an instance finally removes the copy on top of the stack. As an example, consider Figure 4.3 and 4.4 presenting the implementation of `rollback` and `setMilestone` for class `JNukeLock`. The idea is all the same for other classes.

There is one exception. The heap manager does not copy each Java instance when

```
clone = JNuke_malloc (this->mem, sizeof (JNukeLock));
memcpy (clone, lock, sizeof (JNukeLock));
JNukeVector_push (lock->milestones, clone);
```

Figure 4.3: Implementation of `JNukeLock_setMilestone`

```
c = JNukeVector_count (lock->milestones);
milestone = JNukeVector_get (lock->milestones, c - 1);
lock->waitList = JNukeObj_clone (milestone->waitList);
```

Figure 4.4: Implementation of `JNukeLock_rollback`

establishing a milestone. Since only a fraction of all Java instances are usually modified, copying all Java instances would waste to much memory and time. Instead, the heap manager records modification of Java instances in a log (see `JNukeHeapLog`). When a Java instance is modified for the first time, a copy of this instance is created. A rollback writes back any recorded instances. The heap manager defines two recording strategies which differ in their granularity. When a Java instance is modified for the first time, the heap manager can either record the whole instance or just the field which is written. Recording of whole instances usually consumes more memory. However, it is considered faster as fewer copies need to be written back.

Assume an array of 100’000 integer components. The total size of this instance is about 400 kilobytes. Recorded as a whole, the log has one entry consuming about 400 kilobytes which can be written back as a whole. Recording single components, however, creates one entry for each written component. This results in many small log entries where each of them is written back individually. This is time consuming. It is difficult to predict which strategy is best. Therefore, the heap manager provides the method `setLoggingGranularity` allowing to choose the strategy at startup.

4.3 Summary

The milestone/rollback mechanism presented in this chapter provides an easy to use interface consisting of just three methods: `setMilestone`, `rollback`, and `remove-Milestone`. When a milestone is established the state of the virtual machine is saved. A rollback writes the state back where instances keeps their addresses. Heap objects are saved on their first write access and written back on a rollback the same way. The rollback mechanism works in-place, which means that Java and `JNuke` objects never change their addresses because of a rollback. This allows a tool to work with references to objects regardless of whether rollbacks are performed in the meanwhile.

Chapter 5

Experiments

This chapter is structured into two sections. The next section presents the results of running the virtual machine on a number of example Java programs. Section 5.2 shows how the systematic scheduler detects concurrency errors by means of a number of small examples.

The testing platform used was an Intel Pentium IV with a processor of 2 GHz and 512 MB of RAM. The workstation runs Red Hat Linux 8.1 Beta 2 running kernel 2.4.20 with a back-port of the NPTL (Native POSIX thread library) [34] from Kernel 2.5. The new virtual machine, the `JNuke` VM, was measured against Sun's JDK 1.4 and the former interpreter of JNuke. As Sun's JDK compiles Java byte code to native machine code with a just-in-time compiler, a comparison is not fair from that point of view. However, it can show whether the virtual machine's performance allows to execute extensive programs.

The virtual machine and the interpreter were compiled with GCC 3.2.1 with the following settings: `-O3 -march=pentium4 -mcpu=pentium4`. All tests were run multiple times; results were averaged.

5.1 Performance and memory usage of the VM

This section focuses on performance and memory footprint of the virtual machine. In the first part, some basic tests consider execution performance of single-threaded applications. The second part illustrates that the virtual machine uses memory sparingly, as opposed to the `JNuke` interpreter. The third part shows some benchmarks for multi-threaded programs.

Single-threaded tests

There are eight different basic tests which consider various aspects of the virtual machine. They have in common that each test is single-threaded and therefore no side effects due to the scheduling can interfere these tests. The tests are as follows:

MethodInvocation considers the performance of method invocations. Since Java uses late binding, invocations of methods is a time-critical part. The test consists of a class filled with 500 methods. Each of them is then called 4000 times (see Section B.2).

ReadManyFields extensively tests read and write operations on fields. The test program is shown in Section B.3. Class *ReadManyFields* contains 5000 static fields where each field is written and read two hundred times.

Iteration consists of a loop repeated one hundred million times (see Section B.4).

Array iterates over an array of integer values. The array contains ten million components where each component is written ten times. (see Section B.5).

MultiArray iterates over a six dimensional array ($10\times10\times10\times10\times10\times10$) where each component is written once (see Section B.6).

DoubleOp performs a couple of floating point operations in a loop with two million iterations. The program is shown in Section B.7.

BubbleSort implements a simple bubble sort algorithm. All the tests above are quite simple tests and not representative as each of them test more or less one aspect. Thus, this test tries to be more representative. It sorts an array with ten thousand components. The source is shown in Section B.8.

JASPA [5] is an extensive benchmark that performs sparse matrix multiplications (see Section B.9). The JASPA project provides sources for F90, C and Java. Since the JNuke VM does not support I/O yet, the program contains the input data statically in a class file. The input matrix is real unsymmetric and filled with 2659 values and contains 180×180 rows and columns. The matrix is obtained from Matrix Market [11]. Like Bubble Sort, this test is quite representative as sparse matrix multiplications appear frequently in large scale scientific and engineering applications.

Each of the eight tests above was run on the JNuke VM, the former interpreter and Sun's VM 1.4. The results are presented in Table 5.1 and Figure 5.1 on the next page.

As expected, Sun's VM beats our custom virtual machine and the interpreter in almost any discipline. Since byte-codes for iteration and other simple operations can be compiled into very efficient machine code, Sun's VM is very strong at such disciplines. For instance, tests Iteration, Bubble Sort and JASPA are three benchmarks where Sun's VM is between fifty and hundred times faster than the JNuke VM. Byte codes for method invocations or field accesses hardly profit of a just-in-time compiler. The JNuke VM catches up a bit. However, Sun's VM is still twenty times faster. Averaged over all test cases, Sun's VM is about thirty times faster than the JNuke VM. Sun's VM allows also to disable the just-in-time compiler[1]. Sun's VM without just-in-time compiling is still eight times faster. Since the JNuke VM is in an early stage and not fully optimized yet, I am confident that it will catch up in the future. In particular, method invocations, field resolution, and execution of primitive byte codes can be improved in many ways.

All eight test cases were also executed by the JNuke interpreter where the eighth test, JASPA, crashes. The JNuke Interpreter can neither keep up with the JNuke VM nor with Sun's VM. In particular, the interpreter is very slow at method invocations and field resolution. The JNuke VM is about 120 times faster in this discipline. Overall,

[1]Use option `-Xint`.

the JNuke VM passes all tests about thirty times faster. Sun's VM manages this even thousand times faster.

Table 5.1 compares also the execution time of optimized and non-optimized register byte code. Test MethodInvocation becomes slower as the optimization process by itself consumes extra time. Averaged however, the benefit is about twelve percent.

Test	Sun JVM		JNuke JVM		JNuke
	with JIT	w/o JIT	RBC	OptRBC	Interpreter
MethodInvocation	0.51	0.51	11.37	13.09	1589.99
ReadManyFields	0.63	0.58	19.43	18.78	4422.92
Iteration	0.38	2.00	23.71	19.56	166.78
Array	0.87	4.74	57.91	44.79	1000.74
MultiArray	2.04	5.23	21.76	14.91	517.92
DoubleOp	9.19	10.55	25.65	24.71	152.03
BubbleSort	0.76	15.21	115.23	90.40	2155.76
JASPA	0.45	3.86	24.02	23.46	–
Total	14.83	42.68	299.08	249.70	10006.14
Geometric mean	0.81	3.12	28.76	25.31	794.16
Overhead	**1.00**	**3.87**	**35.62**	**31.34**	**983.58**

Table 5.1: Results of the basic tests in seconds

Memory consumption

Memory consumption is another aspect to analyze. For these purposes it was analyzed how much memory is allocated for an example Java object whose class is shown in Figure 5.2. The class contains 9 integer fields, 9 long fields, and 9 references. Integers and references consume at least 4 bytes on an i386 platform. Long fields need at least 8 bytes of memory. Thus, altogether at least 148 bytes for one instance. Our custom virtual machine adds a header of 16 bytes at the beginning of the block such that the raw instance needs 164 bytes (without the instance descriptor, which is shared). The interpreter wastes much more memory: an approximate size of 2280 bytes for each instance was determined which is 14 times more.

The same applies to arrays as Figure 5.2 shows. An array of integer values with 100'000 components takes at least 400 kilobytes of memory. Our custom virtual machine adds a header of 16 bytes whereas the interpreter needs about 2 megabytes memory for the same array. This is four times more.

	JNuke VM	JNuke Interpreter
`new SampleClass()`	164 Bytes	2280 Bytes
`new int[100000]`	400 kilobytes	2000 kilobytes

Table 5.2: Comparison of memory footprint between the JNuke VM and the former interpreter of JNuke.

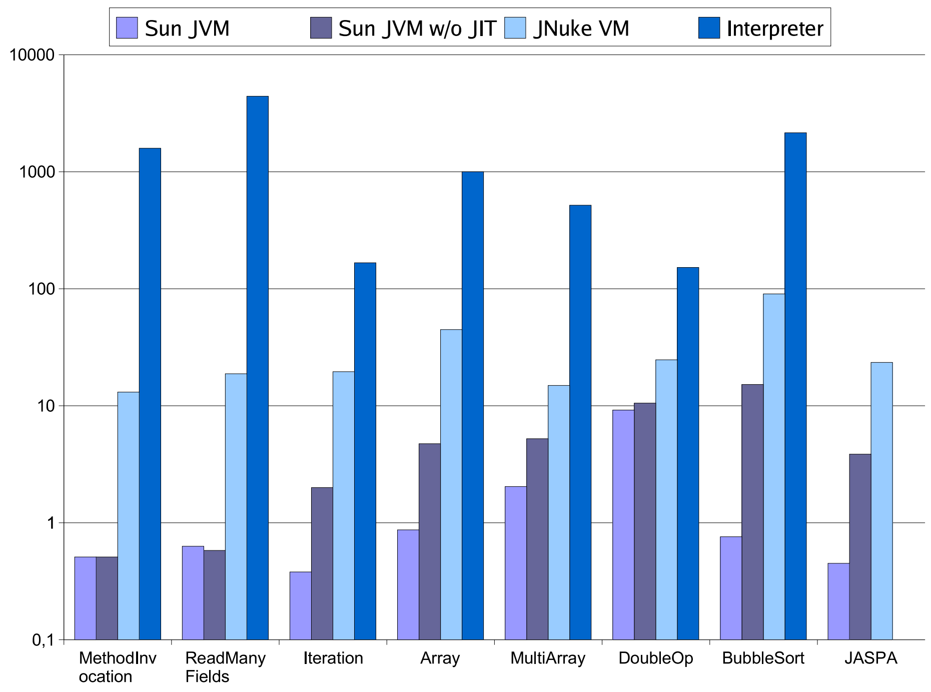

Figure 5.1: Benchmark results as a chart.

```
public class SampleClass {
  int a1, a2, a3, a4, a5, a6, a7, a8, a9;
  long l1, l2, l3, l4, l5, l6, l8, l9;
  Object o1, o2, o3, o4, o5, o6, o7, o8;
}
```

Figure 5.2: Example class with a couple of fields in order to compare the memory consumption between the JNuke VM and the intepreter

Multi-threaded tests

This section tests the performance of multi-threaded programs. The test program are presented in Appendix B. Since the former interpreter does not support threading, the JVM of Sun and our custom JVM only pit one's strength against each other. We consider four programs:

Dining philosophers is a classic synchronization problem. The problem consists of three philosophers sitting at a table who do nothing but think and eat. Between each philosopher, there is a single stick. In order to eat, a philosopher has to obtain both sticks. A problem can arise if each philosopher grabs the stick on the right, then waits for the stick on the left. In this case a deadlock has occurred, and all philosophers will starve. The solution is that one philosopher has to acquire its sticks in the opposite order. Since performance is measured, this implementation meets this criterion.

Producer-consumer problem is another classic synchronization problem which is also called *bounded buffer problem.* There is a set of producers and consumers. Producers write elements into a buffer as long the buffer is not full. Consumers consume elements from the buffer as long the buffer is not empty. The producer-consumer problem illustrates the need for synchronization in systems where many processes share a resource.

JGFCrypt performs *IDEA* (*International Data Encryption Algorithm*) encryption and decryption on an array of three million components. JGFCrypt is part of the *Java Grande Forum Benchmark Suite* [40]. The algorithm involves two principle loops, whose iterations are independent and are divided between the threads in a block fashion (see Section B.10).

JGFSeries is also part of the Java Grande Forum Benchmark Suite. It computes the first ten thousand fourier coefficients of the function $f(x) = (x+1)^x$ in multiple threads. The most time consuming component of the benchmark is the loop over the Fourier coefficients. Each iteration of the loop is independent of every other loop and the work may be distributed simply between the threads. The work of this loop is divided evenly between the threads in a block fashion, with each thread updating the elements of its own block [40] (see Section B.11).

JGFSparseMatmult uses an unstructured sparse matrix stored in compressed-row format with a prescribed sparsity structure. This test exercises indirection addressing and non-regular memory references. A 50'000$\times$50'000 sparse matrix is used for 200 iterations. The principle computation involves an outer loop over iterations and an inner loop over the size of the principal arrays. The simplest parallelization mechanism is to divide the loop across the array length between threads. Parallelizing this loop creates the potential for more than one thread to up-date the same element of the result vector. To avoid this the non zero elements are sorted by their row value. The loop has then been parallelized by dividing the iterations into blocks, which are approximately equal, but adjusted to ensure that no row is accessed by more than one thread [40] (see Section B.12).

First, a *Dining Philosopher* program is considered with three dining philosophers where each of them acquires their shared resources 30'000 times. The whole program executes 28’564’800 byte codes where the resulting execution performance depends upon the number of thread switches performed by the scheduler. Since the number of thread switches is determined by the size of the execution slice each thread receives, the *Round Robin Scheduler* was configured with different slices as Table 5.3 illustrates. The test has shown, that if a thread is able to execute many byte codes without interruption the overall performance is better due to fewer thread switches. However, the lost of efficiency between the first and the last test is 13% only. As a result, the number of thread switches hardly affect the performance of the JNuke VM.

Time slice [μs]	Bytecode slice	Time [s]	Thread switches	Instr/sec
34	5	27.55	793216	1036834
125	100	24.46	195496	1167816
704	1000	24.32	34520	1174539
6984	10000	23.86	3416	1197183

Table 5.3: Results of the Dining Philosophers program where the scheduler allows each thread to execute 5, 100, 1000, or even 10000 bytecodes without interruption.

The *Producer-consumer* program (Producer-consumer-1) contains a producer and a consumer thread working on a buffer with one slot. Each thread performs 120’000 enqueue or dequeue operations on this buffer. The main difference to the dining philosophers program is that threads in the producer-consumer program are very often blocked. Since the buffer has just one slot, the producer thread and consumer thread are alternated by the scheduler. Threads always compete for this one slot. The dining philosophers program, however, causes fewer lock collisions as n threads share n locks. The consequences are shown in Table 5.4: As opposed to Sun’s VM, the JNuke VM catches up if threads are often blocked. It seems that the JNuke VM takes advantage of user-level scheduling instead of kernel level scheduling. When a thread is blocked a thread switch occurs, where scheduling on user-level consumes less time.

The second producer-consumer program (Producer-consumer-2) contains one hundred producers writing into one buffer slot one thousand times. There is one consumer thread consuming all produced elements. The one consumer has to be alternated with all producers. In worst case, it happens that the scheduler wakes up many producers where each of them immediately goes to sleep again as the slot is still full. This is enormous time-consuming as any thread switch carries weight. In particular, this applies to kernel-level threads. As a result, Sun’s VM needs without NPTL [34] about six minutes for Producer-consumer-2. The JNuke VM passes this test in forty seconds. Apparently, the NPTL scales much better than the current *Pthreads-Lib* of kernel 2.4.x.

The JGF benchmarks are larger and more representative than the previous examples (see Table 5.4). The Sun VM executes JGFCrypt about fifty times faster and JGFSeries about twenty times faster. With disabled just-in-time compiler, the Sun VM is still twenty times faster for JGFCrypt and eight times faster for JGFSeries. Averaged over all multi-threaded tests, Sun’s VM with enabled just-in-time compiler is about seventeen times faster than the JNuke VM. Sun’s VM with disable just-in-time compiler is still seven times faster.

Benchmark	Sun JVM		JNuke VM
	with JIT	w/o JIT	
Dining philosophers	1.46	4.64	23.86
Producer-consumer-1	4.5	7.18	23.28
Producer-consumer-2	7.70	12.41	40.48
Producer-consumer-2 (w/o NPTL)	371.1	602.34	40.48
JGFCrypt (2 threads)	3.31	8.80	167.92
JGFCrypt (20 threads)	3.22	8.29	162.55
JGFCrypt (200 threads)	3.28	8.51	155.10
JGFSeries (2 threads)	22.32	61.78	510.52
JGFSeries (20 threads)	22.62	63.81	515.43
JGFSparseMatMult (2 threads)	5.79	20.17	537.36
JGFSparseMatMult (20 threads)	3.84	17.64	527.41
Geometric mean	7.89	20.10	135.70
Overhead	**1.00**	**2.55**	**17.20**

Table 5.4: Results of various multi-threaded benchmarks

5.2 ExitBlock and ExitBlock-RW

This section presents experiments of running *ExitBlock* and *ExitBlock-RW* on a number of example Java programs. The programs presented here are small programs, but they are scaled-down versions of what one would expect to encounter in real applications. They all exhibit different types of concurrency errors which are detected. The listings can be found in Appendix B.

Performance

The first program does not contain any errors but it is dedicated to measure how many schedules *ExitBlock-RW* executes for a specified number of threads and locks per thread. The program creates a number of threads and a number of locks. Each thread acquires each lock once and does nothing else (see listing in Section B.13). Thus, there are no inter-thread data dependencies. Table 5.5 lists the results of running the program with various number of threads and locks. The number of schedules per second is only reported for tests lasting longer than one second, otherwise the initialization of the virtual machine would falsify the result.

This test is taken from the Rivet paper originally describing *ExitBlock-RW* [7]. This paper also provides results of running the target program with various number of threads and locks. Since the Rivet virtual machine presumes a JDK1.1.5 which is not available anymore, time comparisons cannot presented in this thesis. It is, however, possible to compare the number of schedules executed, as shown in Table 5.6. It attracts attention that the number of schedules are the same for tests with two threads. However, for tests with more than two threads, the number of executed schedules are not the same anymore. I enumerated the number of schedules for simple examples by hand in order to proof whose implementation is right. The number of schedules enumerated by hand always accorded to the number of schedules executed by our implementation. I assume that the implementation of Rivet handles creation of new threads differently. This may explain why the differences occur only for tests with more than two threads.

Threads	Locks	Test case	Memory	Schedules	Time [s]	Schedules/sec
2	1	17	764.4 kBytes	4	0.11	
2	2	18	764.4 kBytes	5	0.11	
2	100	19	2.7 MBytes	103	2.89	35.6
3	1	20	764.4 kBytes	9	0.11	
3	50	21	1.8 MBytes	156	1.63	95.7
3	100	22	4.2 MBytes	306	9.77	31.3
4	20	23	1.2 MBytes	130	1.07	121.5

Table 5.5: Test case rv/exitblock/17 ... 23 execute `Performance.java` for different number of threads and locks.

Threads	Locks	Schedules	
		Rivet [7]	JNuke
2	1	4	4
2	2	5	5
2	100	103	103
3	1	13	9
3	50	2757	156
3	100	10507	306
4	20	11155	130

Table 5.6: Comparison of executed schedules by Rivet's systematic scheduler and our systematic scheduler using *ExitBlock-RW*.

Deadlock

Figure 5.3 on this page contains two threads acquiring two locks in a different order. The first thread obtains the two locks in order (B, A), while the second thread obtains them in the order (A, B). If the first thread obtains B and then the second thread obtains A we have reached a deadlock, since each thread holds the lock the other thread seeks. The reverse lock cycle detector correctly discovers this deadlock regardless of whether *ExitBlock* or *ExitBlock-RW* is chosen. Figure 5.4 shows how the deadlock is reported.

```
T0:                           T1:
synchronized(B) {             synchronized(B) {
  synchronized(A) {             synchronized(A) {
  }                             }
}                             }
```

Figure 5.3: Two threads, both containing nested synchronized regions where each thread acquires the locks in a different order.

```
Deadlock found at LockAB.run ()V (line 15) (pc 9) (thread 1)
The according schedule is:
(JNukeSchedule
  (JNukeThreadSwitch (from_thread 0) (to_thread 0)
    (JNukeMethod "Main1.main" (JNukeSignature "V" (JNukeVector
      "[Ljava/lang/String;"))) (pc 8) (line 9))
  (JNukeThreadSwitch (before) (from_thread 0) (to_thread 1)
    (JNukeMethod "Main1.main" (JNukeSignature "V" (JNukeVector
      "[Ljava/lang/String;"))) (pc 15) (line 13))
)
```

Figure 5.4: The output produced by rv/rlcanalizer/4 testing the program shown in Section B.14.

Deadlock3

The program shown in Figure 5.5 contains three threads where the first thread acquires locks A and B, the second thread acquires locks B and C and finally the third thread acquires lock C and A. The deadlock occurs when the first thread acquires A, the second thread acquires B, and the third acquires C. Then none of the threads can proceed since a different thread hold the lock it seeks.

```
T0:                    T1:                    T2:
synchronized(A) {      synchronized(B) {      synchronized(C) {
  synchronized(B) {      synchronized(C) {      synchronized(A) {
  }                      }                      }
}                      }                      }
```

Figure 5.5: Three threads, each acquiring two locks in a cycle order.

The *ExitBlock* algorithm explores 946 different schedules (see rv/exitblock/24) and the reverse lock cycle analyzer is able to detect the deadlock shown in test case rv/rlcanalizer/7. The *ExitBlock-RW* however, that explores 11 schedules, is not able to detect the deadlock. This is because no data dependencies exist between any atomic block. Therefore, the *ExitBlock-RW* algorithm alternates only the order in which each thread comes to run where no other thread is re-enabled and scheduled during the execution.

DeadlockWait

The program presented in Section B.18 shows a common type of condition deadlocks. A first thread holding two Locks `A` and `B` from a nested *synchronized* section performs *wait* on `B` whereupon the thread releases the lock `B` but not `A`. Since the first thread still holds Lock `A` no one else can obtain this lock while the first thread is sleeping. Thus, a second thread aims to notify the first thread but blocks on lock `A` instead. *ExitBlock* and *ExitBlock-RW* find the condition deadlock.

SplitSync

The fragment shown in Figure 5.6 demonstrates another timing-dependent bug. Depending on the schedule the invariant *r.x == y* may fail. By splitting the increment of *r.x* into two synchronized statements, an error will occur if the two threads are interleaved between the synchronized statements. Both threads will read the original value of *r.x*, and both will then set it to one plus its original value, resulting in the loss of one of the increments (see Section B.16).

```
T0 and T1:
0: synchronized (r) { y = r.x; }
1: synchronized (r) { r.x = y + 1; }
```

Figure 5.6: SplitSync example.

The program SplitSync was successfully tested by *ExitBlock* and also by *ExitBlock-RW*. Both algorithms find the assertion violation at the second explored schedule where *ExitBlock* executes 37 and *Exitblock-RW* executes 17 schedules.

Dining Philosophers

The dining philosophers program is a typical problem where condition deadlocks may occur. The code example from Section B.17 is such an example where all the philosophers try to acquire their forks in the same order. The condition deadlock occurs if each philosopher has successfully acquired one fork and is waiting for the other one. In this case, each thread is waiting for an infinite time. According Figure 5.7, which shows the implementation of `Fork.java`, there are no inter-thread data dependencies. Remember, entering a synchronized region is not considered as a write action to the lock object.

The program has been tested with *ExitBlock* (rv/exitblock/29) and *ExitBlock-RW* (rv/exitblock/30). *ExitBlock* executes 5871 schedules where the condition deadlock is found after 843 explored schedules. *ExitBlock-RW* executes 51 schedules where the condition deadlock is not detected due to lack of inter-thread data dependencies. The number of schedules enormously explodes for *ExitBlock* so that it cannot applied to realistic problems. While *ExitBlock-RW* is able to cut down the number of schedules, the condition deadlock is not discovered.

```
public synchronized void acquire(Philosopher p)
    throws InterruptedException {
  while( owner != null ) { wait(); }
  owner = p;
}
public synchronized void release() {
  owner = null;
  notifyAll();
}
```

Figure 5.7: Implementation of `Fork.java`

Test Case	Memory	Time [s]	Threads	Schedules	p-factor
rv/exitblock/30	767.0 kBytes	0.10	3	51	3.75
rv/exitblock/31	767.0 kBytes	0.13	4	75	3.11
rv/exitblock/32	1001.9 kBytes	0.92	10	1146	3.06
rv/exitblock/33	1.8 MBytes	309.24	20	305978	4.22

Figure 5.8: *Dining Philosophers* for 3, 4, 10 and 20 running threads.

Figure 5.8 shows the result of *ExitBlock-RW* applied to the *Dining Philosopher* problem for 3, 4, 10 and finally 20 concurrent threads. Even though the condition deadlock is undetected, this test is appropriate to examine growth factors of *ExitBlock-RW*. The p-factor is defined as follows:

$$p = \frac{\log(number\_of\_schedules)}{\log(number\_of\_threads)}$$

It shows the polynomial dependency between the number of threads and the resulting number of schedules. As all the p-factors are more or less the same, it can be assumed that *ExitBlock-RW* is polynomial in the number of threads for this problem. The memory consumption grows linearly.

BufferIf

The *BufferIf* program contains a bounded buffer that has an error in its *enqueue* method. The program is shown as a whole in Section B.19 and as a simplified portion in Figure 5.9. The program is easy to understand: a producer inserts an element into the buffer if the buffer is not full. Otherwise, it performs `wait` on the buffer object. This puts the thread to sleep until another thread performs `notifyAll` on the same buffer object. Similarly, any consumer performs `wait` if the buffer is empty. The consumer thus sleeps until another thread performs `notifyAll`. When this happens the consumer checks whether the buffer is still empty. If so, it performs `wait` again. Otherwise, the consumer consumes one element from the buffer and leaves the method. The error is that the enqueue method of the buffer does not re-check the condition on notification. The program uses an `if` to test the condition that the buffer is full instead of a `while` loop. This is a problem if more than one thread sleeps on the buffer object waiting for enqueuing an element. Those threads are woken up by `notifyAll` when a consumer has consumed one element as there is space for one new element again. Since the condition is not rechecked at the enqueue method, it may happen that more than one thread enqueues an element which causes a buffer overflow.

```
synchronized enqueue(Object x)
if( full ) /* BUG */
  wait();
/** insert element */
notifyAll();

synchronized dequeue()
while( empty )
  wait();
/** consume element */
notifyAll();
```

Figure 5.9: *Producer/Consumer* example where the condition is not rechecked in the *enqueue* method.

The sample program creates two producers and one consumer. Both, *ExitBlock* and *ExitBlock-RW* discover many schedules where the assertion in the enqueue method fails. Since we just check the assertion (*jnuke.Assertion.check*) when the assertion fails (execution does not stop), the systematic scheduler encounters later in execution a condition deadlock at the dequeue method. Iff the assertion fails, the consumer thread sleeping in the dequeue method will never leave the while loop anymore. After all producers are terminated, the consumer will be never woken up, which is properly detected as a condition deadlock.

BufferWhile

The BufferWhile program is a modification of the BufferIf program from above. The enqueue method uses a `while` loop now instead of an `if` statement in order to check the condition (see Section B.20 for the source code). Even when two producer threads are waiting and awakened by `notifyAll`, the assertion cannot fail as the condition is rechecked by each thread. The thread that was scheduled first can enqueue the next element whereupon the second scheduled thread is not able to escape the `while` loop. The correctness is shown by *ExitBlock* and by *ExitBlock-RW*. There is, according to the log, neither a condition deadlock nor an assertion violation.

5.2.1 Performance of the milestone/rollback mechanism

Table 5.7 presents how much time is consumed by the rollback/milestone mechanism. It illustrates also the number of created milestones and the number of rollbacks performed. Test case A and B are the results from the dining philosophers program with ten or twenty threads, respectively. C, D, E executes `Performance.java` with different number of threads and locks:

- C: two threads and hundred locks
- D: three threads and fifty locks
- E: three threads and hundred locks

Test case	Rollback		Milestone		Total Execution	User
	Number	Time [s]	Number	Time[s]	Time [s]	Time [s]
A	4061	0.36	2915	0.15	0.92	0.41
B	644753	82.19	338653	31.84	309.24	195.21
C	10405	1.80	10302	0.72	2.89	0.37
D	8009	0.92	7853	0.38	1.63	0.33
E	31009	6.25	30703	2.45	9.77	1.07

Table 5.7: Time behaviour of the milestone/rollback mechanism

As Figure 5.10 shows, the milestone/rollback mechanism consumes for the dining philosophers program (test A and B) between 40 and 55% of the total execution time. Test C, D and E even need up to 90% of the execution time for the milestone/rollback mechanism. The results do not surprise, as both test programs hardly contain any real code. They primarily consist of code causing creation of milestones where there is hardly any code in-between two milestones. Due to lack of time I could not test more representative programs. However, we can consider time consumed by one rollback or milestone creation, respectively. Table 5.8 presents these figures. In a previous test (see Table 5.3), we stated that the virtual machine executes about one million instructions per second for the dining philosopher program (this is one instruction per μs). In average, a rollback or a milestone creation cost about 100 μs for test case A and B.

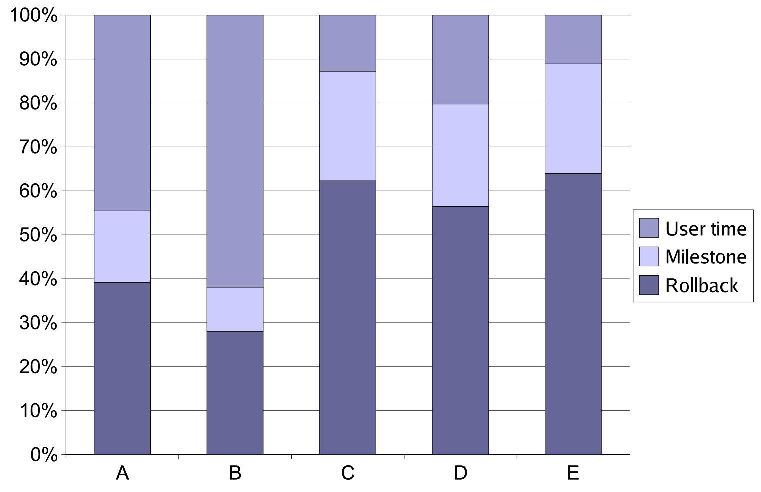

Figure 5.10: Time consumption of the rollback/milestone mechanism

5.3 Summary

This chapter has shown that the JNuke VM executes programs about thrity times faster than the former interpreter of JNuke. Sun's virtual machine is much faster than our custom virtual machine due to the just-in-time compiler. In average, it is about seven-

Test case	Rollback [μs]	Milestone [μs]
A	88.6	51.6
B	127.4	94.0
C	172.9	69.9
D	114.8	48.4
E	201.5	79.8

Table 5.8: Time consumption for one rollback or milestone.

teen times faster. This depends on the mix of byte codes. Programs that mainly consist of byte-codes for branching, calculation, etc. can be compiled into very efficient machine code. In such disciplines our custom virtual machine cannot compete with the virtual machine of Sun. If a program, however, consists of complex bytes such as method invocation, field accesses, or floating point operations, our custom virtual machine can catch up a bit due to efficient implementation of the heap manager and the virtual tables. Our virtual machine profits of the user-level scheduling, which allows an efficient implementation of thread switches. In a concurrent program where many threads compete for a small set of locks, our custom virtual machine even beats Sun's virtual machine under some circumstances.

This chapter has also considered experiments with the systematic scheduler. It has been shown that many types of common errors in concurrent Java programs can be discovered. *ExitBlock* finds all the errors in the example programs. It enumerates an enormous number of schedules such that it apparently cannot be applied to larger examples. *ExitBlock-RW* also tests larger examples. However, some deadlocks are not discovered due to lack of data dependencies. Therefore, Chapter 6 discusses among other things approaches that enables *ExitBlock-RW* to detect any deadlock.

Chapter 6

Future Work

6.1 Future work on the virtual machine

The virtual machine is far from complete as three months of programming is a rather short time to implement a whole virtual machine. At certain points the virtual machine is either incomplete, not standards compliant enough, or not optimized. This section names these deficiencies below.

Input/output

The virtual machine currently does not provide any I/O facilities. I/O classes use native methods which are not implemented yet. The implementation is straightforward. One problem, however, arises for blocking I/O in connection with multi-threaded Java programs. Since the virtual machine uses user-level instead of kernel-level threads, any thread that performs a blocking operation would block all other threads. The virtual machine is designed as one process which is preempted by the operating system on *blocking system calls*. As a result, the whole virtual machine is blocked waiting for data, regardless of whether other Java threads could run. Kernel-level threads, however, profit of the operating system which immediately preempts blocking threads and put them on a system wait queue. As soon as the data are ready to deliver, the thread is awaken again [3].

The virtual machine, therefore, needs similar mechanisms avoiding blocking of the whole virtual machine. There are two different approaches:

- Replacing blocking system-calls such as `read`, `write`, `connect`, `accept`, `sleep`, `wait`, etc. by non-blocking versions. The virtual machine uses `select` internally to manage non-blocking I/O. A Thread invoking a blocking system call is disabled until the virtual machine has completed the system call. A few user-level pthreads implementations do the same [35]. The main advantage is that any standard foundation library can be used with our virtual machine without any modifications.

- Replacing classes of the foundation library providing blocking I/O. This approach is easier to implement; however, replacing parts of commercial foundation libraries may give rise to copyright problems.

Unlike our virtual machine, the Rivet virtual machine fully supports I/O operations as it can make use of the underlying native libraries and virtual machine [7].

Java native interface

The virtual machine provides basic support for native methods. Native methods are implemented in the virtual machine and statically registered there. Native methods cannot be loaded from shared objects at runtime which would be standards compliant [20]. We need a standards compliant Java native interface otherwise the Java foundation classes (JFC) can hardly be used, since many of them uses native methods.

Java foundation classes (JFC)

Commercial java virtual machines usually provide their own implementation of the JFC. The license, however, often forbids to use them with our virtual machine. There is a project called *GNU Classpath* whose goal is to provide a free replacement for Sun's proprietary implementation [8]. Using GNU Classpath as the class library for our Java virtual machine does not affect the licensing of the JVM. If we use GNU Classpath, the JNuke VM may need some minor adjustments.

Dynamic loading and linking

A standards compliant virtual machine dynamically loads, links and initializes classes and interfaces on demand. Loading is the process of finding the binary representation of a class or interface type with a particular name. Linking is the process of taking a class or interface and combining it into the runtime state of the Java virtual machine so that it can be executed [26]. Loading and linking are currently done at startup of the virtual machine by means of a list of class files to load. There is no possibility to load and link classes or interfaces at run-time which is a desirable feature as the runtime environment aborts if a required class file was not predetermined at the virtual machine's startup.

Standards compliant handling of NaN and infinity

Since the classloader currently omits values *not a number* and *infinity* [26], the virtual machine does not execute programs correctly using these special values as they cannot be represented at the moment.

Deterministic replay of a schedule

So far the virtual machine provides a round-robin scheduler, which cannot be used for deterministic replay. Errors detected by a runtime-verification tool, such as *ExitBlock* or Eraser [36], should be replayed on our virtual machine in order to reproduce an error. So, we need a further scheduler which is able to execute a multi-threaded program according a schedule history created by a runtime-verification tool. Schedule histories can be recorded with `JNukeSchedule`. The desired replay scheduler can retrieve the history from an instance of `JNukeSchedule`.

Writing of cx-files

Marcel Baur has written a Java bytecode instrumentation tool allowing deterministic replays of schedules on an arbitrary virtual machine [1]. The schedule is described in a cx-file. Class `JNukeSchedule`, that encapsulates a schedule, should be extended such that cx-files can be generated.

Use of 32-bit registers on SPARC v8

SPARC v8 is a 32-bit architecture which insists on 8-byte alignment for 8-byte values. If registers of the virtual machine consist of 4 bytes and an 8-byte value is stored in a register that is 4-byte aligned, a bus error occurs on read or write of this address. Therefore, registers on SPARC v8 consist of 8 bytes which makes sure that any register is 8-byte aligned. This wastes memory. The solution is to split one 8-byte access into two 4 byte accesses.

Just-in-time compiler

The experiments in Chapter 5 have shown that our custom virtual machine is much slower than Sun's virtual machine due to lack of a just-in-time compiler. A just-in-time compiler, however, may come at a cost, as some flexibility get lost. However, most tools for runtime verification are mainly interested in activities concerning locks, field accesses, thread states and method invocations. As long as our custom virtual machine provides this information to tools, a just-in-time compiler hardly affect the flexibility. Since `JNuke` uses register byte code, writing of code generators for various platforms is straightforward.

6.2 Future work on the milestone/rollback mechanism

There are Java instances whose state also depends on external properties where restoring blocks of memory is not sufficient for a rollback. For instance, I/O classes can use files. A proper rollback may consist of closing file handles or even make write accesses undone. The milestone/rollback mechanism implemented in this thesis is not able to do so. Such Java classes need to implement the rollback mechanism by themselves as the virtual machine does not know enough about their accurate states.

One idea is that such classes implement a special interface as considered in Figure 6.1. Classes implementing this interface are bound to implement an according milestone/rollback mechanism. The milestone/rollback mechanism of the virtual machine omits classes derived from `Revertable` and delegates those tasks by invoking the corresponding method of the `Revertable` interface.

As a result, this approach enables almost any class, even those classes with native states, to accurately establish milestones which can be reverted to. In particular, this is interesting for I/O classes. The Rivet virtual machine does not provide such a facility.

```
public interface Revertable {
  /** save current state */
  void setMilestone();
  /** revert to last state */
  void rollback();
  /** remove last saved state */
  void removeMilestone();
}
```

Figure 6.1: Interface `Revertable`

6.3 Future Work on ExitBlock-RW

Handle a lock enter as a write to the lock object

Lock-cycle deadlocks may not be found due to lack of data dependencies. If, however, a lock enter is considered to be a write to the lock object, *ExitBlock-RW* discovers more lock-cycle deadlocks. The penalty in extra paths considered should not be too high as synchronized sections often have data dependencies. A program usually acquires an object lock if it aims to read or write the object. This often results in data dependencies.

Re-enable disabled threads on potential condition deadlocks

Threads once disabled are enabled in further branches only if data dependencies exist between the disabled thread and the current thread. If the scheduler runs out of enabled threads, it declares a condition deadlock unless there are disabled threads. The current implementation of *ExitBlock-RW* aborts these paths without declaring a condition deadlock. *ExitBlock-RW*, however, could execute those disabled threads nevertheless to determine if it truly is a condition deadlock.

This idea combined with the idea from above may guarantee complete deadlock detection even for *ExitBlock-RW*. Note that this has not been proven.

Allowing to define start point, depth and width of the depth-first search

ExitBlock and *ExitBlock-RW* allow to define classes or package which are assumed thread-safe[1]. For large applications the number of considered schedules is still too high all the same. It should be possible to test parts of an application allowing to define start point, depth and width of the depth-first search. This allows to test particular aspects and parts of a program separately.

Heuristic or randomized branching behaviour

Exhaustive testing often last too long. Since timing-dependent errors usually occur on several paths, they may be discovered even some branches are omitted. A heuristic algorithm should prune paths which lead to similar behaviours. The most difficult part is to develop a strategy that decides which branches to omit. Static analysis of

[1]see `JNukeExitBlock_addSafeClasses`.

the target program may help to find critical sections in a program which finally helps to find paths of interest. Randomized branching behaviour relies on the same observation. Since timing-dependent errors occur on several paths, randomized branching should also find most errors.

Explore several paths at a time

Since the depth-first search of *ExitBlock* and *ExitBlock-RW* work strictly sequentially, there is no profit of multi-processor environments. Applying a divide-and-conquer strategy in order to explore the tree of schedules, allows it to implement massive parallelization of the problem. When a milestone is created the enabled set can be divided into n subsets. Then $n-1$ additional processes are forked [3]. Each process is accountable for its own subset of enabled threads creating own branches. This results in execution of multiple paths at the same time. Since each process has its own address space bothering of other processes is impossible. It is not that difficult to build a systematic scheduler using multi-processing. The number of processes should be tuned to the number of processing units available. Usual symmetric multiprocessing systems typically have 2, 4, 8 or even more processing units. As child processes in Unix share unmodified memory pages, each child process needs just a little memory. The paralyzed divide-and-conquer strategy can also be applied to clustered environments [6] that may allow exhaustive testing of even larger programs with *ExitBlock-RW*. As an example, consider the program in Figure 6.2 and one possible parallelization, shown in Figure 6.3.

```
T0:                            T1:
1: t1 = new LockAB (A, B);     7:  synchronized (A) {
2: t1.start();                 8:    synchronized(B) {
3: synchronized (B) {          9:    }
4:   synchronized (A) {        10: }
5:   }
6: }
```

Figure 6.2: Sample multi-threaded program with two threads.

6.4 Summary

This chapter has shown some future work on the virtual machine, the milestone/rollback mechanism, and the *ExitBlock-RW* algorithm. The virtual machine is incomplete at some points. This concerns I/O, special values as *NaN* and *infinity*, and the Java native interface. This chapter has also considered a possible implementation of a just-in-time compiler. Access to the the internal execution environment should be still possible whereupon the execution speed is increased. We hope that our custom virtual machine can then achieve a performance similar to the one of Sun's virtual machine.

This chapter also has discussed some modifications on *ExitBlock-RW* allowing more accurate deadlock detection. It was also considered how heuristics and randomized algorithms may help to reduce the number of schedules to be considered. Another idea was to profit of multi-processing environments by applying a divide-and-conquer strategy to the depth-first search.

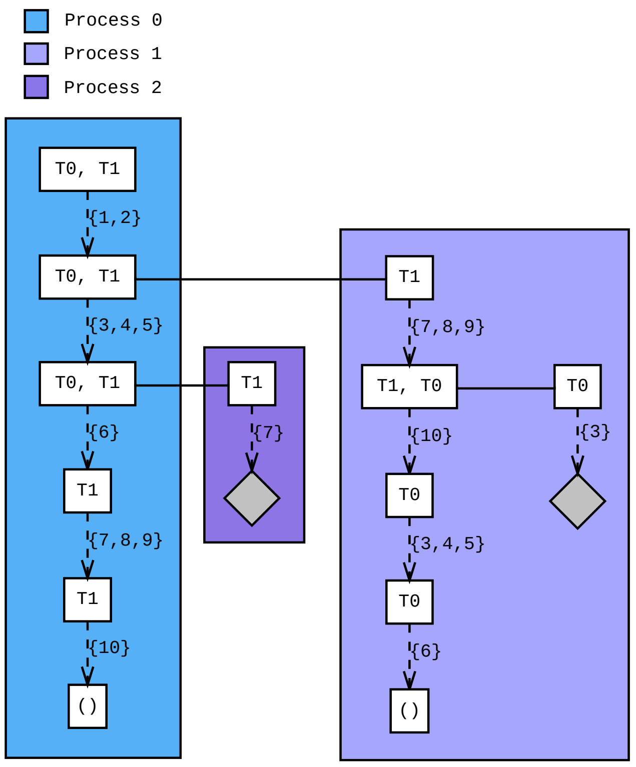

Figure 6.3: Example parallelization of the program shown in Figure 6.2.

Chapter 7

Conclusions

The execution order of concurrent programs is nondeterministic due to the apparent randomness in the way threads are scheduled. Techniques for testing sequential programs consist of test suites where the target program is run on different representative sets of inputs. A concurrent program may pass such a test suite in spite of concurrent errors. As only a fraction of the possible schedules is covered, the test suite cannot cover the behaviour of the entire program. Therefore, a behaviour-complete systematic scheduler is needed. This presumes a virtual machine that exposes its internal execution environment in a consistent way. The systematic scheduler performs a depth-first search to enumerate possible schedules. So the virtual machine provides a milestone/rollback mechanism that allows to establish milestones, which can be reverted to. We have built such a mechanism on top of a custom virtual machine.

There are two systematic schedulers implemented in this thesis: *ExitBlock* and *ExitBlock-RW*. Both consider only interleavings of synchronized sections, assuming that a target program follows a mutual-exclusion locking discipline. *ExitBlock-RW* additionally avoids interleaving of synchronized sections without data-dependencies. The number of schedules considered by *ExitBlock* grows exponentially in the number of threads and locks per thread. This allows only to test very small examples. The number of schedules considered by *ExitBlock-RW* grows polynomially in the number of locks per thread.

The milestone/rollback mechanism achieves good performance and low memory footprint. Our custom virtual machine is fast enough to execute one schedule even though Sun's JVM is much faster. Exhaustive testing of large programs, however, needs a faster virtual machine. A just-in-time compiler would find a remedy. Good heuristics or randomized procedures combined with a multi-process depth-first search may help as well. In addition, users often debug just a part of a program. Thus, the systematic scheduler should allow to limit the scope of the depth-first search.

Since our systematic scheduler is able to discover numerous timing-dependent errors (deadlocks and assertion violations), it is worth spending more time for behaviour-complete testing, assuming we manage to reduce the number of schedules to be considered and increase performance of the virtual machine.

Appendix A

API Documentation

This chapter contains the complete API documentation extracted from the source code. The following table lists all existing classes used either for the virtual machine or the systematic scheduler. Figure A.1 on the next page shows a complete UML diagram of the virtual machine and the systematic scheduler. It helps to understand the virtual machine and to see the relationships between classes.

Name	Page	Description
JNukeArrayInstanceDesc	69	Describes an array instance
JNukeExitBlock	70	*ExitBlock* and *Exitblock-RW* algorithms
JNukeHeapLog	71	Records modification of Java instances
JNukeHeapManager	72	Manages Java instances
JNukeInstanceDesc	77	Describes a class or object instance
JNukeLock	78	Represents an instance lock
JNukeLockManager	80	Manages instance locks
JNukeRBCInstruction	81	Executes register byte codes
JNukeRLCAnalyzer	90	Reverse lock chain analyzer algorithm
JNukeRRScheduler	90	Implements the round robin scheduler
JNukeRuntimeEnvironment	91	The runtime environment
JNukeSchedule	96	Records thread switches during execution
JNukeStackFrame	97	Represents a stack frame of the call stack
JNukeThread	99	Represents a Java thread
JNukeVirtualTable	104	Contains a virtual table for a Java class
JNukeVMState	106	Holds a VM state for reporting purposes
JNukeWaitList	108	Represents a wait list
JNukeWaitsetManager	109	Manages wait sets of instances

Table A.1: List of JNuke classes which are part of the virtual machine or the systematic scheduler.

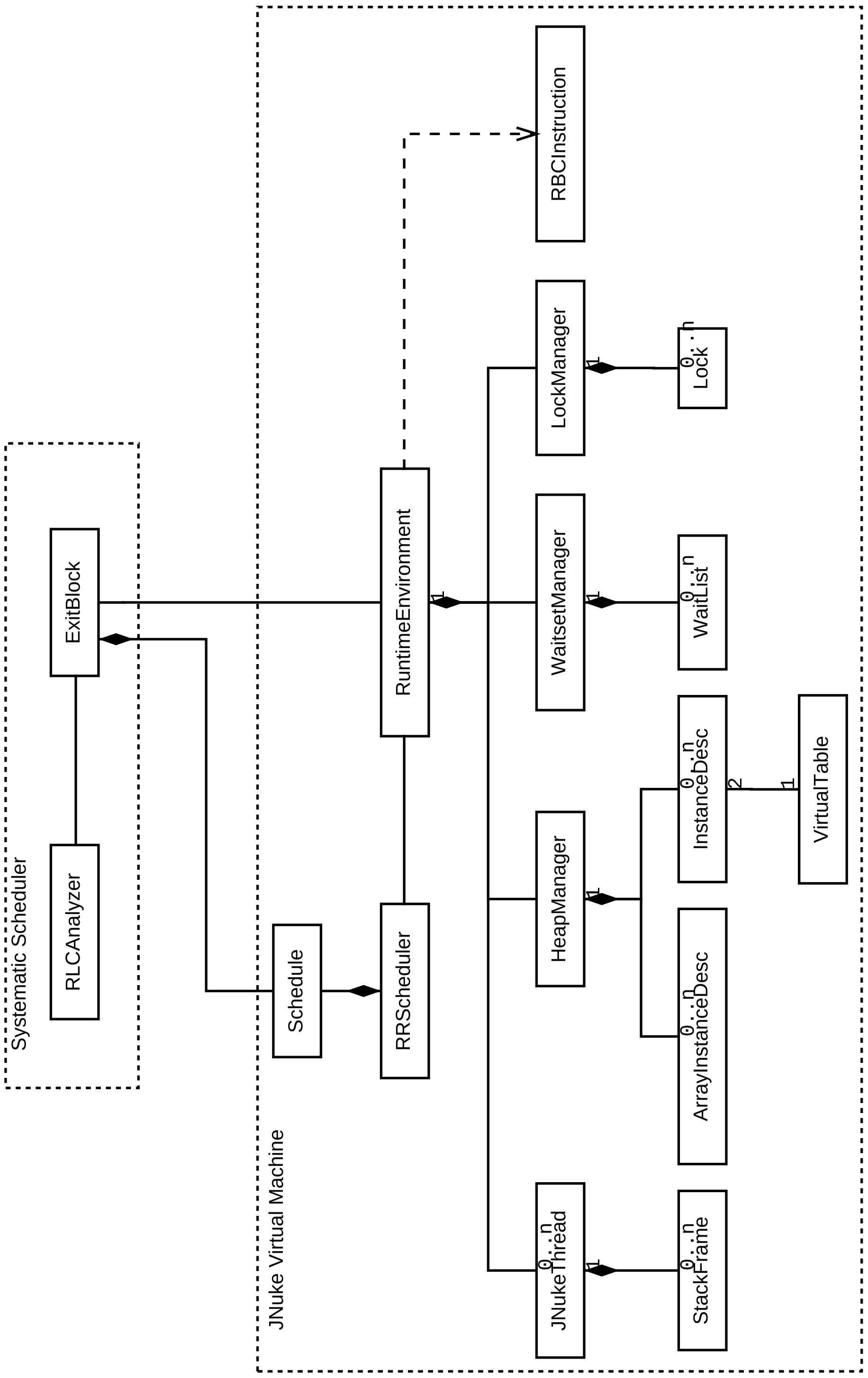

Figure A.1: UML diagram of the virtual machine and the systematic scheduler

A.1 JNukeArrayInstanceDesc

Describes array instances of the same type. It provides methods for component access and array instance creation.

JNukeArrayInstanceDesc_createInstance

```
void * createInstance (JNukeObj * this, int size)
```

Creates an array instance (one dimension only). Returns the pointer to the newly created array instance.
in:
size – number of elements

JNukeArrayInstanceDesc_dereferenceType

```
JNukeObj * dereferenceType (JNukeObj * this)
```

Dereferences the type of the array ([[I becomes [I, ...). The dereferenced type is returned as UCSString. Note that it is up to the client to release the UCSString's memory. Thus, it is best to insert this string into the constant pool.

JNukeArrayInstanceDesc_getEntryOffset

```
int getEntryOffset (JNukeObj * this, int n)
```

Determines the offset for the n-th component of the array.
in:
n – n-th element (0...)

JNukeArrayInstanceDesc_getEntrySize

```
int getEntrySize (JNukeObj * this)
```

Returns the size of one component in bytes. This number is at least as large as the alignment, determined by the underlying architecture.

JNukeArrayInstanceDesc_getLength

```
int getLength (void *instance)
```

Returns the length of the array instance. Length means the number of components. It equals to the length operator of the Java language. Method getLength returns -1 if the instance is NULL.

JNukeArrayInstanceDesc_getType

```
JNukeObj * getType (JNukeObj * this)
```

Returns the type of the array as UCSString.

JNukeArrayInstanceDesc_new

```
JNukeObj * new (JNukeMem * mem, JNukeObj * heapMgr, JNukeObj * type)
```

Constructor for JNukeArrayInstanceDesc.
in:
heapMgr – the corresponding heap manager.
type – type of the array as UCSString.

A.2 JNukeExitBlock

JNukeExitBlock implements the *ExitBlock* and *ExitBlock-RW* algorithm.

JNukeExitBlock_addOnEndOfPathListener

```
void addOnEndOfPathListener (JNukeObj * this, JNukeObj * listenerObj,
JNukeExitBlockEndOfPathListener (listenerFunc))
```

Registers a listener that is called when an end of a path is reached. For instance, this call back can be used to print out a schedule or to analyze the executed schedule.

JNukeExitBlock_addOnLockAcquirementFailedListener

```
void addOnLockAcquirementFailedListener (JNukeObj * this, JNukeObj *
listenerObj, JNukeExitBlockLockListener (listenerFunc))
```

Registers a listener that is called when a lock could not be acquired. Limitation: there can be only one such listener at the same time.
in:
listenerObj – object reference to the listener
listenerFunc – function pointer to the call back function

JNukeExitBlock_addSafeClasses

```
void addSafeClasses (JNukeObj * this, const char *path)
```

Classes that are considered safe can be marked as such. In such classes no milestones are created. As a result this reduces the number of executed schedules. The argument can either contains a class or even whole packages. Consider following examples: java/lang/String, java, or java/lang.
in:
path – this is either a class or a package

JNukeExitBlock_getSchedule

```
JNukeObj * getSchedule (const JNukeObj * this)
```

Returns the schedule (instance of JNukeSchedule). The returned schedule includes all thread switches until the current point of execution. Such schedule can be used to replay a schedule in order to reproduce an discovered error.

JNukeExitBlock_init

```
void init (JNukeObj * this, JNukeObj * rtenv)
```

Initializes the ExitBlock algorithm. Needs to be called prior to the start of the virtual machine.
in:
rtenv – the runtime environment

JNukeExitBlock_new

```
JNukeObj * new (JNukeMem * mem)
```

JNukeExitBlock_setLog

```
void setLog (JNukeObj * this, FILE * log, int logLevel)
```

Sets the file stream used for the log. The argument logLevel determines how verbose the log will be. 0 means non verbose whereas 1 is verbose.

JNukeExitBlock_setMode

```
void setMode (JNukeObj * this, JNukeExitBlockMode mode)
```

ExitBlock supports both ExitBlock and ExitBlock-RW. Thus, the argument mode can be set either to JNukePureExitBlock or JNukeExitBlockRW.

A.3 JNukeHeapLog

Class JNukeHeapLog is a class which records heap modifications. It can record fields or entire instances. Based on the recorded modifications JNukeHeapLog allows to restore the state of the heap.

JNukeHeapLog_count

```
int count (const JNukeObj * this)
```

Returns the number of log entries

JNukeHeapLog_logFieldWriteAccess

```
void logFieldWriteAccess (JNukeObj * this, void **obj_root, int offset,
int size)
```

Called when a field access shall be logged. A log entry is created and stored iff neither the field nor the whole object has been recorded before.
in:
obj_root – pointer to root of the object
offset – offset from obj_root
size – number of bytes to save

JNukeHeapLog_logObjectCreation

```
void logObjectCreation (JNukeObj * this, void *root)
```

Called when a object creation shall be logged. Write accesses to such objects are not recorded then as they are removed at a rollback by all means.
in:
root – pointer to root of the object

JNukeHeapLog_logObjectWriteAccess

```
void logObjectWriteAccess (JNukeObj * this, void **obj, int size)
```

Called when an object write access has to be logged. A log entry is created and stored iff neither the field nor the whole object has been recorded before.
in:
obj_root – pointer to root of the object
size – number of bytes to save

JNukeHeapLog_new

```
JNukeObj * new (JNukeMem * mem)
```

JNukeHeapLog_rollback

```
void rollback (JNukeObj * this)
```

Backs up the heap according the recorded heap log.

JNukeHeapLog_setHeapManager

```
void setHeapManager (JNukeObj * this, JNukeObj * manager)
```

Sets the heap manager
in:
manager – JNukeHeapManager reference

A.4 JNukeHeapManager

Class JNukeHeapManager provides methods for instance creation and for field accesses of instances. The heap manager can record heap modifications in connection with the class JNukeHeapLog. This allows to back up the heap at any time to a previous state.

JNukeHeapManager_aLoad

```
int aLoad (JNukeObj * this, void *obj, int n, JNukeRegister * value)
```

Loads a value from an array instance into the target register. The method fails if the index is out of bounds. The result is 0, then. Otherwise, 1.
in:
obj – pointer to the array instance
n – array offset
value – pointer to the target register

JNukeHeapManager_aStore

```
int aStore (JNukeObj * this, void *obj, int n, JNukeRegister * value)
```

Writes a value to an array instance at the declared offset. Fails if the array index is out of bounds. The result is 0 then. Otherwise, 1.
in:
obj (JNukePtr) pointer to array instance –
n – array offset
value – pointer to a register
out:
if method fails return value is 0. Otherwise, 1.
Method fails iff n is out of array range

JNukeHeapManager_addReadAccessListener

```
void addReadAccessListener (JNukeObj * this, JNukeObj * listenerObj,
JNukeHeapManagerActionListener (l))
```

Registers a listener that is notified when a heap objects was read
in:
listenerObj – listener object
l – listener function pointer

JNukeHeapManager_addWriteAccessListener

```
void addWriteAccessListener (JNukeObj * this, JNukeObj * listenerObj,
JNukeHeapManagerActionListener (l))
```

Registers a listener that is notified when a heap objects was written
in:
l – listener

JNukeHeapManager_countArrayInstances

```
int countArrayInstances (JNukeObj * this)
```

Returns the number of array instances managed by the heap manager

JNukeHeapManager_countObjectInstances

`int countObjectInstances (JNukeObj * this)`

Returns the number of objects instances managed by the heap manager

JNukeHeapManager_createArray

`void * createArray (JNukeObj * this, JNukeObj * type, int dimension, int *sizes)`

Creates an (multi-)array with declared dimension.
in:
type – type as UCSString ("[L", "[[[[I", ...)
dimension – depth of creation recursion
sizes – array of integers providing size of each dimension
out:
Returns the pointer to the allocated memory block. NULL if method failed.

JNukeHeapManager_createObject

`void * createObject (JNukeObj * this, JNukeObj * class)`

Creates an object instance of a class.
in:
class – name of class (UCSString)
out:
Returns a pointer to the allocated memory block. NULL if method failed.

JNukeHeapManager_deleteLatestArrayInstances

`void deleteLatestArrayInstances (JNukeObj * this, int n)`

Deletes the n latest object instances. Called by the HeapLog when a rollback is performed.

JNukeHeapManager_deleteLatestObjectInstances

`void deleteLatestObjectInstances (JNukeObj * this, int n)`

Deletes the n latest object instances. Called by the HeapLog when a rollback is performed.

JNukeHeapManager_getClassInstanceDesc

`JNukeObj * getClassInstanceDesc (JNukeObj * this, JNukeObj * class)`

Finds a class instance descriptor by name.
in:
class – name of class (UCSString)

out:
Returns the pointer to corresponding JNukeInstanceDesc. NULL if no descriptor was found.

JNukeHeapManager_getField

```
int getField (JNukeObj * this, JNukeObj * class, JNukeObj * field, void
*obj, JNukeRegister * value)
```

Reads a field of an object instance. The result is written into the declared register. The method fails if the desired field does not exist.
in:
class – name of class (UCSString)
field – name of field (UCSString)
obj – this pointer of target object
value – pointer to the target register

JNukeHeapManager_getHeapLog

```
JNukeObj * getHeapLog (const JNukeObj * this)
```

Returns the current heap log. If no log is defined, NULL is returned.

JNukeHeapManager_getObjectInstanceDesc

```
JNukeObj * getObjectInstanceDesc (JNukeObj * this, JNukeObj * class)
```

Finds an object instance descriptor. Returns NULL if no corresponding descriptor exists.
in:
class – name of class (UCSString)

JNukeHeapManager_getStatic

```
int getStatic (JNukeObj * this, JNukeObj * class, JNukeObj * field, JNukeRegister
* value)
```

Reads a static field of a class instance. The result is written into the target register.
in:
class (UCString) – name of class
field (UCString) – name of field
value – pointer to the target register

out:
If method fails, return value is 0. Otherwise, 1.

JNukeHeapManager_new

```
JNukeObj * new (JNukeMem * mem, JNukeObj * classPool)
```

out:

JNukeHeapManager_putField

```
int putField (JNukeObj * this, JNukeObj * class, JNukeObj * field, void
*obj, JNukeRegister * value)
```

Writes a field of an object instance. If method fails return value is 0. Otherwise, 1.
in:
class – name of class (UCSString)
field – name of field (UCSString)
obj – this pointer of target object
value – pointer to the source register

JNukeHeapManager_putStatic

```
int putStatic (JNukeObj * this, JNukeObj * class, JNukeObj * field, JNukeRegister
* value)
```

Writes a static field of a class instance.
in:
class – name of class (UCString)
field – name of field (UCString)
value – pointer to the source register
out:
if method fails return value is 0. Otherwise, 1.

JNukeHeapManager_setHeapLog

```
void setHeapLog (JNukeObj * this, JNukeObj * heapLog)
```

Sets the the heap log.
in:
heapLog – reference to heap log to use. If reference is NULL, no log is written.

JNukeHeapManager_setLoggingGranularity

```
void setLoggingGranularity (JNukeObj * this, enum JNukeLoggingGranularity
granularity)
```

Sets the logging granularity. This is either JNukeFieldGranularity or JNukeObject-Granularity.

A.5 JNukeInstanceDesc

Describes class and object instances of the same type. It provides methods for calculation of field offsets, but also methods used for creation of instances.

JNukeInstanceDesc_createInstance

```
void * createInstance (JNukeObj * this)
```

Factory method that creates an instance according the description. Returns the pointer to newly created instance.

JNukeInstanceDesc_getClass

```
JNukeObj * getClass (JNukeObj * this)
```

Returns the class description corresponding to the descriptor.

JNukeInstanceDesc_getClassInstance

```
void * getClassInstance (const JNukeObj * this)
```

Returns the pointer to the class instance.

JNukeInstanceDesc_getFieldInfo

```
int getFieldInfo (JNukeObj * this, JNukeObj * class, JNukeObj * field,
int *offset, int *size)
```

Returns the offset and size for a given field. The unit of an offset is either 4 or 8 bytes, respectively. This depends on the platform.
in:
class – name of class (UCSString)
field – name of field (UCSString)

out:
offset – offset in slots (4 or 8 byte slots)
size – size in bytes

JNukeInstanceDesc_getSize

```
int getSize (JNukeObj * this)
```

Returns the used memory in bytes for an instance of this descriptor.

JNukeInstanceDesc_getVirtualTable

```
JNukeObj * getVirtualTable (const JNukeObj * this)
```

Returns the virtual table of this instance

JNukeInstanceDesc_new

`JNukeObj * new (JNukeMem * mem, enum instance_desc_types type, JNukeObj * classDesc, JNukeObj * classPool)`

Creates a new instance descriptor.
in:
heapMgr – the heap manager
classDesc – the class description

JNukeInstanceDesc_setClassInstance

`void setClassInstance (JNukeObj * this, void *classInstance)`

Sets the class instance.

JNukeInstanceDesc_setVirtualTable

`void setVirtualTable (JNukeObj * this, JNukeObj * vtable)`

Sets the virtual table of this instance.

A.6 JNukeLock

Represents an object lock. A lock has an owner thread, a corresponding object, and a wait list with threads that aims to acquire this lock, too.

JNukeLock_acquire

`JNukeObj * acquire (void *object, JNukeObj * thread)`

Tries to obtain a lock at this object for this thread. If a lock could be obtained, the reference to the lock is returned.
in:
thread – thread that would like to obtain the look
object – instance that should be locked.

JNukeLock_getN

`int getN (const JNukeObj * this)`

Returns the number of times the current owner has acquired the lock.

JNukeLock_getObject

`void * getObject (const JNukeObj * this)`

Ech lock belongs to an object. This method returns this object.

JNukeLock_getOwner

```
JNukeObj * getOwner (const JNukeObj * this)
```

Returns the current owner thread.

JNukeLock_new

```
JNukeObj * new (JNukeMem * mem)
```

JNukeLock_release

```
int release (JNukeObj * this)
```

Releases a lock. Returns the number of times the lock is still locked by its owner.

JNukeLock_releaseAll

```
void releaseAll (JNukeObj * this)
```

Performs a complete lock release such that the lock can be reacquired again by other threads.

JNukeLock_removeMilestone

```
int removeMilestone (JNukeObj * this)
```

Removes the current milestone from the top of the stack.

JNukeLock_resumeAll

```
void resumeAll (JNukeObj * this)
```

Awakens all threads from the wait set. The wait set is flushed. This is usually used by notifyAll (awaken threads from the wait set even though the current object is still locked by another thread)

JNukeLock_resumeNext

```
void resumeNext (JNukeObj * this)
```

Awakens the next thread waiting on the lock.

JNukeLock_rollback

```
int rollback (JNukeObj * this)
```

Backs up a lock state. Returns 1 if there was at least one milestone remaining. Otherwise, rollback() returns with 0.

JNukeLock_setMilestone

```
void setMilestone (JNukeObj * this)
```

Creates a milestone. This means that the state of the current lock is copied and pushed on to a stack of prior lock states. A prior lock state can be restored by performing rollback() on this lock.

A.7 JNukeLockManager

JNukeLockManager manages all objects locks and provides locking primitives, such as acquireObjectLock and releaseObjectLock.

JNukeLockManager_acquireObjectLock

```
int acquireObjectLock (JNukeObj * this, void *object, JNukeObj * thread)
```

Acquires a lock on given object for a given thread. If the object is either unlocked or already locked by this thread the result is 1. Otherwise, the result is zero and the thread is appended to the waitset. Further, this thread lost its readyToRun flag.

JNukeLockManager_addOnLockAcquirementFailedListener

```
void addOnLockAcquirementFailedListener (JNukeObj * this, JNukeObj *
listenerObj, JNukeLockManagerActionListener (listenerFunc))
```

Registers a listener that is called when a lock could not be acquired. Limitation: there can be only one such listener at the same time.
in:
listenerObj – object reference of the listener
listenerFunc – function pointer to the call back function

JNukeLockManager_addOnLockAcquirementSucceedListener

```
void addOnLockAcquirementSucceedListener (JNukeObj * this, JNukeObj *
listenerObj, JNukeLockManagerActionListener (listenerFunc))
```

Registers a listener that is called when a lock could be acquired. Limitation: there can be only one such listener at the same time.
in:
listenerObj – object reference of the listener
listenerFunc – function pointer to the call back function

JNukeLockManager_addOnLockReleasedListener

```
void addOnLockReleasedListener (JNukeObj * this, JNukeObj * listenerObj,
JNukeLockManagerActionListener (listenerFunc))
```

Registers a listener that is called when a lock was released. Limitation: there can be only one such listener at the same time.

in:
listenerObj – object reference of the listener
listenerFunc – function pointer to the call back function

JNukeLockManager_new

```
JNukeObj * new (JNukeMem * mem)
```

JNukeLockManager_releaseObjectLock

```
void releaseObjectLock (JNukeObj * this, void *object)
```

Releases an object lock. If the recurive lock counter becomes zero the lock is removed from the thread's lock set.

JNukeLockManager_releaseThreadLocks

```
int releaseThreadLocks (JNukeObj * this, JNukeObj * thread)
```

Releases completely all locks belonging to the given thread

JNukeLockManager_removeMilestone

```
void removeMilestone (JNukeObj * this)
```

Removes the current milestone.

JNukeLockManager_rollback

```
void rollback (JNukeObj * this)
```

Backs up the state of any lock.
Note: call this rollback prior to the rollback of the heap manager. Otherwise, it may happen that an object is removed from heap and this rollback attempts to write to an object that has been deleted just before.

JNukeLockManager_setMilestone

```
void setMilestone (JNukeObj * this)
```

Sets a milestone which means that JNukeLock_setMilestone is called at any lock.

A.8 JNukeRBCInstruction

JNukeRBCInstruction is a static class providing handlers for register byte codes. The runtime environment delegates the execution of byte codes to this class.

JNukeRBCInstruction_executeALoad

```
enum JNukeExecutionFailure executeALoad (JNukeXByteCode * xbc, int *pc,
JNukeRegister regs[], JNukeObj * rtenv)
```

Executes an array store operation.
in:
xbc – xbytecode
pc – current pc
regs – registers
rtenv – runtime environment

out:
Possible JNukeExecutionFailure value:
– none
– null_pointer_exception
– array_index_out_of_bound_exception

JNukeRBCInstruction_executeAStore

```
enum JNukeExecutionFailure executeAStore (JNukeXByteCode * xbc, int *pc,
JNukeRegister regs[], JNukeObj * rtenv)
```

Executes an array store operation
in:
xbc – xbytecode
pc – current pc
regs – registers
rtenv – runtime environment

out:
Possible JNukeExecutionFailure value:
– none
– null_pointer_exception
– array_index_out_of_bound_exception

JNukeRBCInstruction_executeArrayLength

```
enum JNukeExecutionFailure executeArrayLength (JNukeXByteCode * xbc,
int *pc, JNukeRegister regs[], JNukeObj * rtenv)
```

Executes the Java operator length applied to an array
in:
xbc – xbytecode
pc – current pc
regs – registers
rtenv – runtime environment

out:

JNukeExecutionFailure value (either none or null_pointer_exception)

JNukeRBCInstruction_executeAthrow

```
enum JNukeExecutionFailure executeAthrow (JNukeXByteCode * xbc, int *pc,
JNukeRegister regs[], JNukeObj * rtenv)
```

Executes a throw operation.
in:
xbc – xbytecode
pc – current pc
regs – registers
rtenv – runtime environment

out:
JNukeExecutionFailure value: none or null_pointer_exception

JNukeRBCInstruction_executeCheckcast

```
enum JNukeExecutionFailure executeCheckcast (JNukeXByteCode * xbc, int
*pc, JNukeRegister regs[], JNukeObj * rtenv)
```

Executes a check cast operation.
xbc – xbytecode
pc – current pc
regs – registers
rtenv – runtime environment

out:
JNukeExecutionFailure value (either none or class_cast_exception)

JNukeRBCInstruction_executeCond

```
enum JNukeExecutionFailure executeCond (JNukeXByteCode * xbc, int *pc,
JNukeRegister regs[], JNukeObj * rtenv)
```

Executes a condition operation. If the branch is taken, the program counter is accordingly set to the target.
in:
xbc – xbytecode
pc – current pc
regs – registers
rtenv – runtime environment

out:
JNukeExecutionFailure value "none"

JNukeRBCInstruction_executeConst

`enum JNukeExecutionFailure executeConst (JNukeXByteCode * xbc, int *pc, JNukeRegister regs[], JNukeObj * rtenv)`

Executes a constant value assignment.
in:
xbc – xbytecode
pc – current pc
regs – registers
rtenv – runtime environment

out:
JNukeExecutionFailure value that is always "none"

JNukeRBCInstruction_executeGetField

`enum JNukeExecutionFailure executeGetField (JNukeXByteCode * xbc, int *pc, JNukeRegister regs[], JNukeObj * rtenv)`

Executes a load field operation.
in:
xbc – xbytecode
pc – current pc
regs – registers
rtenv – runtime environment

out:
Possible JNukeExecutionFailure value:
– none
– null_pointer_exception
– no_such_field_error

JNukeRBCInstruction_executeGetStatic

`enum JNukeExecutionFailure executeGetStatic (JNukeXByteCode * xbc, int *pc, JNukeRegister regs[], JNukeObj * rtenv)`

Executes a get static field operation.
in:
xbc – xbytecode
pc – current pc
regs – registers
rtenv – runtime environment

out:
Possible JNukeExecutionFailure value:
– none
– no_such_field_error

JNukeRBCInstruction_executeInstanceof

```
enum JNukeExecutionFailure executeInstanceof (JNukeXByteCode * xbc, int
*pc, JNukeRegister regs[], JNukeObj * rtenv)
```

Executes a instanceof operation.
Instanceof calls Checkcast. The difference between these two operations is the behaviour on null pointers. InstanceOf writes false into the result register und Checkcast succeeds. Operation instanceOf does not throw any cast failure exception. It writes either true or false into the result register.
in:
xbc – xbytecode
pc – current pc
regs – registers
rtenv – runtime environment

out:
JNukeExecutionFailure value (always none)

JNukeRBCInstruction_executeInvokeSpecial

```
enum JNukeExecutionFailure executeInvokeSpecial (JNukeXByteCode * xbc,
int *pc, JNukeRegister regs[], JNukeObj * rtenv)
```

Executes an invocation of a special method.
in:
xbc – xbytecode
pc – current pc
regs – registers
rtenv – runtime environment

out:
JNukeExecutionFailure value (either none or no_such_method_error)

JNukeRBCInstruction_executeInvokeStatic

```
enum JNukeExecutionFailure executeInvokeStatic (JNukeXByteCode * xbc,
int *pc, JNukeRegister regs[], JNukeObj * rtenv)
```

Executes an invocation of a static method.
in:
xbc – xbytecode
pc – current pc
regs – registers
rtenv – runtime environment

out:
JNukeExecutionFailure value (either none or no_such_method_error)

JNukeRBCInstruction_executeInvokeVirtual

```
enum JNukeExecutionFailure executeInvokeVirtual (JNukeXByteCode * xbc,
int *pc, JNukeRegister regs[], JNukeObj * rtenv)
```

Executes an invocation of a virtual method.
in:
xbc – xbytecode
pc – current pc
regs – registers
rtenv – runtime environment

out:
JNukeExecutionFailure value (either none or no_such_method_error)

JNukeRBCInstruction_executeMonitorEnter

```
enum JNukeExecutionFailure executeMonitorEnter (JNukeXByteCode * xbc,
int *pc, JNukeRegister regs[], JNukeObj * rtenv)
```

Executes a MonitorEnter operation.
in:
xbc – xbytecode
pc – current pc
regs – registers
rtenv – runtime environment

out:
JNukeExecutionFailure value (either none, null_pointer_exception,
or failed)

JNukeRBCInstruction_executeMonitorExit

```
enum JNukeExecutionFailure executeMonitorExit (JNukeXByteCode * xbc,
int *pc, JNukeRegister regs[], JNukeObj * rtenv)
```

Executes a MonitorExit operation.
in:
xbc – xbytecode
pc – current pc
regs – registers
rtenv – runtime environment

out:
JNukeExecutionFailure value (either none, null_pointer_exception
or failed)

JNukeRBCInstruction_executeNew

```
enum JNukeExecutionFailure executeNew (JNukeXByteCode * xbc, int *pc,
JNukeRegister regs[], JNukeObj * rtenv)
```

Executes a new operation (used for creation of objects).
in:
xbc – xbytecode
pc – current pc
regs – registers
rtenv – runtime environment

out:
JNukeExecutionFailure value "none"

JNukeRBCInstruction_executeNewArray

```
enum JNukeExecutionFailure executeNewArray (JNukeXByteCode * xbc, int
*pc, JNukeRegister regs[], JNukeObj * rtenv)
```

Executes the newArray operation.
in:
xbc – xbytecode
pc – current pc
regs – registers
rtenv – runtime environment

out:
JNukeExecutionFailure value "none"

JNukeRBCInstruction_executePrim

```
enum JNukeExecutionFailure executePrim (JNukeXByteCode * xbc, int *pc,
JNukeRegister regs[], JNukeObj * rtenv)
```

Executes a primitive operation. This can be an operation as follows: add, sub, mul, div, neg, mod, shift and logical ops, or cmp.
in:
xbc – xbytecode
pc – current pc
regs – registers
rtenv – runtime environment

out:
JNukeExecutionFailure value. If a division by zero has been encountered division_by_zero is returned. Otherwise, the result is "none".

JNukeRBCInstruction_executePutField

```
enum JNukeExecutionFailure executePutField (JNukeXByteCode * xbc, int
*pc, JNukeRegister regs[], JNukeObj * rtenv)
```

Executes a store field operation
in:
xbc – xbytecode
pc – current pc
regs – registers
rtenv – runtime environment

out:
Possible JNukeExecutionFailure value:
– none
– null_pointer_exception
– no_such_field_error

JNukeRBCInstruction_executePutStatic

```
enum JNukeExecutionFailure executePutStatic (JNukeXByteCode * xbc, int
*pc, JNukeRegister regs[], JNukeObj * rtenv)
```

Executes a put static field operation.
in:
xbc – xbytecode
pc – current pc
regs – registers
rtenv – runtime environment

out:
Possible JNukeExecutionFailure value:
– none
– no_such_field_error

JNukeRBCInstruction_executeReturn

```
enum JNukeExecutionFailure executeReturn (JNukeXByteCode * xbc, int *pc,
JNukeRegister regs[], JNukeObj * rtenv)
```

Executes a Return operation.
in:
xbc – xbytecode
pc – current pc
regs – registers
rtenv – runtime environment

out:
JNukeExecutionFailure value (either none or failed)

JNukeRBCInstruction_executeSwitch

```
enum JNukeExecutionFailure executeSwitch (JNukeXByteCode * xbc, int *pc,
JNukeRegister regs[], JNukeObj * rtenv)
```

Executes a tableswitch or lookupswitch operation
in:
xbc – xbytecode
pc – current pc
regs – registers
rtenv – runtime environment
out:
Sets the according target offset. It returns always "none"

JNukeRBCInstruction_extractContentType

```
JNukeObj * extractContentType (const char *arrayType, int dim, JNukeObj
* classPool)
```

For instance: [[[[LMyClass; —> MyClass

JNukeRBCInstruction_getClass

```
JNukeObj * getClass (JNukeObj * classPool, JNukeObj * classString)
```

Finds the JNukeClass instance that fits to the class string
in:
class – (UCSString)

out:
JNukeClass instance (NULL if failed)

JNukeRBCInstruction_getJavaLangString

```
void getJavaLangString (JNukeRegister regs[], int resReg, int cur_pc,
JNukeObj * string, JNukeObj * rtenv)
```

Returns a java.lang.String according to the declared UCSString. If the string has been used in the Java program yet, the string is taken from a pool. Otherwise, a new instance of java/lang/String is created and delivered.

JNukeRBCInstruction_gotoExceptionHandler

```
void gotoExceptionHandler (void *exception, int *pc, JNukeObj * issuer_method,
JNukeObj * rtenv)
```

Finds an exception handler. If such a handler could be found the runtime environment is set to that point. If a handler could not be found in the current method exitMethod() is performed until a handler could be found. If the top level is reached and no handler was found the exception is printed out such as any VM it does.

JNukeRBCInstruction_isSuper

```
int isSuper (JNukeObj * super, JNukeObj * sub, JNukeObj * classPool)
```

Tests whether super is supertype of sub

A.9 JNukeRLCAnalyzer

JNukeRLCAnalyzer implements a reverse lock chain analyzer which detects lock-cycle deadlocks. This class is either used on top of *ExitBlock* or *ExitBlock-RW*.

JNukeRLCAnalyzer_init

```
void init (JNukeObj * this, JNukeObj * rtenv, JNukeObj * exitBlock)
```

in:
rtenv – reference to the runtime environment
exitBlock – reference to the exit block scheduler

JNukeRLCAnalyzer_new

```
JNukeObj * new (JNukeMem * mem)
```

JNukeRLCAnalyzer_setLog

```
void setLog (JNukeObj * this, FILE * log)
```

Sets the file stream used for logging.

A.10 JNukeRRScheduler

JNukeRRScheduler is a basic scheduler scheduling threads in a round-robin order. It grants each thread a fixed time slice.

JNukeRRScheduler_enableTracking

```
void enableTracking (JNukeObj * this)
```

Enables the tracking of context switches. Each thread context switch will be written to a log which can be retrieved with getSchedule.

JNukeRRScheduler_getSchedule

```
JNukeObj * getSchedule (const JNukeObj * this)
```

Returns the schedule (JNukeSchedule) which contains a complete history of thread context switches.

JNukeRRScheduler_init

```
void init (JNukeObj * this, int maxTTL, JNukeObj * rtenv)
```

Initializes the the scheduler. Method init() registers the scheduler as a listener of the declared runtime environment. Further, init() registers a thread state changed listener. This enables the scheduler to schedule another thread if the current thread comes to an end.
in:
maxTTL – maximum time to live for a running thread
rtenv – JNukeRuntimeEnvironment

JNukeRRScheduler_new

```
JNukeObj * new (JNukeMem * mem)
```

JNukeRRScheduler_setLog

```
void setLog (JNukeObj * this, FILE * log)
```

Sets the log file stream.

A.11 JNukeRuntimeEnvironment

JNukeRuntimeEnvironment holds the runtime environment and drives the execution of register byte code in an execution loop.

JNukeRuntimeEnvironment_addJavaString

```
void addJavaString (JNukeObj * this, JNukeObj * string, void *javaString)
```

Adds a java string into internal pool of java strings.
in:
string – UCSString
javaString – reference to the corresponding java/lang/String object

JNukeRuntimeEnvironment_addOnExecuteListener

```
void addOnExecuteListener (JNukeObj * this, int bc_mask, JNukeObj * listenerObj,
JNukeExecutionListener (l))
```

Registers a listener for a declared set of bytecodes. The listener is notified when such a byte code is scheduled to execute. bc_mask is a bit mask of RBC_xyz_mask values. Note that it is possible to install different listeners for different byte codes.
in:
bc_mask – bitmask of RBC_xyz_mask
l – listener

JNukeRuntimeEnvironment_addOnExecutedListener

```
void addOnExecutedListener (JNukeObj * this, int bc_mask, JNukeObj *
listenerObj, JNukeExecutionListener (l))
```

Registers a listener for a declared set of bytecodes. The listener is notified after execution of such a byte code. bc_mask is a bit mask of RBC_xyz_mask values.
in:
bc_mask – bitmask of RBC_xyz_mask
l – listener

JNukeRuntimeEnvironment_addThreadStateListener

```
void addThreadStateListener (JNukeObj * this, JNukeObj * listenerObj,
JNukeThreadStateChangedListener (l))
```

Registers a listener that get notified when the current running thread changes its thread state. This happens if the current thread has executed wait or join. Also this happens if the current thread could not obtain a lock or has even died.
in:
l – listener

JNukeRuntimeEnvironment_createThread

```
JNukeObj * createThread (JNukeObj * this)
```

Creates a JNuke thread and memorize the thread in the list of threads.
out:
Retruns the reference to the therad (JNukeThread).

JNukeRuntimeEnvironment_exitMethod

```
int exitMethod (JNukeObj * this)
```

Returns from the current method to the caller method and re-loads the context of the caller. Returns 1 if there is a caller method. Otherwise, 0.

JNukeRuntimeEnvironment_findJavaString

```
void * findJavaString (JNukeObj * this, JNukeObj * string)
```

Finds a java string in the pool of java strings.
in:
string – UCSString
out:
Returns a reference to the corresponding java/lang/String object.

JNukeRuntimeEnvironment_getClassPool

```
JNukeObj * getClassPool (JNukeObj * this)
```

Returns the class pool.

JNukeRuntimeEnvironment_getCounter

```
int getCounter (JNukeObj * this)
```

Returns the total number of executed byte codes.

JNukeRuntimeEnvironment_getCurrentLineNumber

```
int getCurrentLineNumber (const JNukeObj * this)
```

Returns the current line number of the executed target program.

JNukeRuntimeEnvironment_getCurrentMethod

```
JNukeObj * getCurrentMethod (const JNukeObj * this)
```

Returns current method.

JNukeRuntimeEnvironment_getCurrentRegisters

```
JNukeRegister * getCurrentRegisters (JNukeObj * this, int *maxStack)
```

Returns the current register set.

JNukeRuntimeEnvironment_getCurrentThread

```
JNukeObj * getCurrentThread (JNukeObj * this)
```

Returns the currently running thread.

JNukeRuntimeEnvironment_getErrorLog

```
FILE * getErrorLog (const JNukeObj * this)
```

Returns the error log stream.

JNukeRuntimeEnvironment_getHeapManager

```
JNukeObj * getHeapManager (JNukeObj * this)
```

Returns the heap manager .

JNukeRuntimeEnvironment_getLockManager

```
JNukeObj * getLockManager (const JNukeObj * this)
```

Returns the reference to the lock manager.

JNukeRuntimeEnvironment_getLog

```
FILE * getLog (const JNukeObj * this)
```

Returns the log stream.

JNukeRuntimeEnvironment_getNumberOfByteCodes

```
int getNumberOfByteCodes (JNukeObj * this)
```

Returns the number of byte codes of the current method.

JNukeRuntimeEnvironment_getPC

```
int getPC (const JNukeObj * this)
```

Returns the program counter.

JNukeRuntimeEnvironment_getThreads

```
JNukeObj * getThreads (const JNukeObj * this)
```

Returns a vector of existing threads.

JNukeRuntimeEnvironment_getVMState

```
void getVMState (JNukeObj * this, JNukeObj * vmstate)
```

Writes the current state of the virtual machine into the second argument that has to be a valid instance of JNukeVMState (see vmstate.c).

JNukeRuntimeEnvironment_getWaitSetManager

```
JNukeObj * getWaitSetManager (JNukeObj * this)
```

Returns the wait set manager.

JNukeRuntimeEnvironment_init

```
int init (JNukeObj * this, JNukeObj * heapMgr, JNukeObj * classPool)
```

Sets up the environment. Must be called after creation of an environment. init() creates the main thread, executes any <clinit> methods, and looks up for a main method. If a main method could be found this method is set as the current method. Otherwise, the init() fails and returns zero. The user may call run() after this to start the previously found main() method.

JNukeRuntimeEnvironment_interrupt

```
void interrupt (JNukeObj * this)
```

Interrupts the excution loop immediately.

JNukeRuntimeEnvironment_loadDefaultClasses

`int loadDefaultClasses (JNukeObj * this, JNukeObj * classPool, const char *classpath)`

This method loads some default classes which are used to print out boolean values such that test cases are at least able to print success or failure behaviour.

JNukeRuntimeEnvironment_loadUserClasses

`int loadUserClasses (JNukeObj * this, JNukeObj * classPool, const char *classpath, int n, const char **extraClasses)`

Loads user classes given as argument.
in:
classPool – reference to the class pool.
classpath – path to the user classes
n – number of extra user classes to load
extraClasses – array of additional user classes to load (absolut paths)

JNukeRuntimeEnvironment_new

`JNukeObj * new (JNukeMem * mem)`

JNukeRuntimeEnvironment_removeMilestone

`int removeMilestone (JNukeObj * this)`

Removes the last milestone.

JNukeRuntimeEnvironment_rollback

`int rollback (JNukeObj * this)`

Backs up the last stored state of the virtual machine.

JNukeRuntimeEnvironment_run

`int run (JNukeObj * this)`

Executes the current method from current PC. One usually has to call init() or setMethod() prior to call run(). Otherwise, an assertion is thrown since no byte code is to execute.

JNukeRuntimeEnvironment_setErrorLog

`void setErrorLog (JNukeObj * this, FILE * file)`

Sets the error log stream.
in:
file – file stream

JNukeRuntimeEnvironment_setLog

```
void setLog (JNukeObj * this, FILE * file)
```

Sets the log stream.
in:
file – file stream

JNukeRuntimeEnvironment_setMethod

```
JNukeObj * setMethod (JNukeObj * this, JNukeObj * method)
```

Sets declared method as the current method. Returns a new stack frame used for this method.

JNukeRuntimeEnvironment_setMilestone

```
void setMilestone (JNukeObj * this)
```

Sets a milestone which saves the whole state of the virtual machine.

JNukeRuntimeEnvironment_setPC

```
void setPC (JNukeObj * this, int pc)
```

Sets the program counter manually.

JNukeRuntimeEnvironment_setScheduler

```
void setScheduler (JNukeObj * this, JNukeObj * scheduler)
```

Sets the scheduler.

JNukeRuntimeEnvironment_switchThread

```
void switchThread (JNukeObj * this, JNukeObj * nextThread)
```

Safes the context and performs a thread context switch.
in:
nextThread – Thread instance switch to

A.12 JNukeSchedule

JNukeSchedule is a container recording thread switches during execution used for later replay or further analysis. JNukeRRScheduler and JNukeExitBlock use this class for this purpose.

JNukeSchedule_append

```
void append (JNukeObj * this, JNukeObj * vmstate, JNukeObj * next_thread)
```

Appends a new context switch according the arguments.
in:
vmstate – a valid reference to a vmstate (see rtenv.c and vmstate.c)
next_thread – depicts the next thread that is going to be scheduled

JNukeSchedule_clear

```
void clear (JNukeObj * this)
```

Clears the history.

JNukeSchedule_concat

```
void concat (JNukeObj * this, JNukeObj * sched)
```

Concats two schedules where entries of the second schedule are copyied.

JNukeSchedule_count

```
int count (const JNukeObj * this)
```

Returns the number of schedule entries.

JNukeSchedule_get

```
JNukeContextSwitchInfo * get (const JNukeObj * this, int index)
```

Returns the schedule entry at given index.

JNukeSchedule_getHistory

```
JNukeIterator getHistory (const JNukeObj * this)
```

Returns an iterator to the list of context switches.

JNukeSchedule_new

```
JNukeObj * new (JNukeMem * mem)
```

A.13 JNukeStackFrame

JNukeStackFrame holds the register set of the according method context.

JNukeStackFrame_decRefCounter

```
int decRefCounter (JNukeObj * this)
```

Decrements the reference counter.

JNukeStackFrame_getMaxLocals

```
int getMaxLocals (const JNukeObj * this)
```

Returns the number of local variables in the method of this stackframe.

JNukeStackFrame_getMaxStack

```
int getMaxStack (const JNukeObj * this)
```

Returns the maximum stack height which corresponds to the number of registers.

JNukeStackFrame_getMethodDesc

```
JNukeObj * getMethodDesc (const JNukeObj * this)
```

Returns the method descriptor.

JNukeStackFrame_getMonitorLock

```
JNukeObj * getMonitorLock (const JNukeObj * this)
```

If the method of this stackframe is a synchronized method, getMonitorLock returns the corresponding lock.

JNukeStackFrame_getRefCounter

```
int getRefCounter (const JNukeObj * this)
```

Returns the reference counter.

JNukeStackFrame_getRegisters

```
JNukeRegister * getRegisters (const JNukeObj * this)
```

Returns the register set.

JNukeStackFrame_getResultRegister

```
void getResultRegister (JNukeObj * this, int *resReg, int *resLen)
```

Returns the result register and its length.

JNukeStackFrame_getReturnPoint

```
int getReturnPoint (const JNukeObj * this)
```

Returns the return point. Returns -1 iff no return point is defined.

JNukeStackFrame_new

```
JNukeObj * new (JNukeMem * mem)
```

JNukeStackFrame_removeMilestone

`int removeMilestone (JNukeObj * this)`

Removes the last milestone.

JNukeStackFrame_rollback

`int rollback (JNukeObj * this)`

Backs up a stackframe state. Returns 1 if there was at least one milestone remaining. Otherwise, rollback() returns with 0.

JNukeStackFrame_setMethodDesc

`void setMethodDesc (JNukeObj * this, JNukeObj * methoddesc)`

Sets the method description. The register set is created according the number of registers and locals from the descriptor.

JNukeStackFrame_setMilestone

`void setMilestone (JNukeObj * this)`

Sets a milestone at the current stack frame. The register set is cloned and copyied to the stack.

JNukeStackFrame_setMonitorLock

`void setMonitorLock (JNukeObj * this, JNukeObj * lock)`

Sets the monitor lock, called when the virtual machine enters a synchronized method.

JNukeStackFrame_setResultRegister

`void setResultRegister (JNukeObj * this, int resReg, int resLen)`

Sets the result register and its length.

JNukeStackFrame_setReturnPoint

`void setReturnPoint (JNukeObj * this, int rp)`

Sets the return point.

A.14 JNukeThread

JNukeThread stores information about the state of a Java thread (call stack, flags, etc.). Each Java thread is assigned to one JNukeThread instance.

JNukeThread_addLock

`void addLock (JNukeObj * this, JNukeObj * lock)`

Adds a lock.

JNukeThread_canReacquireLocks

```
int canReacquireLocks (JNukeObj * this)
```

If a thread sleeps because of an invocation of wait(), the thread has to check whether it is able to reacquire its locks. This method returns 1 if this is possible. Otherwise, the result is 0 which means that the current thread has to go to sleep again.

JNukeThread_createStackFrame

```
JNukeObj * createStackFrame (JNukeObj * this, JNukeObj * method)
```

Creates a new stack frame on top of the call stack. Returns the stack frame.

JNukeThread_decLockCounter

```
int decLockCounter (JNukeObj * this)
```

Decrements the lock counter.

JNukeThread_getCurrentMethod

```
JNukeObj * getCurrentMethod (const JNukeObj * this)
```

Returns the method of the top stack frame.

JNukeThread_getCurrentRegisters

```
JNukeRegister * getCurrentRegisters (const JNukeObj * this)
```

Returns a reference to the current register set.

JNukeThread_getCurrentStackFrame

```
JNukeObj * getCurrentStackFrame (const JNukeObj * this)
```

Returns the top stack frame.

JNukeThread_getJavaThread

```
void * getJavaThread (const JNukeObj * this)
```

Returns the memory reference to the real Java instance of java/lang/Thread.

JNukeThread_getLastHeldLock

```
JNukeObj * getLastHeldLock (const JNukeObj * this)
```

Returns the last lock held.

JNukeThread_getLocks

```
JNukeIterator getLocks (const JNukeObj * this)
```

Returns a vector of locks owned by this thread.

JNukeThread_getNumberOfLocks

`int getNumberOfLocks (const JNukeObj * this)`

Returns the number of locks owned by this thread.

JNukeThread_getPC

`int getPC (const JNukeObj * this)`

Returns the current program counter of this thread.

JNukeThread_getPos

`int getPos (const JNukeObj * this)`

Returns the position of this thread in the runtime environement's thread vector.

JNukeThread_getStackLevel

`int getStackLevel (const JNukeObj * this)`

Returns the height of the call stack.

JNukeThread_incLockCounter

`int incLockCounter (JNukeObj * this)`

Increments the number of locks owned by this thread.

JNukeThread_interrupt

`void interrupt (JNukeObj * this)`

Interrupts a thread. The interrupt flag is set to true. If another thread has joined this thread InterruptedException is thrown.

JNukeThread_isAlive

`int isAlive (const JNukeObj * this)`

Says whether the thread is alive.

JNukeThread_isInterrupted

`int isInterrupted (const JNukeObj * this, int clearFlag)`

Says whether the thread has been interrupted.

JNukeThread_isReadyToRun

`int isReadyToRun (const JNukeObj * this)`

Returns the value of the flag readyToRun.

JNukeThread_isWaiting

`int isWaiting (const JNukeObj * this)`

Returns the value of the flag waiting.

JNukeThread_isYielded

`int isYielded (const JNukeObj * this, int clearFlag)`

Returns 1 if the current thread is yielded. Otherwise, 0.

JNukeThread_join

`void join (JNukeObj * this, JNukeObj * thread)`

Adds a thread to the join list of this thread.

JNukeThread_new

`JNukeObj * new (JNukeMem * mem)`

JNukeThread_pendingInterruptedException

`int pendingInterruptedException (const JNukeObj * this, int clear)`

Returns 1, if the current thread has to handle a InterruptedException. A sleeping thread that has invoked either join or wait, has to check this after being rescheduled.

JNukeThread_popStackFrame

`JNukeObj * popStackFrame (JNukeObj * this)`

Removes the top stackframe and returns the pointer to this stackframe.

JNukeThread_reacquireLocks

`int reacquireLocks (JNukeObj * this)`

Has to be called prior to schedule a thread in order that locks that need to be locked can be acquired again. If the locks could be acquired the result is 1. Otherwise, the result is 0 and the thread is put back to the wait set of this lock.

JNukeThread_removeLock

`void removeLock (JNukeObj * this, JNukeObj * lock)`

Removes a lock from the thread's lock list.

JNukeThread_removeMilestone

`int removeMilestone (JNukeObj * this)`

Removes the last milestone.

JNukeThread_rollback

```
int rollback (JNukeObj * this)
```

Backs up the the thread state. If there is at least one milestone this method returns with 1. Otherwise, 0.

JNukeThread_setAlive

```
void setAlive (JNukeObj * this, int alive)
```

Sets the flag alive. If the flag is set to 0 all joined threads are woken up.

JNukeThread_setJavaThread

```
void setJavaThread (JNukeObj * this, void *thread)
```

Assigns this thread a real Java thread instance.

JNukeThread_setJoining

```
void setJoining (JNukeObj * this)
```

Sets the flag joining to true.

JNukeThread_setMilestone

```
void setMilestone (JNukeObj * this)
```

Sets a milestone for this thread and for each stackframe which belongs to this thread.

JNukeThread_setPC

```
void setPC (JNukeObj * this, int pc)
```

Sets the current program counter.

JNukeThread_setPos

```
void setPos (JNukeObj * this, int pos)
```

The runtime environment keeps a vector of threads. The position in this vector is unique. Method setPos is called by the runtime environment in order to inform the thread about its position in this vector. This position can be used as thread identifier.

JNukeThread_setReadyToRun

```
void setReadyToRun (JNukeObj * this, int readyToRun)
```

Sets this thread readyToRun. Usually, this succeeds and the return value 1. However, if the thread is sleeping and is not able to relock its locks the method returns with 0 and the thread is not set ready to run.

JNukeThread_setRuntimeEnvironment

```
void setRuntimeEnvironment (JNukeObj * this, JNukeObj * renv)
```

Sets the runtime environment.

JNukeThread_setWaiting

```
void setWaiting (JNukeObj * this, void *object, int n)
```

Sets this thread waiting.
in:
object – object on which wait() was performed.
n – number of times the object lock was acquired.

JNukeThread_yield

```
void yield (const JNukeObj * this)
```

Yields a thread, which means, that the current thread gives other threads the chance to run.

A.15 JNukeVirtualTable

JNukeVirtualTable contains a table with method descriptors and references to the actual implementations. Each Java class has a JNukeVirtualTable assigned to its instance descriptor (JNukeInstanceDesc). Class JNukeVirtualTable provides methods for resolution of virtual, static, and special methods.

JNukeVirtualTable_build

```
void build (JNukeObj * this, JNukeObj * class, JNukeObj * clPool)
```

Builds a virtual table out of the declared class. Called once for each class.
in:
class – (JNukeClass)

JNukeVirtualTable_finalize

```
void finalize (JNukeObj * this, JNukeObj * vtables)
```

Completes the vtable by inserting each method of any super vtables into the local method map
in:
vtables – Vector of all existing vtables

JNukeVirtualTable_findSpecial

```
JNukeObj * findSpecial (const JNukeObj * this, JNukeObj * func, int *isNative)
```

Find a special method by its complete signature.
in:
func – the method desriptor

out:
A pointer to JNukeMethod that points to the according method
If no corresponding method could be found the method fails. That
means NULL is returned

JNukeVirtualTable_findSpecialByName

```
JNukeObj * findSpecialByName (const JNukeObj * this, const char *class_name,
const char *method_name, const char *signature, int *native)
```

Finds a special method described by a class name, method name, and a signature. This
method is used if no method descriptor is available.
in:
class_name – char buffer
method_name – char buffer
signature – char buffer

out:
Returns a pointer to JNukeMethod that points to the according method
If no corresponding method could be found the method fails. That
means NULL is returned

JNukeVirtualTable_findVirtual

```
JNukeObj * findVirtual (const JNukeObj * this, JNukeObj * func, int *isNative)
```

Finds a virtual machine by its complete signature.
in:
func – the method descriptor

out:
Returns a pointer to JNukeMethod that points to the according method
If no corresponding method could be found the method fails. That
means NULL is returned.

JNukeVirtualTable_findVirtualByName

```
JNukeObj * findVirtualByName (const JNukeObj * this, const char *method_name,
const char *signature, int *native)
```

Finds a virtual method described by a method name and a signature. This method is
used if no method descriptor is available.

in:
method_name – char buffer
signature – char buffer

out:
Returns a pointer to JNukeMethod that points to the according method
If no corresponding method could be found the method fails. That
means NULL is returned

JNukeVirtualTable_getClass

```
JNukeObj * getClass (const JNukeObj * this)
```

Returns the class description assigned with this virtual table.

JNukeVirtualTable_getMethods

```
JNukeObj * getMethods (const JNukeObj * this)
```

Returns the map of methods.

JNukeVirtualTable_new

```
JNukeObj * new (JNukeMem * mem)
```

A.16 JNukeVMState

This class holds a snapshot of the state of the VM. At the moment, this is the current method, program counter, and the current line. This class can be extended (or subclassed) if necessary (current register set, current set of threads, ...). A `JNukeVMState` instance may be used for storing historical VM states for logging or reporting of interesting states.

Usage

Creation of a snapshot of the current VM:

```
   vmstate = JNukeVMState_new (this->mem);
   JNukeVMState_snapshot (vmstate, rtenv);
```

JNukeVMState_getCounter

```
int getCounter (const JNukeObj * this)
```

Returns the byte code counter

JNukeVMState_getCurrentThreadId

```
int getCurrentThreadId (const JNukeObj * this)
```

Returns the id of the current thread

JNukeVMState_getLineNumber

```
int getLineNumber (const JNukeObj * this)
```

Returns the line number

JNukeVMState_getMethod

```
JNukeObj * getMethod (const JNukeObj * this)
```

Returns the method that was being executed at the time when the snapshot was taken.

JNukeVMState_getPC

```
int getPC (const JNukeObj * this)
```

Returns the program counter

JNukeVMState_new

```
JNukeObj * new (JNukeMem * mem)
```

JNukeVMState_setCounter

```
void setCounter (JNukeObj * this, int counter)
```

Sets the byte code counter

JNukeVMState_setCurrentThreadId

```
void setCurrentThreadId (JNukeObj * this, int threadId)
```

Sets the id of the current thread

JNukeVMState_setLineNumber

```
void setLineNumber (JNukeObj * this, int lineNumber)
```

Sets the line number

JNukeVMState_setMethod

```
void setMethod (JNukeObj * this, JNukeObj * method)
```

Sets the method of this vmstate.

JNukeVMState_setPC

`void setPC (JNukeObj * this, int pc)`

Sets the program counter

JNukeVMState_snapshot

`void snapshot (JNukeObj * this, JNukeObj * rtenv)`

Takes a snapshot of the current runtime environment

A.17 JNukeWaitList

Each Java instance is assigned to a wait list. Threads that call `wait` on an instance are inserted into the wait list of the instance. A thread in a wait list lies dormant until it is either awakened by `notify` or `notifyAll`.

JNukeWaitList_count

`int count (JNukeObj * this)`

Removes the number of thread in the wait list.

JNukeWaitList_insert

`void insert (JNukeObj * this, JNukeObj * thread)`

Inserts a thread into the wait list (and sets the readyToRun flag to 0)

JNukeWaitList_new

`JNukeObj * new (JNukeMem * mem)`

JNukeWaitList_removeMilestone

`void removeMilestone (JNukeObj * this)`

Removes the last milestone.

JNukeWaitList_resumeAll

`void resumeAll (JNukeObj * this)`

Wakes all threads in the wait list up (which means that they become ready to run). Threads woken up are removed from the wait list.

JNukeWaitList_resumeNext

`JNukeObj * resumeNext (JNukeObj * this)`

Wakes up the thread sleeping for the longest time

JNukeWaitList_rollback

`void rollback (JNukeObj * this)`

Backs up the last state of the wait list.

JNukeWaitList_setMilestone

`void setMilestone (JNukeObj * this)`

Sets a milestone.

A.18 JNukeWaitsetManager

Class JNukeWaitsetManager manages all wait lists. Its interface provides among other things the three methods `wait`, `notify`, and `notifyAll`.

JNukeWaitSetManager_new

`JNukeObj * new (JNukeMem * mem)`

JNukeWaitSetManager_notify

`JNukeObj * notify (void *object)`

Wakes up the next sleeping thread from the wait list assigned to this object

JNukeWaitSetManager_notifyAll

`void notifyAll (void *object)`

Awakens all sleeping threads that are in the wait set assigned to this object.

JNukeWaitSetManager_removeMilestone

`void removeMilestone (JNukeObj * this)`

Removes the last milestone.

JNukeWaitSetManager_rollback

`void rollback (JNukeObj * this)`

Backs up to the last state of the wait set manager.

JNukeWaitSetManager_setMilestone

`void setMilestone (JNukeObj * this)`

Sets a milestone.

JNukeWaitSetManager_wait

```
int wait (JNukeObj * this, void *object, JNukeObj * thread)
```

Performs a wait operations. This means that the current thread releases the object lock completely and is going to sleep in the wait queue. If the this thread is not the owner of the lock, which is illegal according the Java spec, this method returns with 0. Otherwise, 1.

Appendix B

Code Examples

B.1 JNukeHeapManagerActionEvent

Listing B.1 : The struct `JNukeHeapManagerActionEvent`

```
struct JNukeHeapManagerActionEvent
{
  /** the heap manager that has issued this event */
  JNukeObj *issuer;

  /** base pointer to the instance */
  void *instance;

  /** the offset to the field. addr = instance + offset */
  int offset;

  /** size of the field (4 or 8 bytes) */
  int size;

  /** reference to the instance descriptor */
  JNukeObj *instanceDesc;

  /** name of the class. NULL for arrays */
  JNukeObj *class;

  /** name of the field. NULL for arrays */
  JNukeObj *field;
};
```

B.2 MethodInvocation

Listing B.2 : `MethodInvocation.java`: performs many method invocations

```
class MethodInvocation {
  int foo0(int i) { return i + 1; }
  int foo1(int i) { return i + 1; }
  int foo2(int i) { return i + 1; }
  int foo3(int i) { return i + 1; }
  ...
  int foo499(int i) { return i + 1; }
  public static void main(String[] args) {
    Test7 t = new Test7();
    int j = 0;
    int i;

    for (i=0; i<4000; i++)
    {
      j += t.foo0( i );
      j += t.foo1( i );
      ...
      j += t.foo499( i );
    }
  }
}
```

B.3 ReadManyFields

Listing B.3 : `ReadManyFields.java`: performs read and write field accesses

```
class ReadManyFields {
  static long a;
  static long a1;
  static long a2;
  ...
  static long a5000;

  public static void main(String[] args) {
    for (int i=0; i<200; i++)
    {
      a = a + 4;
      a1 = a1 + 4;
      a2 = a2 + 4;
      ...
      a5000 = a5000 + 4;
    }
  }
}
```

B.4 Iteration

Listing B.4 : `Iteration.java`: a loop with 100’000’000 iterations

```
int i;
for (i = 0; i < 100000000; i++)
{
}
```

B.5 Array

Listing B.5 : `Array.java`: iteration over an array of 10’000’000 elements

```java
int i,j;
int a[] = new int[10000000];

for (j=0; j < 10; j++)
{
  for (i=0; i < 10000000; i++)
  {
    a[i] = i + j;
  }
}
```

B.6 MultiArray

Listing B.6 : `MultiArray1.java`: iteration of a six dimensional array

```java
int a,b,c,d,e,f;
int array[][][][][][] = new int[10][10][10][10][10][10];

for (a=0; a < 10; a++) {
  for (b=0; b < 10; b++) {
    for (c=0; c < 10; c++) {
      for (d=0; d < 10; d++) {
        for (e=0; e < 10; e++) {
          for (f=0; f < 10; f++) {
            array[a][b][c][d][e][f] = a + b + c + d + e + f;
          }
        }
      }
    }
  }
}
```

B.7 DoubleOp

Listing B.7 : `DoubleOp.java`: double operations performed in a loop

```java
public class DoubleOp {
  double a,b,c,d,e,f;

  void go() {
    int i;

    for (i=0; i < 1000000 * 2; i++) {
      a = i * 0.002 + c;
      b = (float) a * (float) 2.0;
      c = (b - a - b) * i;
      d = b + a;
      e = 3.141 * d + a + b;
      f = 0.0;
    }
  }

  public static void main(String[] args) {
    Test8 test = new Test8();
    test.go();
  }

}
```

B.8 BubbleSort

Listing B.8 : `BubbleSort.java`: an example implementation of bubble sort.

```
class BubbleSort {

  public static void main(String[] args) {
    int max = 10000;
    int i, j, t, n = max - 1;
    boolean s = false;
    int[] a = new int[max];

    for ( i = 0; i < max; i++ )
      {
        a[i] = max - i;
      }

    for (i = 0; i < n; i++)
      {
        s = false;
        for (j=n; j>i; j--)
        {
          if ( a[j] < a[j-1] )
          {
            t = a[j];
            a[j] = a[j-1];
            a[j-1] = t;
            s = true;
          }
        }

        if (!s)
        break;
      }

    boolean res = true;
    for ( i = 0; i < max; i++ )
      {
         res = res && a[i] == i + 1;
      }

    System.out.println(res);
  }

}
```

B.9 JASPA

Listing B.9 : MccaJaspa.java: Jaspa benchmark with matrix mcca

```
public class MccaJaspa{
   public static final double CPU_MIN = 2.0;

   public static void main (String[] args) throws Exception
   {
      int nrun, nz, n;
      long endTime;
      long startTime = 0;
      int[] irn;
      int[] jcn;
      double[] val;

      n = 180;
      nz = 2659;
      irn = irn_values;
      jcn = jcn_values;
      val = new double[nz];
      for (int j = 0; j < nz; j++) {
         val[j] = 1.0;
      }

      int[] ia = new int[n+2]; //extra space for FORTRAN style
      int[] ja = new int[nz+1]; //extra space for FORTRAN style
      double[] a = new double[nz+1]; //extra space for FORTRAN style
      ijval2csr(n,nz,irn,jcn,val,ia,ja,a);
      int m = n;

      int nz1;
      double[] c;
      for (nrun = 1;;nrun++){
      nz1 = spmatmul_size(n, m, ia,ja, ia, ja);

      // allocate space for the multiplication
      int[] ic = new int[n+2]; //extra space for FORTRAN style
      int[] jc = new int[nz1+1]; //extra space for FORTRAN style
      c = new double[nz1+1]; //extra space for FORTRAN style
      spmatmul_double(n,m,a,ia,ja,a,ia,ja,c,ic,jc);
      if (nrun > 100) break;
      }

      // check the answer: average entry value
      double avg = 0;
      for (int j = 1; j <= nz1; j++) {
      avg = avg + c[j];
      }

   }//end main

   public static void ijval2csr(int n, int nz,
            int[] irn, int[] jcn, double[] val,
            int[] ia, int[] ja, double[] a) {
   int i,ii,jj;

   // convert to compact sparse row format
   int[] nrow = new int[n+2];//extra space for FORTRAN style

   for (i = 1; i <= n+1; i++) {
      nrow[i] = 0;
}
for (i = 0; i < nz; i++) {
   nrow[irn[i]] = nrow[irn[i]] + 1;
}

ia[1] = 1;// [1]: fortran style
for (i = 1; i <= n; i++) {
   ia[i+1] = ia[i] + nrow[i];
}

for (i = 1; i <= n+1; i++) {
   nrow[i] = ia[i];
}

for (i = 0; i < nz; i++) {
   ii = irn[i];
   jj = jcn[i];
   ja[nrow[ii]] = jj;
   a[nrow[ii]] = val[i];
   nrow[ii] = nrow[ii] + 1;
}
}//end method ijval2csr


static int spmatmul_size(int n, int m,
          int[] ia,int[] ja,
          int[] ib,int[] jb)

{
int i,j,k,nz,icol_add;



int[] mask = new int[m+1]; //m=1 rather than m: fortran style

for (i = 1; i <= m; i++) mask[i] = -1; // start from 1: fortran style


nz = 0;

for (i = 1; i <= n; i++) // fortran style
   {
   for (j = ia[i]; j < ia[i+1]; j++)
      {
      int neigh = ja[j];
      for (k = ib[neigh]; k < ib[neigh+1]; k++)
         {
         icol_add = jb[k];
         if (mask[icol_add] != i)
            {
            nz++;
            mask[icol_add] = i; // add mask
            }
         }
      }
   }

return nz;
}

static void spmatmul_double(
       int n, int m,
        double[] a, int[] ia,int[] ja,
        double[] b, int[] ib,int[] jb,
        double[] c, int[] ic,int[] jc)
{
int nz;
int i,j,k,l,icol,icol_add;
double aij;
int neighbour;

int[] mask = new int[m+1]; // extra space for FORTRAN like array indexing

for (l = 1; l <= m; l++) mask[l] = 0; // starting from one for FORTRAN like array index

ic[0] = 1;
nz = 0;
for (i = 1; i <= n; i++) { // starting from one for FORTRAN like array index
   for (j = ia[i]; j < ia[i+1]; j++){
   neighbour = ja[j];
   aij = a[j];
   for (k = ib[neighbour]; k < ib[neighbour+1]; k++){
      icol_add = jb[k];
      icol = mask[icol_add];
      if (icol == 0) {
      jc[++nz] = icol_add;
      c[nz] = aij*b[k];
      mask[icol_add] = nz;
      }
      else {
      c[icol] += aij*b[k];
      }
   }
   }
   for (j = ic[i]; j < nz + 1; j++) mask[jc[j]] = 0;
   ic[i+1] = nz+1;
}

}

static int spmatmul_flops(int n, int m,
          int[] ia,int[] ja,
          int[] ib,int[] jb)

{
int i,j,k,nz,icol_add,flops;



int[] mask = new int[m+1]; //m=1 rather than m: fortran style

for (i = 1; i <= m; i++) mask[i] = -1; // start from 1: fortran style


nz = 0;
flops = 0;

for (i = 1; i <= n; i++) // fortran style
   {
   for (j = ia[i]; j < ia[i+1]; j++)
      {
      int neigh = ja[j];
      for (k = ib[neigh]; k < ib[neigh+1]; k++)
         {
            icol_add = jb[k];
            flops += 2;
            if (mask[icol_add] != i)
               {
               nz++;
               mask[icol_add] = i; // add mask
               }
            }
         }
      }
   flops += nz;
   return flops;
   }


   static int jcn_values[] = { ... }; /* input data */

   static int irn_values[] = { ... }; /* input data */
}//end Spmatmul class
```

B.10 JGFCrypt

Listing B.10 : `JGFCryptBenchSizeA.java`: main class

```
/**************************************************************************
*                                                                         *
*        Java Grande Forum Benchmark Suite - Thread Version 1.0           *
*                                                                         *
*                            produced by                                  *
*                                                                         *
*                  Java Grande Benchmarking Project                       *
*                                                                         *
*                                at                                       *
*                                                                         *
*                Edinburgh Parallel Computing Centre                      *
*                                                                         *
*                email: epcc-javagrande@epcc.ed.ac.uk                     *
*                                                                         *
*                                                                         *
*      This version copyright (c) The University of Edinburgh, 2001.      *
*                         All rights reserved.                            *
*                                                                         *
**************************************************************************/

public class JGFCryptBenchSizeA{

  public static int nthreads;

  public static void main(String argv[]){

    nthreads = 2;

    JGFCryptBench cb = new JGFCryptBench(nthreads);
    cb.JGFrun(0);

  }
}
```

Listing B.11 : `JGFCryptBench.java`: benchmark driver

```
/**************************************************************************
*                                                                         *
*         Java Grande Forum Benchmark Suite - Thread Version 1.0          *
*                                                                         *
*                            produced by                                  *
*                                                                         *
*                  Java Grande Benchmarking Project                       *
*                                                                         *
*                                at                                       *
*                                                                         *
*                Edinburgh Parallel Computing Centre                      *
*                                                                         *
*                email: epcc-javagrande@epcc.ed.ac.uk                     *
*                                                                         *
*                                                                         *
*      This version copyright (c) The University of Edinburgh, 2001.      *
*                         All rights reserved.                            *
*                                                                         *
**************************************************************************/


public class JGFCryptBench extends IDEATest {

  private int size;
  private int datasizes[]={3000000,20000000,50000000};
    public static int nthreads;

public JGFCryptBench(int nthreads)
    {
    this.nthreads=nthreads;
    }


  public void JGFsetsize(int size){
    this.size = size;
  }

  public void JGFinitialise(){
    array_rows = datasizes[size];
    buildTestData();
  }

  public void JGFkernel(){
    Do();
  }

  public void JGFvalidate(){
    boolean error;

    error = false;
    for (int i = 0; i < array_rows; i++){
      error = (plain1 [i] != plain2 [i]);
      if (error){
    System.out.println("Validation␣failed");
    System.out.println("Original␣Byte␣" + i + "␣=␣" + plain1[i]);
    System.out.println("Encrypted␣Byte␣" + i + "␣=␣" + crypt1[i]);
    System.out.println("Decrypted␣Byte␣" + i + "␣=␣" + plain2[i]);
    //break;
      }
    }
  }
  public void JGFtidyup(){
    freeTestData();
  }



  public void JGFrun(int size){

    JGFsetsize(size);
    JGFinitialise();
    JGFkernel();
    JGFvalidate();
    JGFtidyup();
  }
}
```

Listing B.12 : `IDEARunner.java`: worker thread

```
class IDEARunner implements Runnable {

    int id,key[];
    byte text1[],text2[];

    public IDEARunner(int id, byte [] text1, byte [] text2, int [] key) {
    this.id = id;
    this.text1=text1;
    this.text2=text2;
    this.key=key;
    }
/*
* run()
*
* IDEA encryption/decryption algorithm. It processes plaintext in
* 64-bit blocks, one at a time, breaking the block into four 16-bit
* unsigned subblocks. It goes through eight rounds of processing
* using 6 new subkeys each time, plus four for last step. The source
* text is in array text1, the destination text goes into array text2
* The routine represents 16-bit subblocks and subkeys as type int so
* that they can be treated more easily as unsigned. Multiplication
* modulo 0x10001 interprets a zero sub-block as 0x10000; it must to
* fit in 16 bits.
*/

    public void run() {
        int ilow, iupper, slice, tslice, ttslice;

        tslice = text1.length / 8;
        ttslice = (tslice + JGFCryptBench.nthreads-1) / JGFCryptBench.nthreads;
        slice = ttslice*8;

        ilow = id*slice;
        iupper = (id+1)*slice;
        if(iupper > text1.length) iupper = text1.length;

int i1 = ilow;             // Index into first text array.
int i2 = ilow;             // Index into second text array.
int ik;                 // Index into key array.
int x1, x2, x3, x4, t1, t2; // Four "16-bit" blocks, two temps.
int r;                  // Eight rounds of processing.

 for (int i =ilow ; i <iupper ; i +=8)
{

   ik = 0;              // Restart key index.
   r = 8;               // Eight rounds of processing.

   // Load eight plain1 bytes as four 16-bit "unsigned" integers.
   // Masking with 0xff prevents sign extension with cast to int.

   x1 = text1[i1++] & 0xff;      // Build 16-bit x1 from 2 bytes,
   x1 |= (text1[i1++] & 0xff) << 8; // assuming low-order byte first.
   x2 = text1[i1++] & 0xff;
   x2 |= (text1[i1++] & 0xff) << 8;
   x3 = text1[i1++] & 0xff;
   x3 |= (text1[i1++] & 0xff) << 8;
   x4 = text1[i1++] & 0xff;
   x4 |= (text1[i1++] & 0xff) << 8;

   do {
      // 1) Multiply (modulo 0x10001), 1st text sub-block
      // with 1st key sub-block.

      x1 = (int) ((long) x1 * key[ik++] % 0x10001L & 0xffff);

      // 2) Add (modulo 0x10000), 2nd text sub-block
      // with 2nd key sub-block.

      x2 = x2 + key[ik++] & 0xffff;

      // 3) Add (modulo 0x10000), 3rd text sub-block
      // with 3rd key sub-block.

      x3 = x3 + key[ik++] & 0xffff;

      // 4) Multiply (modulo 0x10001), 4th text sub-block
      // with 4th key sub-block.

      x4 = (int) ((long) x4 * key[ik++] % 0x10001L & 0xffff);

      // 5) XOR results from steps 1 and 3.

      t2 = x1 ^ x3;

      // 6) XOR results from steps 2 and 4.
      // Included in step 8.

      // 7) Multiply (modulo 0x10001), result of step 5
      // with 5th key sub-block.

      t2 = (int) ((long) t2 * key[ik++] % 0x10001L & 0xffff);

      // 8) Add (modulo 0x10000), results of steps 6 and 7.

      t1 = t2 + (x2 ^ x4) & 0xffff;

      // 9) Multiply (modulo 0x10001), result of step 8
      // with 6th key sub-block.
```

```
        t1 = (int) ((long) t1 * key[ik++] % 0x10001L & 0xffff);

        // 10) Add (modulo 0x10000), results of steps 7 and 9.

        t2 = t1 + t2 & 0xffff;

        // 11) XOR results from steps 1 and 9.

        x1 ^= t1;

        // 14) XOR results from steps 4 and 10. (Out of order).

        x4 ^= t2;

        // 13) XOR results from steps 2 and 10. (Out of order).

        t2 ^= x2;

        // 12) XOR results from steps 3 and 9. (Out of order).

        x2 = x3 ^ t1;

        x3 = t2;      // Results of x2 and x3 now swapped.

    } while(--r != 0); // Repeats seven more rounds.

    // Final output transform (4 steps).

    // 1) Multiply (modulo 0x10001), 1st text-block
    // with 1st key sub-block.

    x1 = (int) ((long) x1 * key[ik++] % 0x10001L & 0xffff);

    // 2) Add (modulo 0x10000), 2nd text sub-block
    // with 2nd key sub-block. It says x3, but that is to undo swap
    // of subblocks 2 and 3 in 8th processing round.

    x3 = x3 + key[ik++] & 0xffff;

    // 3) Add (modulo 0x10000), 3rd text sub-block
    // with 3rd key sub-block. It says x2, but that is to undo swap
    // of subblocks 2 and 3 in 8th processing round.

    x2 = x2 + key[ik++] & 0xffff;

    // 4) Multiply (modulo 0x10001), 4th text-block
    // with 4th key sub-block.

    x4 = (int) ((long) x4 * key[ik++] % 0x10001L & 0xffff);

    // Repackage from 16-bit sub-blocks to 8-bit byte array text2.

    text2[i2++] = (byte) x1;
    text2[i2++] = (byte) (x1 >>> 8);
    text2[i2++] = (byte) x3;          // x3 and x2 are switched
    text2[i2++] = (byte) (x3 >>> 8);  // only in name.
    text2[i2++] = (byte) x2;
    text2[i2++] = (byte) (x2 >>> 8);
    text2[i2++] = (byte) x4;
    text2[i2++] = (byte) (x4 >>> 8);

}  // End for loop.
```

```
    }  // End routine.

}  // End of class
```

Listing B.13 : `IDETest.java`: IDEA encryption/decryption

```
/**************************************************************************
*                                                                         *
*        Java Grande Forum Benchmark Suite - Thread Version 1.0           *
*                                                                         *
*                            produced by                                  *
*                                                                         *
*                  Java Grande Benchmarking Project                       *
*                                                                         *
*                                at                                       *
*                                                                         *
*                Edinburgh Parallel Computing Centre                      *
*                                                                         *
*                email: epcc-javagrande@epcc.ed.ac.uk                     *
*                                                                         *
*                  Original version of this code by                       *
*                 Gabriel Zachmann (zach@igd.fhg.de)                      *
*                                                                         *
*      This version copyright (c) The University of Edinburgh, 2001.     *
*                         All rights reserved.                            *
*                                                                         *
**************************************************************************/


/**
* Class IDEATest
*
* This test performs IDEA encryption then decryption. IDEA stands
* for International Data Encryption Algorithm. The test is based
* on code presented in Applied Cryptography by Bruce Schnier,
* which was based on code developed by Xuejia Lai and James L.
* Massey.

**/

import java.util.*;

class IDEATest
{

// Declare class data. Byte buffer plain1 holds the original
// data for encryption, crypt1 holds the encrypted data, and
// plain2 holds the decrypted data, which should match plain1
// byte for byte.

int array_rows;

byte [] plain1;    // Buffer for plaintext data.
byte [] crypt1;    // Buffer for encrypted data.
byte [] plain2;    // Buffer for decrypted data.
```

```
short [] userkey;  // Key for encryption/decryption.
int [] Z;          // Encryption subkey (userkey derived).
int [] DK;         // Decryption subkey (userkey derived).



void Do()
{

   Runnable thobjects[] = new Runnable [JGFCryptBench.nthreads];
   Thread th[] = new Thread [JGFCryptBench.nthreads];


  // Encrypt plain1.
  for(int i=0;i<JGFCryptBench.nthreads;i++) {
      thobjects[i] = new IDEARunner(i,plain1,crypt1,Z);
      th[i] = new Thread(thobjects[i]);
      th[i].start();
   }

      // thobjects[0] = new IDEARunner(0,plain1,crypt1,Z);
      // thobjects[0].run();


   for(int i=0;i<JGFCryptBench.nthreads;i++) {
      try {
      th[i].join();
      }
      catch (InterruptedException e) {}
   }

   // Decrypt.
   for(int i=0;i<JGFCryptBench.nthreads;i++) {
      thobjects[i] = new IDEARunner(i,crypt1,plain2,DK);
      th[i] = new Thread(thobjects[i]);
      th[i].start();
   }


   for(int i=0;i<JGFCryptBench.nthreads;i++) {
      try {
      th[i].join();
      }
      catch (InterruptedException e) {}
   }


}

/*
* buildTestData
*
* Builds the data used for the test -- each time the test is run.
*/

void buildTestData()
{


   // Create three byte arrays that will be used (and reused) for
   // encryption/decryption operations.
```

```
    plain1 = new byte [array_rows];
    crypt1 = new byte [array_rows];
    plain2 = new byte [array_rows];


    Random rndnum = new Random(136506717L); // Create random number generator.


    // Allocate three arrays to hold keys: userkey is the 128-bit key.
    // Z is the set of 16-bit encryption subkeys derived from userkey,
    // while DK is the set of 16-bit decryption subkeys also derived
    // from userkey. NOTE: The 16-bit values are stored here in
    // 32-bit int arrays so that the values may be used in calculations
    // as if they are unsigned. Each 64-bit block of plaintext goes
    // through eight processing rounds involving six of the subkeys
    // then a final output transform with four of the keys; (8 * 6)
    // + 4 = 52 subkeys.

    userkey = new short [8]; // User key has 8 16-bit shorts.
    Z = new int [52];     // Encryption subkey (user key derived).
    DK = new int [52];    // Decryption subkey (user key derived).

    // Generate user key randomly; eight 16-bit values in an array.

    for (int i = 0; i < 8; i++)
    {
        // Again, the random number function returns int. Converting
        // to a short type preserves the bit pattern in the lower 16
        // bits of the int and discards the rest.

      userkey[i] = (short) rndnum.nextInt();
    }

    // Compute encryption and decryption subkeys.

    calcEncryptKey();
    calcDecryptKey();

    // Fill plain1 with "text."
    for (int i = 0; i < array_rows; i++)
    {
      plain1[i] = (byte) i;

      // Converting to a byte
      // type preserves the bit pattern in the lower 8 bits of the
      // int and discards the rest.
    }
}

/*
* calcEncryptKey
*
* Builds the 52 16-bit encryption subkeys Z[] from the user key and
* stores in 32-bit int array. The routing corrects an error in the
* source code in the Schnier book. Basically, the sense of the 7-
* and 9-bit shifts are reversed. It still works reversed, but would
* encrypted code would not decrypt with someone else's IDEA code.
*/

private void calcEncryptKey()
{
    int j;                  // Utility variable.
```

```
   for (int i = 0; i < 52; i++) // Zero out the 52-int Z array.
      Z[i] = 0;

   for (int i = 0; i < 8; i++) // First 8 subkeys are userkey itself.
   {
      Z[i] = userkey[i] & 0xffff; // Convert "unsigned"
                                  // short to int.
   }

   // Each set of 8 subkeys thereafter is derived from left rotating
   // the whole 128-bit key 25 bits to left (once between each set of
   // eight keys and then before the last four). Instead of actually
   // rotating the whole key, this routine just grabs the 16 bits
   // that are 25 bits to the right of the corresponding subkey
   // eight positions below the current subkey. That 16-bit extent
   // straddles two array members, so bits are shifted left in one
   // member and right (with zero fill) in the other. For the last
   // two subkeys in any group of eight, those 16 bits start to
   // wrap around to the first two members of the previous eight.

   for (int i = 8; i < 52; i++)
   {
      j = i % 8;
      if (j < 6)
      {
         Z[i] = ((Z[i -7]>>>9) | (Z[i-6]<<7)) // Shift and combine.
               & 0xFFFF;                // Just 16 bits.
         continue;                      // Next iteration.
      }

      if (j == 6)  // Wrap to beginning for second chunk.
      {
         Z[i] = ((Z[i -7]>>>9) | (Z[i-14]<<7))
               & 0xFFFF;
         continue;
      }

       // j == 7 so wrap to beginning for both chunks.

      Z[i] = ((Z[i -15]>>>9) | (Z[i-14]<<7))
               & 0xFFFF;
   }
}

/*
* calcDecryptKey
*
* Builds the 52 16-bit encryption subkeys DK[] from the encryption-
* subkeys Z[]. DK[] is a 32-bit int array holding 16-bit values as
* unsigned.
*/

private void calcDecryptKey()
{
   int j, k;             // Index counters.
   int t1, t2, t3;       // Temps to hold decrypt subkeys.

   t1 = inv(Z[0]);       // Multiplicative inverse (mod x10001).
   t2 = - Z[1] & 0xffff;  // Additive inverse, 2nd encrypt subkey.
   t3 = - Z[2] & 0xffff;  // Additive inverse, 3rd encrypt subkey.
   DK[51] = inv(Z[3]);    // Multiplicative inverse (mod x10001).
   DK[50] = t3;
   DK[49] = t2;
   DK[48] = t1;

   j = 47;                // Indices into temp and encrypt arrays.
   k = 4;
   for (int i = 0; i < 7; i++)
   {
      t1 = Z[k++];
      DK[j--] = Z[k++];
      DK[j--] = t1;
      t1 = inv(Z[k++]);
      t2 = -Z[k++] & 0xffff;
      t3 = -Z[k++] & 0xffff;
      DK[j--] = inv(Z[k++]);
      DK[j--] = t2;
      DK[j--] = t3;
      DK[j--] = t1;
   }

   t1 = Z[k++];
   DK[j--] = Z[k++];
   DK[j--] = t1;
   t1 = inv(Z[k++]);
   t2 = -Z[k++] & 0xffff;
   t3 = -Z[k++] & 0xffff;
   DK[j--] = inv(Z[k++]);
   DK[j--] = t3;
   DK[j--] = t2;
   DK[j--] = t1;
}




/*
* mul
*
* Performs multiplication, modulo (2**16)+1. This code is structured
* on the assumption that untaken branches are cheaper than taken
* branches, and that the compiler doesn't schedule branches.
* Java: Must work with 32-bit int and one 64-bit long to keep
* 16-bit values and their products "unsigned." The routine assumes
* that both a and b could fit in 16 bits even though they come in
* as 32-bit ints. Lots of "& 0xFFFF" masks here to keep things 16-bit.
* Also, because the routine stores mod (2**16)+1 results in a 2**16
* space, the result is truncated to zero whenever the result would
* zero, be 2**16. And if one of the multiplicands is 0, the result
* is not zero, but (2**16) + 1 minus the other multiplicand (sort
* of an additive inverse mod 0x10001).

* NOTE: The java conversion of this routine works correctly, but
* is half the speed of using Java's modulus division function (%)
* on the multiplication with a 16-bit masking of the result--running
* in the Symantec Caje IDE. So it's not called for now; the test
* uses Java % instead.
*/

private int mul(int a, int b) throws ArithmeticException
{
   long p;          // Large enough to catch 16-bit multiply
                    // without hitting sign bit.
   if (a != 0)
   {
      if(b != 0)
      {
         p = (long) a * b;
         b = (int) p & 0xFFFF;   // Lower 16 bits.
         a = (int) p >>> 16;     // Upper 16 bits.

         return (b - a + (b < a ? 1 : 0) & 0xFFFF);
      }
      else
         return ((1 - a) & 0xFFFF); // If b = 0, then same as
                                 // 0x10001 - a.
   }
   else                          // If a = 0, then return
      return((1 - b) & 0xFFFF);  // same as 0x10001 - b.
}

/*
* inv
*
* Compute multiplicative inverse of x, modulo (2**16)+1 using
* extended Euclid's GCD (greatest common divisor) algorithm.
* It is unrolled twice to avoid swapping the meaning of
* the registers. And some subtracts are changed to adds.
* Java: Though it uses signed 32-bit ints, the interpretation
* of the bits within is strictly unsigned 16-bit.
*/

private int inv(int x)
{
   int t0, t1;
   int q, y;

   if (x <= 1)          // Assumes positive x.
      return(x);        // 0 and 1 are self-inverse.

   t1 = 0x10001 / x;    // (2**16+1)/x; x is >= 2, so fits 16 bits.
   y = 0x10001 % x;
   if (y == 1)
      return((1 - t1) & 0xFFFF);

   t0 = 1;
   do {
      q = x / y;
      x = x % y;
      t0 += q * t1;
      if (x == 1) return(t0);
      q = y / x;
      y = y % x;
      t1 += q * t0;
   } while (y != 1);

   return((1 - t1) & 0xFFFF);
}

/*
* freeTestData
*
* Nulls arrays and forces garbage collection to free up memory.
```

```
*/

void freeTestData()
{
   plain1 = null;
   crypt1 = null;
   plain2 = null;
   userkey = null;
   Z = null;
   DK = null;

}
}
```

B.11 JGFSeries

Listing B.14 : `JGFSeriesBenchSizeA.java`: main class

```
/**************************************************************************
*                                                                         *
*         Java Grande Forum Benchmark Suite - Thread Version 1.0          *
*                                                                         *
*                            produced by                                  *
*                                                                         *
*                  Java Grande Benchmarking Project                       *
*                                                                         *
*                                at                                       *
*                                                                         *
*                Edinburgh Parallel Computing Centre                      *
*                                                                         *
*                email: epcc-javagrande@epcc.ed.ac.uk                     *
*                                                                         *
*                                                                         *
*      This version copyright (c) The University of Edinburgh, 2001.     *
*                         All rights reserved.                            *
*                                                                         *
**************************************************************************/


public class JGFSeriesBenchSizeA{

  public static int nthreads;

  public static void main(String argv[]){
    nthreads = 2;

    JGFSeriesBench se = new JGFSeriesBench(nthreads);
    se.JGFrun(0);

  }
}
```

Listing B.15 : JGFSeriesBench.java: benchmark driver

```
/**************************************************************************
*                                                                         *
*         Java Grande Forum Benchmark Suite - Thread Version 1.0          *
*                                                                         *
*                            produced by                                  *
*                                                                         *
*                  Java Grande Benchmarking Project                       *
*                                                                         *
*                                at                                       *
*                                                                         *
*                Edinburgh Parallel Computing Centre                      *
*                                                                         *
*                email: epcc-javagrande@epcc.ed.ac.uk                     *
*                                                                         *
*                                                                         *
*      This version copyright (c) The University of Edinburgh, 2001.     *
*                         All rights reserved.                            *
*                                                                         *
**************************************************************************/

public class JGFSeriesBench extends SeriesTest {

  public static int nthreads;
  private int size;
  private int datasizes[]={10000,100000,1000000};
  //private int datasizes[]={10,100,1000};

  public JGFSeriesBench(int nthreads) {
    this.nthreads=nthreads;
  }

  public void JGFsetsize(int size){
    this.size = size;
  }

  public void JGFinitialise(){
    array_rows = datasizes[size];
    buildTestData();
  }

  public void JGFkernel(){
    Do();
  }

  public void JGFvalidate(){
     double ref[][] = {{2.8729524964837996, 0.0},
                   {1.1161046676147888, -1.8819691893398025},
                   {0.34429060398168704, -1.1645642623320958},
                   {0.15238898702519288, -0.8143461113044298}};

    for (int i = 0; i < 4; i++){
      for (int j = 0; j < 2; j++){
    double error = Math.abs(TestArray[j][i] - ref[i][j]);
    if (error > 1.0e-12 ){
      /*System.out.println("Validation failed for coefficient " +
      j + "," + i );
      System.out.println("Computed value = " + TestArray[j][i]);
      System.out.println("Reference value = " + ref[i][j]);*/
    }
      }
    }
  }

  public void JGFtidyup(){
    freeTestData();
  }



  public void JGFrun(int size){

    JGFsetsize(size);
    JGFinitialise();
    JGFkernel();
    JGFvalidate();
    JGFtidyup();

  }
}
```

Listing B.16 : `SeriesRunner.java`: worker thread

```
//This is the Thread

class SeriesRunner implements Runnable {

    int id;

    public SeriesRunner(int id){
    this.id=id;
    }

    public void run() {

    double omega;    // Fundamental frequency.
    int ilow,iupper,slice;

    //int array_rows=SeriesTest.array_rows;

    // Calculate the fourier series. Begin by calculating A[0].

    if (id==0) {
    SeriesTest.TestArray[0][0]=TrapezoidIntegrate((double)0.0, //Lower bound.
                         (double)2.0,        // Upper bound.
                         1000,                // # of steps.
                         (double)0.0,        // No omega*n needed.
                         0) / (double)2.0;   // 0 = term A[0].
        }

    // Calculate the fundamental frequency.
    // ( 2 * pi ) / period...and since the period
    // is 2, omega is simply pi.

    omega = (double) 3.1415926535897932;

    slice = (SeriesTest.array_rows + JGFSeriesBench.nthreads-1)/JGFSeriesBench.nthreads;

    ilow = id*slice;
    if(id==0) ilow=id*slice+1;
```

```
   iupper = (id+1)*slice;
   if (iupper > SeriesTest.array_rows ) iupper=SeriesTest.array_rows;


   for (int i = ilow; i < iupper; i++)
   {
      // Calculate A[i] terms. Note, once again, that we
      // can ignore the 2/period term outside the integral
      // since the period is 2 and the term cancels itself
      // out.

      SeriesTest.TestArray[0][i] = TrapezoidIntegrate((double)0.0,
                      (double)2.0,
                      1000,
                      omega * (double)i,
              1);                   // 1 = cosine term.

                 // Calculate the B[i] terms.

      SeriesTest.TestArray[1][i] = TrapezoidIntegrate((double)0.0,
                      (double)2.0,
                      1000,
                      omega * (double)i,
                      2);                   // 2 = sine term.
   }




   }

/*
* TrapezoidIntegrate
*
* Perform a simple trapezoid integration on the function (x+1)**x.
* x0,x1 set the lower and upper bounds of the integration.
* nsteps indicates # of trapezoidal sections.
* omegan is the fundamental frequency times the series member #.
* select = 0 for the A[0] term, 1 for cosine terms, and 2 for
* sine terms. Returns the value.
*/

private double TrapezoidIntegrate (double x0, // Lower bound.
                     double x1,           // Upper bound.
                     int nsteps,          // # of steps.
                     double omegan,       // omega * n.
                     int select)          // Term type.
{
   double x;           // Independent variable.
   double dx;          // Step size.
   double rvalue;      // Return value.

   // Initialize independent variable.

   x = x0;

   // Calculate stepsize.

   dx = (x1 - x0) / (double)nsteps;

   // Initialize the return value.
   rvalue = thefunction(x0, omegan, select) / (double)2.0;

   // Compute the other terms of the integral.

   if (nsteps != 1)
   {
         --nsteps;           // Already done 1 step.
         while (--nsteps > 0)
         {
               x += dx;
               rvalue += thefunction(x, omegan, select);
         }
   }

   // Finish computation.

   rvalue=(rvalue + thefunction(x1,omegan,select) / (double)2.0) * dx;
   return(rvalue);
}

/*
* thefunction
*
* This routine selects the function to be used in the Trapezoid
* integration. x is the independent variable, omegan is omega * n,
* and select chooses which of the sine/cosine functions
* are used. Note the special case for select=0.
*/

private double thefunction(double x, // Independent variable.
              double omegan,         // Omega * term.
              int select)            // Choose type.
{

   // Use select to pick which function we call.

   switch(select)
   {
      case 0: return(Math.pow(x+(double)1.0,x));

      case 1: return(Math.pow(x+(double)1.0,x) * Math.cos(omegan*x));

      case 2: return(Math.pow(x+(double)1.0,x) * Math.sin(omegan*x));
   }

   // We should never reach this point, but the following
   // keeps compilers from issuing a warning message.

   return (0.0);
}
}
```

Listing B.17 : `SeriesTest.java`: performs calculation

```java
/**************************************************************************
*                                                                         *
*       Java Grande Forum Benchmark Suite - Thread Version 1.0            *
*                                                                         *
*                            produced by                                  *
*                                                                         *
*                  Java Grande Benchmarking Project                       *
*                                                                         *
*                                at                                       *
*                                                                         *
*                Edinburgh Parallel Computing Centre                      *
*                                                                         *
*                email: epcc-javagrande@epcc.ed.ac.uk                     *
*                                                                         *
*                  Original version of this code by                       *
*                 Gabriel Zachmann (zach@igd.fhg.de)                      *
*                                                                         *
*     This version copyright (c) The University of Edinburgh, 2001.       *
*                         All rights reserved.                            *
*                                                                         *
**************************************************************************/

/**
* Class SeriesTest
*
* Performs the transcendental/trigonometric portion of the
* benchmark. This test calculates the first n fourier
* coefficients of the function (x+1)^x defined on the interval
* 0,2 (where n is an arbitrary number that is set to make the
* test last long enough to be accurately measured by the system
* clock). Results are reported in number of coefficients calculated
* per sec.
*
* The first four pairs of coefficients calculated shoud be:
* (2.83777, 0), (1.04578, -1.8791), (0.2741, -1.15884), and
* (0.0824148, -0.805759).
*/


public class SeriesTest
{

// Declare class data.

static int array_rows;
public static double [] [] TestArray; // Array of arrays.




/*
* buildTestData
*
*/

// Instantiate array(s) to hold fourier coefficients.

void buildTestData()
{
   // Allocate appropriate length for the double array of doubles.
   TestArray = new double [2][array_rows];
}



/*
* Do
*
* This consists of calculating the
* first n pairs of fourier coefficients of the function (x+1)^x on
* the interval 0,2. n is given by array_rows, the array size.
* NOTE: The # of integration steps is fixed at 1000.
*/

void Do()
{

   int i,j;
   Runnable thobjects[] = new Runnable [JGFSeriesBench.nthreads];
   Thread th[] = new Thread [JGFSeriesBench.nthreads];


   //Start Threads

   for(i=0;i<JGFSeriesBench.nthreads;i++) {
   thobjects[i] = new SeriesRunner(i);
   th[i] = new Thread(thobjects[i]);
   th[i].start();
   }


   for(i=0;i<JGFSeriesBench.nthreads;i++) {
      try {
     th[i].join();
   }
      catch (InterruptedException e) {}
   }


}
void freeTestData()
{
   TestArray = null; // Destroy the array.
}
}
```

B.12 JGFSparseMatmult

Listing B.18 : `JGFSparseMatmultBenchSizeA.java`: main class

```
/**************************************************************************
*                                                                         *
*         Java Grande Forum Benchmark Suite - Thread Version 1.0          *
*                                                                         *
*                            produced by                                  *
*                                                                         *
*                  Java Grande Benchmarking Project                       *
*                                                                         *
*                                at                                       *
*                                                                         *
*                Edinburgh Parallel Computing Centre                      *
*                                                                         *
*                email: epcc-javagrande@epcc.ed.ac.uk                     *
*                                                                         *
*                                                                         *
*      This version copyright (c) The University of Edinburgh, 2001.      *
*                         All rights reserved.                            *
*                                                                         *
**************************************************************************/

public class JGFSparseMatmultBenchSizeA{

  public static int nthreads;

  public static void main(String argv[]){

  nthreads = 2;


  JGFSparseMatmultBench smm = new JGFSparseMatmultBench(nthreads);
  smm.JGFrun(0);

  }
}
```

Listing B.19 : `JGFSparseMatmultBench.java`: benchmark driver

```
/**************************************************************************
*                                                                         *
*         Java Grande Forum Benchmark Suite - Thread Version 1.0          *
*                                                                         *
*                            produced by                                  *
*                                                                         *
*                  Java Grande Benchmarking Project                       *
*                                                                         *
*                                at                                       *
*                                                                         *
*                Edinburgh Parallel Computing Centre                      *
*                                                                         *
*                email: epcc-javagrande@epcc.ed.ac.uk                     *
*                                                                         *
*                                                                         *
*      This version copyright (c) The University of Edinburgh, 2001.      *
*                         All rights reserved.                            *
*                                                                       *
*************************************************************************/
import java.util.Random;

public class JGFSparseMatmultBench extends SparseMatmult {

  public static int nthreads;

  private int size;
  private static final long RANDOM_SEED = 10101010;

  private static final int datasizes_M[] = {50000,100000,500000};
  private static final int datasizes_N[] = {50000,100000,500000};
  private static final int datasizes_nz[] = {250000,500000,2500000};
  private static final int SPARSE_NUM_ITER = 200;
  /*private static final int datasizes_M[] = {5000,10000,50000};
  private static final int datasizes_N[] = {5000,10000,50000};
  private static final int datasizes_nz[] = {25000,50000,250000};
  private static final int SPARSE_NUM_ITER = 1; */

  Random R = new Random(RANDOM_SEED);

  double [] x;
  double [] y;
  double [] val;
  int [] col;
  int [] row;
  int [] lowsum;
  int [] highsum;

  public JGFSparseMatmultBench(int nthreads) {
   this.nthreads = nthreads;
  }

  public void JGFsetsize(int size){
    this.size = size;

  }

  public void JGFinitialise(){

  x = RandomVector(datasizes_N[size], R);
  y = new double[datasizes_M[size]];

  val = new double[datasizes_nz[size]];
  col = new int[datasizes_nz[size]];
  row = new int[datasizes_nz[size]];

  int [] ilow = new int[nthreads];
  int [] iup = new int[nthreads];
  int [] sum = new int[nthreads+1];
  lowsum = new int[nthreads+1];
  highsum = new int[nthreads+1];
  int [] rowt = new int[datasizes_nz[size]];
  int [] colt = new int[datasizes_nz[size]];
  double [] valt = new double[datasizes_nz[size]];
  int sect;

    for (int i=0; i<datasizes_nz[size]; i++) {

        // generate random row index (0, M-1)
        row[i] = Math.abs(R.nextInt()) % datasizes_M[size];
        // generate random column index (0, N-1)
        col[i] = Math.abs(R.nextInt()) % datasizes_N[size];

        val[i] = R.nextDouble();

    }

// reorder arrays for parallel decomposition

    sect = (datasizes_M[size] + nthreads-1)/nthreads;

    for (int i=0; i<nthreads; i++) {
      ilow[i] = i*sect;
      iup[i] = ((i+1)*sect)-1;
      if(iup[i] > datasizes_M[size]) iup[i] = datasizes_M[size];
    }

    for (int i=0; i<datasizes_nz[size]; i++) {
      for (int j=0; j<nthreads; j++) {
        if((row[i] >= ilow[j]) && (row[i] <= iup[j])) {
          sum[j+1]++;
        }
      }
    }

    for (int j=0; j<nthreads; j++) {
      for (int i=0; i<=j; i++) {
         lowsum[j] = lowsum[j] + sum[j-i];
         highsum[j] = highsum[j] + sum[j-i];
      }
    }

    for (int i=0; i<datasizes_nz[size]; i++) {
      for (int j=0; j<nthreads; j++) {
        if((row[i] >= ilow[j]) && (row[i] <= iup[j])) {
          rowt[highsum[j]] = row[i];
          colt[highsum[j]] = col[i];
          valt[highsum[j]] = val[i];
          highsum[j]++;
        }
      }
    }

    for (int i=0; i<datasizes_nz[size]; i++) {
      row[i] = rowt[i];
      col[i] = colt[i];
      val[i] = valt[i];
    }

  }

  public void JGFkernel(){

    SparseMatmult.test(y, val, row, col, x, SPARSE_NUM_ITER, lowsum, highsum);

  }

  public void JGFvalidate(){

    double refval[] = {75.02484945753453,150.0130719633895,749.5245870753752};
    double dev = Math.abs(ytotal - refval[size]);
    if (dev > 1.0e-10 ){
      /*System.out.println("Validation failed");
      System.out.println("ytotal = " + ytotal + " " + dev + " " + size);*/
    }

  }

  public void JGFtidyup(){

  }



  public void JGFrun(int size){

    JGFsetsize(size);
    JGFinitialise();
    JGFkernel();
    JGFvalidate();
    JGFtidyup();
  }

  private static double[] RandomVector(int N, java.util.Random R)
  {
     double A[] = new double[N];
     for (int i=0; i<N; i++)
       A[i] = A[i] = R.nextDouble() * 1e-6;

     return A;
  }
}
```

Listing B.20 : `SparseRunner.java`: worker thread

```
class SparseRunner implements Runnable {

    int id,nz,row[],col[],NUM_ITERATIONS;
    double val[],x[];
    int lowsum[];
    int highsum[];

   public SparseRunner(int id, double val[], int row[],int col[], double x[], int NUM_ITERATIONS,int nz, int lowsum[], i
        this.id = id;
        this.x=x;
        this.val=val;
        this.col=col;
        this.row=row;
        this.nz=nz;
        this.NUM_ITERATIONS=NUM_ITERATIONS;
        this.lowsum = lowsum;
        this.highsum = highsum;
    }

   public void run() {

         for (int reps=0; reps<NUM_ITERATIONS; reps++) {
           for (int i=lowsum[id]; i<highsum[id]; i++) {
             SparseMatmult.yt[ row[i] ] += x[ col[i] ] * val[i];
```

```
          }
        }

  }
}
```

Listing B.21 : `SparseMatmult.java`: performs the matrix multiplications

```
/**************************************************************************
*                                                                         *
*         Java Grande Forum Benchmark Suite - Thread Version 1.0          *
*                                                                         *
*                            produced by                                  *
*                                                                         *
*                  Java Grande Benchmarking Project                       *
*                                                                         *
*                                at                                       *
*                                                                         *
*                Edinburgh Parallel Computing Centre                      *
*                                                                         *
*                email: epcc-javagrande@epcc.ed.ac.uk                     *
*                                                                         *
*      adapted from SciMark 2.0, author Roldan Pozo (pozo@cam.nist.gov) *
*                                                                         *
*      This version copyright (c) The University of Edinburgh, 2001. *
*                         All rights reserved.                            *
*                                                                         *
**************************************************************************/

public class SparseMatmult
{

  public static double ytotal = 0.0;
  public static double yt[];

    /* 10 iterations used to make kernel have roughly
        same granulairty as other Scimark kernels. */

    public static void test( double y[], double val[], int row[],
                int col[], double x[], int NUM_ITERATIONS, int lowsum[], int highsum[])
    {
        int nz = val.length;
        yt=y;

        SparseRunner thobjects[] = new SparseRunner[JGFSparseMatmultBench.nthreads];
        Thread th[] = new Thread[JGFSparseMatmultBench.nthreads];

        for(int i=0;i<JGFSparseMatmultBench.nthreads;i++) {
          thobjects[i] = new SparseRunner(i,val,row,col,x,NUM_ITERATIONS,nz,lowsum,highsum);
          th[i] = new Thread(thobjects[i]);
          th[i].start();
        }


        for(int i=0;i<JGFSparseMatmultBench.nthreads;i++) {
          try {
           th[i].join();
          } catch (InterruptedException e) {}
        }

          for (int i=0; i<nz; i++) {
            ytotal += yt[ row[i] ];
          }

    }
}
```

B.13 Performance

Listing B.22 : The performance program copied from Figure 4.1 of [7] creates a specified number of threads and a specified number of locks per thread to measure how many paths the *ExitBlock-RW* algorithm must search

```
public class Performance implements Runnable {
  static Lock[] locks;
  static int nThreads;
  static int nLocks;

  public static void main(String[] args) {
    nThreads = 2; /* depends on the test */
    nLocks = 1; /* depends on the test */
    locks = new Lock[nLocks];

    for (int i=0; i<nLocks; i++)
      locks[i] = new Lock();

    for (int i=0; i < nThreads; i++)
      new Performance();

  }

  public Performance() {
    new Thread(this).start();
  }

  public void run() {
    for (int i=0; i <nLocks; i++)
    synchronized (locks[i]) {}
  }
}
```

B.14 Deadlock

Listing B.23 : Simple deadlocking example where two threads tries to acquire the same two locks in a different order.

```
public class Deadlock {
  private static Integer A = new Integer(0);
  private static Integer B = new Integer(0);

  public static void main(String[] args) {
    Thread t1;

    t1 = new LockAB(A,B);
    t1.start();

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

B.15 Deadlock3

Listing B.24 : A deadlock example where three threads deadlock since the locks are acquired in a cycle order

```
public class Deadlock3 implements Runnable {
  static Lock a = new Lock();
  static Lock b = new Lock();
  static Lock c = new Lock();

  public static void main(String[] args)
  {
    new Deadlock3(0);
    new Deadlock3(1);
    new Deadlock3(2);
  }

  int order;

  public Deadlock3(int order) {
    this.order = order;
    new Thread(this).start();
  }

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

B.16 SplitSync

Listing B.25 : `SplitSync.java`: assertion failure due to insufficient locking

```java
public class SplitSync implements Runnable {

  static Resource resource = new Resource();

  public static void main(String[] args) {
    new SplitSync();
    new SplitSync();
  }

  public SplitSync() {
    new Thread(this).start();
  }

  public void run() {
    int y;

    synchronized (resource) {
      y = resource.x;
    }

    synchronized (resource) {
      jnuke.Assertion.check( resource.x == y );
      resource.x = y + 1;
    }
  }
}
```

```
Assertion failed at SplitSync.run ()V (line 21) (pc 26)
  (thread 1)
(JNukeExitBlock (JNukeSchedule
  (JNukeThreadSwitch (from_thread 0) (to_thread 0)
    (JNukeMethod "SplitSync.<init>" (JNukeSignature
      "V" (JNukeVector))) (pc 7) (line 17))
  ...
)
```

Figure B.1: A portion of the output produced by the SplitSync problem. The assertion violation is detected, the according current position and the scheduler history are printed out.

B.17 Dining Philosopher

Listing B.26 : `Fork.java`: class used by dining philosophers

```java
public class Fork {

  Philosopher owner = null;

  public synchronized void acquire(Philosopher p)
      throws InterruptedException {
    while( owner != null ) { wait(); }
    owner = p;
  }

  public synchronized void release() {
    owner = null;
    notifyAll();
  }
}
```

Listing B.27 : `Philosopher.java`: class representing a philosopher

```
public class Philosopher extends Thread
{
  Fork l;
  Fork r;
  int n;
  String name;

  public Philosopher(String name, int n, Fork l, Fork r, boolean
                   takeRightFirst)
  {
    if ( takeRightFirst )
    {
      this.l = r;
      this.r = l;
    } else {
      this.l = l;
      this.r = r;
    }

    this.n = n;
    this.name = name;
  }

  public void run() {
    int i;

    for (i = 0; i < n; i++)
    {
      try {
        l.acquire(this);
        r.acquire(this);
        r.release();
        l.release();
      } catch (InterruptedException e) { return; }
    }
  }
}
```

Listing B.28 : `DiningPhilo.java`: dinining philosophers with three threads

```
public class DiningPhilo {

  public static void main(String[] args) throws InterruptedException {
    Fork f1 = new Fork();
    Fork f2 = new Fork();
    Fork f3 = new Fork();

    Philosopher p1 = new Philosopher( "sophokles", 1, f1, f2, false );
    Philosopher p2 = new Philosopher( "euripides", 1, f2, f3, false );
    Philosopher p3 = new Philosopher( "anaximandres", 1, f3, f1, false );

    p1.start();
    p2.start();
    p3.run();
  }
}
```

B.18 DeadlockWait

Listing B.29 : `DeadlockWait`: condition deadlock due to a unreleaesed lock

```
public class DeadlockWait implements Runnable {

  static Lock a = new Lock();
  static Lock b = new Lock();

  public static void main(String[] args) {
    new DeadlockWait(true);
    new DeadlockWait(false);

    synchronized(a) { a.i = 0; a.j = 0; }
    synchronized(b) { b.i = 0; b.j = 0; }
  }

  boolean ab;

  public DeadlockWait(boolean ab) {
    this.ab = ab;
    new Thread(this).start();
  }

  public void run() {
    if (ab) {
      synchronized (a) {
        synchronized (b) {
          try {
            a.i = b.i; a.j = 2;
            b.j = a.j; b.i = a.j;
            b.wait(); }
          catch (InterruptedException i) {}
        }
        a.i = 0; a.j = 0;
      }
    } else {
      synchronized (a) {
        a.i = 1; a.j = 1;
      }

      synchronized (b) {
        b.i = 1; b.j = 1;
        b.notifyAll();
      }
    }
  }
}
```

B.19 BufferIf

Listing B.30 : Sample bounded buffer class containing a timing-dependent error. The enq function should use a while instaed of an if. With an if, if the enqueueing thread is woken up but some other thread executes before it takes control and changes the condition, the enqueueing thread will go ahead and execute even though the condition is false.]Buffer.java: bounded buffer with an error in its enqueue method.]Sample bounded buffer class containing a timing-dependent error. The enq function should use a while instaed of an if. With an if, if the enqueueing thread is woken up but some other thread executes before it takes control and changes the condition, the enqueueing thread will go ahead and execute even though the condition is false.

```
public class Buffer {
  static final int BUFSIZE = 2;
  private int first, last;
  private Object[] els;

public Buffer() {
  first = 0;
  last = 0;
  els = new Object[BUFSIZE];
}

public synchronized void enq(Object x) throws InterruptedException {
  if ((last+1) % BUFSIZE == first ) /* BUG */
    this.wait();

  jnuke.Assertion.check((last+1) % 2 != first);

  els[last] = x;

  last = (last+1) % BUFSIZE;

  this.notifyAll();
}

public synchronized Object deq() throws InterruptedException {

  while (first == last)
    this.wait();

  Object val = els[first];

  first = (first+1) % BUFSIZE;
  this.notifyAll();

  return val;
}

}
```

Listing B.31 : `Producer.java`: producer thread writing into the buffer

```java
public class Producer implements Runnable {
  private Buffer buffer;

  public Producer(Buffer b) {
    buffer = b;
  }

  public void run() {
    try {
      for (int i=0; i<2; i++)
        buffer.enq(this);
    } catch (InterruptedException i) {}
  }
}
```

Listing B.32 : `Consumer.java`: consumer thread reading from the buffer

```java
public class Consumer implements Runnable {
  private Buffer buffer;

  public Consumer(Buffer b) {
    buffer = b;
  }

  public void run() {
    try {
      for (int i=0; i<4; i++)
        buffer.deq();
    } catch (InterruptedException i) {}
  }
}
```

Listing B.33 : `BufferIf`: producer-consumer problem with two producers and one consumer thread

```java
public class BufferIf {
  static final int IP = 2;

  public static void main(String[] args) {
    Buffer b = new Buffer();

    new Thread( new Producer(b)).start();
    new Thread( new Producer(b)).start();
    new Thread( new Consumer(b)).start();
  }
}
```

B.20 BufferWhile

Listing B.34 : A correct bounded buffer where the invariant is rechecked when the producer thread is rescheduled.]`Buffer2.java`: bounded buffer with a correct enqueue method]A correct bounded buffer where the invariant is rechecked when the producer thread is rescheduled.

```
public class Buffer2 {

  static final int BUFSIZE = 2;
  private int first, last;
  private Object[] els;

public Buffer() {
  first = 0;
  last = 0;
  els = new Object[BUFSIZE];
}

public synchronized void enq(Object x) throws InterruptedException {

  while ((last+1) % BUFSIZE == first ) /* fixed */
    this.wait();

  jnuke.Assertion.check((last+1) % 2 != first);

  els[last] = x;
  last = (last+1) % BUFSIZE;
  this.notifyAll();
}

public synchronized Object deq() throws InterruptedException {
  while (first == last)
    this.wait();

  Object val = els[first];
  first = (first+1) % BUFSIZE;
  this.notifyAll();

  return val;
}

}
```

Appendix C

Test Cases

This chapter lists all test cases of the virtual machine and the runtime verification tools. The table below shows the number of test cases.

Test suite	Number of test cases
Virtual machine	231
Runtime verification	56
Total	287

vm/instancedesc

Number	Description
0	Calculation of field offsets for a simple class
1	Calculation of field offsets for a class with static fields
2	Calculation of field offsets for a class with multi-inheritance
3	Handling of shadowed variables

Table C.1: Test cases of `JNukeInstanceDesc`

vm/arrayinstancedesc

Number	Description
0	Creation of array instance descriptors
1	Dereferencing of array types
2	Offset calculation
3	Creation of instances

Table C.2: Test cases of `JNukeArrayInstanceDesc`

vm/heapmgr

Number	Description
0	Creation of a heap manager
1	Iteration of the instance descriptors
2	Creation of an instance of a simple class
3	Creation of (multidimensional) array instances
4	Read and write array components
5	Read and write components of a two dimensional array
6	C array for performance comparisons. Allows comparison to test case 4 and 5.
7	Out of bounds read and write operations
8	Rollback of an array instance
9	misc
10	Rollback of an array instance
11	Rollback of an instance with class and object fields
12	Component type dereferencing
13	read and write action listener test
14	read and write action listener test

Table C.4: Test cases of `JNukeHeapMgr`

vm/heaplog

Number	Description
0	Logging of field write and read access
1	Logging of field write and read access
2	Creation of a number of objects in connection with several rollbacks
3	Creation of a number of arrays in connection with several rollbacks
4	Release of an non-empty heap log

Table C.5: Test cases of `JNukeHeapLog`

vm/native

Number	Description
0	Tests endianess issues

Table C.6: Test cases of `JNukeNative`

vm/thread

Number	Description
0	Creates threads, set and test some flags
1	Creation of stack-frames
2	Creation of stack-frames
3	One thread joins another one
4	Sets a thread waiting and tries to reacquire the locks
5	Creation of a milestone
6	Creation of one milestone. One rollback is performed
7	Milestone/Rollback test with a thread owning locks
8	Milestone/Rollback test in connection with stack frames
9	Reference Counting of stack frames
10	Milestone/Rollback test in connection with stack frames
11	Performing of several milestone/rollback operations
12	Milestone/Rollback test in connection with thread's lock list
13	Last hold lock test used for reverse lock chain analysis
14	toString

Table C.7: Test cases of `JNukeThread`

vm/stackframe

Number	Description
0	Creation, cloning, release of a stack frame
1	Sets a milestone
2	Sets a milestone, performs one rollback
3	Sets a milestone, performs one rollback
4	One milestone, performing several rollbacks

Table C.8: Test cases of `JNukeStackFrame`

vm/waitlist

Number	Description
0	Inserts threads and performs resumeAll
1	Inserts threads and resumes each
2	Cloning of a wait list
3	Sets a milestone
4	One milestone and one rollback
5	Creates one milestone, performs then several rollbacks

Table C.9: Test cases of `JNukeWaitList`

vm/lock

Number	Description
0	One lock, several threads compete for this lock
1	One thread acquiring several locks
2	resume threads waiting on a lock
3	Creation of several milestones
4	Creation of milestones, several rollbacks
5	Creation of several milestones, no rollback

Table C.10: Test cases of `JNukeLock`

vm/lockmgr

Number	Description
0	Acquires and releases objects locks
1	Thread dies that still owns locks. Locks are released
2	Create one milestone. One rollback is performed
3	Milestone/rollback with changed locks
4	Creation of several milestone. Several rollbacks are performed
5	Listener test

Table C.11: Test cases of `JNukeLockMgr`

vm/waitsetmgr

Number	Description
0	Sets two threads waiting
1	Sets two threads waiting; awaken each by notify
2	Sets two threads waiting; awaken each by notify
3	Sets two threads waiting; awaken threads by notifyAll
4	Sets a milestone
5	Sets a milestone; performs one rollback
6	Two awaken threads competing for a lock
7	One waiting threads which is notified.

Table C.12: Test cases of `JNukeWaitsetMgr`

vm/vtable

Number	Description
0	Creates a vtable of a simple class
1	Creates a vtable of a class with a super class
2	Creates the vtable for java/lang/System
3	Creates the vtable for java/io/PrintStream
4	Creates the vtable for java/lang/String

Table C.13: Test cases of `JNukeVirtualTable`

vm/rtenvironment

Table C.14: Test cases of `JNukeRuntimeEnvironment`

Number	Description
0	Loops of 100’000 iterations
1	Reads and writes static fields
2	Reads and writes static fields
3	Creates object and reads instance fields
4	Creates object and reads instance fields
5	Array iteration
6	Creates array of different types
7	Creates a two dimensional array
8	Tests table switch operation
9	Tests lookup switch operation
10	Tests checkcast and instanceof operator
11	Tests monitorEnter and monitorExit operations
12	Sorts an array with Bubble-Sort
13	Reads many fields of an object
14	Creates many objects
15	Reads a shadowed fields
16	Invocation of virtual and static methods
17	Invocation of virtual methods
18	Calls derived virtual methods
19	Calls a static initializer
20	Native call test
21	Performs system.out.println(...)
22	A program with constant string values
23	Invocation of many static methods
24	Creates some thread instances
25	Creates some thread instances
26	Throws null pointer exceptions
27	Throws and catches array index out of bounds exceptions
28	Throws and catches division by zero exceptions
29	Throws and catches class cast exceptions
30	Creates four milestones

Number	Description
31	Creates a couple of milestone; performs rollbacks
32	Creates four milestone; performs one rollback
33	Tests rollback/milestone in connection with threads
34	Tests rollback/milestone in connection with the heap manager
35	Tests rollback/milestone in connection with locks
36	Tests null pointer exception and out of bounds exception at Object.arraycopy
37	Tests IllegalMonitorStateExceptions and NullPointerExceptions (wait, notify[All])
38	Tests ClassDefNotFoundError
39	Invocation of methods. Tests parameter and return value handling
40	Compliance test for integer overflows
41	Basic test for register operations
42	Performs float and double cast operations
43	Performs left and right shift operations
44	Performs left and right shift operations for long values
45	Performs and, or, and xor operations
46	Tests jnuke.Assertion
47	Performs further float and double cast operations
48	Performs misc float and double operations
49	Performs negative operations
50	dupX
51	Tests NoClassDefFoundError
52	Experiment from Chapter 5
53	Experiment from Chapter 5
54	Experiment from Chapter 5
55	Experiment from Chapter 5
56	Experiment from Chapter 5
57	Experiment from Chapter 5
58	Experiment from Chapter 5
59	Experiment from Chapter 5
60	Experiment from Chapter 5
61	Experiment from Chapter 5
62	Experiment from Chapter 5
63	Throws NoSuchFieldError and NoSuchMethodException
64	Throws an internal error which is not caught. Output written to error file.

vm/rrscheduler

Number	Description
0	Two threads, one joins to other one
1	Two threads, one joins to other one
2	Interruption of a thread
3	Invocation of synchronized methods
4	Invocation of synchronized methods
5	Recursive invocation of synchronized methods
6	MonitorEnter and monitorExit test
7	Acquires recursive monitor locks
8	Producer/consumer example
9	Dining philosopher
10	Calls start() twice causing a IllegalThreadStateException
11	Performs InterruptedException while one thread joins another join
12	Multi-threaded example with a condition deadlock (is detected)
13	BufferIf example
14	Dining philosopher each eating 3000 times (TTL=5)
15	Dining philosopher each eating 3000 times (TTL=100)
16	Dining philosopher each eating 3000 times (TTL=1000)
17	Dining philosopher each eating 3000 times (TTL=10000)
18	Dining philosopher each eating 3000 times (TTL=100000)
19	Producer/Consumer example (120’000 iterations, 3 threads)
20	Producer/Consumer example (120’000 iterations, 100 threads)
21	JGFCrypt with 2 threads
22	JGFCrypt with 20 threads
23	JGFCrypt with 200 threads
24	JGFSeries with 2 threads
25	JGFSparseMatmult with 2 threads
26	JGFSparseMatmult with 20 threads
27	JGFSeries with 20 threads

Table C.15: Test cases of `JNukeRRScheduler`

vm/vmstate

Number	Description
0	Creates and destroys a virtual machine state
1	Clones and compares virtual machine states
2	Clones and hashes virtual machine states
3	Takes a real snapshot of a runtime environment

Table C.16: Test cases of `JNukeVMState`

rv/exitblock

Table C.17: Test cases of `JNukeExitBlock`

Number	Description
0	ExitBlock: explores schedules for two threads
1	ExitBlock: two threads acquiring locks in opposite order
2	ExitBlock: three threads acquiring some locks
3	ExitBlock: two threads acquiring some locks
4	ExitBlock: two threads acquiring some locks
5	The same as #0. Additionally, schedules are printed
6	The same as #1. Additionally, schedules are printed
7	The same as #2. Additionally, schedules are printed
8	The same as #3. Additionally, schedules are printed
9	The same as #4. Additionally, schedules are printed
10	ExitBlock: ignoring of yield
11	ExitBlock: discovering of a condition deadlock
12	ExitBlock: deadlocking dining philosophers (two philosophers)
13	ExitBlock: non-deadlocking dining philosophers (two philosophers)
14	ExitBlock: one thread joins another one
15	ExitBlock: one thread joins another one
16	ExitBlock: three deadlocking dining philosophers.
17	ExitBlock-RW: Performance.java from Chapter 5 (1 thread, 1 lock)
18	ExitBlock-RW: Performance.java from Chapter 5 (2 threads, 2 locks)
19	ExitBlock-RW: Performance.java from Chapter 5 (2 threads, 100 locks)
20	ExitBlock-RW: Performance.java from Chapter 5 (3 threads, 1 lock)
21	ExitBlock-RW: Performance.java from Chapter 5 (3 threads, 50 locks)
22	ExitBlock-RW: Performance.java from Chapter 5 (3 threads, 100 locks)
23	ExitBlock-RW: Performance.java from Chapter 5 (4 threads, 20 locks)
24	ExitBlock: Deadlock3 example from Chapter 5
25	ExitBlock-RW: Deadlock3 example from Chapter 5
26	ExitBlock: SplitSync example from Chapter 5
27	ExitBlock-RW: SplitSync example from Chapter 5
28	ExitBlock-RW: two dining philosophers
29	ExitBlock: three deadlocking dining philosophers
30	ExitBlock-RW: three deadlocking dining philosophers (from Chapter 5)
31	ExitBlock-RW: four deadlocking dining philosophers (from Chapter 5)
32	ExitBlock-RW: ten deadlocking dining philosophers (from Chapter 5)
33	ExitBlock-RW: twenty deadlocking dining philosophers (from Chapter 5)
34	ExitBlock: DeadlockWait example from Chapter 5
35	ExitBlock-RW: DeadlockWait example from Chapter 5
36	ExitBlock: BufferIf example from Chapter 5
37	ExitBlock-RW: BufferIf example from Chapter 5
38	ExitBlock: Tests output of strings in connection with rollbacks
39	ExitBlock: BufferWhile example from Chapter 5
40	ExitBlock-RW: BufferWhile example from Chapter 5
41	ExitBlock: BufferIfNotify example from Chapter 5
42	ExitBlock-RW: BufferIfNotify example from Chapter 5

43	ExitBlock: tests notify()
44	ExitBlock: tests throwing of assertions and immediate abort of execution
45	ExitBlock-RW: tests throwing of assertions and immediate abort of execution
46	Misc

rv/revlockalg

Number	Description
0	ExitBlock:Creates and destroys a reverse lock chain analyzer instance
1	ExitBlock: Detects a deadlock due to locks held in reverse order
2	ExitBlock: Detects a deadlock due to locks held in reverse order
3	ExitBlock: Example with correct locking order. No deadlock therefore.
4	The test as #1; instead ExitBlock-RW is used
5	The test as #2; instead ExitBlock-RW is used
6	The test as #3; instead ExitBlock-RW is used
7	ExitBlock: Lock cycle with three threads
8	ExitBlock-RW: Lock cycle with three threads

Table C.18: Test cases of `JNukeRLCAnalyzer`

Appendix D

Miscellaneous

D.1 Supported platforms

The virtual machine has been tested on following architectures:

- Linux i386
- Mac-OS X
- Alpha Tru64 Unix
- SPARC v8 and v9

Following compilers are tested and supported:

- GCC 2.95.3
- GCC 3.2.1

D.2 Code coverage

The virtual machine has a code coverage of 100% as Table D.1 shows:

Filename	Coverage [%]	Filename	Coverage [%]
arrayinstdesc.c	100	rtenv.c	100
black.c	100	schedule.c	100
heaplog.c	100	stackframe.c	100
heapmgr.c	100	thread.c	100
instancedesc.c	100	vmstate.c	100
lock.c	100	vtable.c	100
lockmgr.c	100	waitlist.c	100
native.c	100	waitsetmgr.c	100
rbcinstr.c	100	black.c	100
rrscheduler.c	100	eraserinfo.c	100
revlockalg.c	100	exitblock.c	100

Table D.1: Code coverage of the virtual machine computed by `gcov`.

D.3 Implemented Java foundation classes

Some test cases need Java foundation classes. At the moment, classes like java/lang/String, java/lang/Math, etc. are borrowed from the Sun JDK 1.3. However, some classes of the Java foundation classes have been implemented by myself. Most of them are not complete as they contain only members used by the test cases. The classes are located at `jnuke/log/vm/rtenvironment/classpath` and are listed below:

Package	Java class	Package	Java class
java/io	PrintStream	java/lang	LinkageError
java/lang	ArithmeticException	java/lang	NoClassDefFoundError
java/lang	ArrayIndexOutOfBoundsException	java/lang	NoSuchMethodException
java/lang	Class	java/lang	NullPointerException
java/lang	ClassCastException	java/lang	Object
java/lang	Error	java/lang	Runnable
java/lang	Exception	java/lang	RuntimeException
java/lang	IllegalArgumentException	java/lang	Thread
java/lang	IllegalMonitorStateException	java/lang	Throwable
java/lang	IllegalThreadStateException	java/lang	NoSuchFieldError
java/lang	IndexOutOfBoundsException	java/lang	System
java/lang	InterruptedException	jnuke	Assertion

Table D.2: Self-implementd Java foundation classes.

Listings

List of Algorithms

List of Tables

List of Figures

Bibliography

[1] Email address of Marcel Baur. mbaur@iiic.ethz.ch.

[2] C. Artho and A. Biere. Applying Static Analysis to Large-Scale, Multi-threaded Java Programs. In D. Grant, editor, *Proc. 13th ASWEC*, pages 68–75. IEEE Computer Society, 2001.

[3] M. J. Bach. *Unix operating system*. Prentice Hall, 1986.

[4] Bandera. http://www.cis.ksu.edu/santos/bandera.

[5] JASPA Benchmark. http://www.dl.ac.uk/tcsc/staff/huyf/software/jaspa/.

[6] Beowulf. http://www.beowulf.org/.

[7] D. Bruening. Systematic Testing of Multithreaded Java Programs. Master's thesis, MIT, 1999.

[8] GNU Classpath. http://www.gnu.org/software/classpath/classpath.html.

[9] Matthew B. Dwyer and John Hatcliff. Slicing software for model construction. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 105–118, 1999.

[10] ESC/Java. http://research.compaq.com/src/esc/.

[11] Matrix MCCA from Matrix Market. http://math.nist.gov/matrixmarket/data/harwell-boeing/astroph/mcca.html.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[13] Patrice Godefroid. *Partial-Order Method for the Verification of Concurrent Systems - An approach to the State-Explosion Problem, volume 1032 of Lecture Notes in Computer Science* . Springer-Verlag, 1996.

[14] Patrice Godefroid. Model checking for programming languages using Verisoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 174–186, January 1997. Paris, France.

[15] NASA Ames Research Group. http://ase.arc.nasa.gov/.

[16] K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In *Proc. First International Workshop on Runtime Verification (RV'01)*, volume 55 of *ENTCS*, pages 97–114, France, 2001. Elsevier Science.

[17] Klaus Havelund and Thomas Pressburger. Model checking JAVA programs using JAVA pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[18] IBM. http://www.ibm.com.

[19] Intel. IA-32 Intel Architecture Software Delevoper's Manual. http://www.intel.com/design/Pentium4/manuals/24547010.pdf.

[20] Release January. Java native interface specification.

[21] Jlint. http://artho.lcom/jlint.

[22] Jlint. http://www.ispras.ru/ knizhnik/jlint/readme.html.

[23] Java PathFinder (JPF). http://ase.arc.nasa.gov/jpf/.

[24] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-mac: a run-time assurance tool for java programs.

[25] kissme Java Virtual Machine. http://kissme.sourceforge.net/.

[26] T. Lindholm and A. Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison-Wesley, 1999.

[27] CACAO Java Virtual Machine. http://www.complang.tuwien.ac.at/java/cacao/.

[28] Japhar Java Virtual Machine. http://www.japhar.org.

[29] KAFFE Java Virtual Machine. http://www.kaffe.org.

[30] LaTTe Java Virtual Machine. http://www.cs.rochester.edu/u/wei/runtime.html.

[31] The Sable Java Virtual Machine. http://www.sablevm.com.

[32] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[33] POSIX. http://www.knosof.co.uk/posix.html.

[34] Native posix thread library (NPTL). http://people.redhat.com/drepper/nptl-design.pdf.

[35] MIT Pthreads. http://www.humanfactor.com/pthreads/mit-pthreads.html.

[36] 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.

[37] Spin. http://spinroot.com/spin/whatispin.html.

[38] R. Staerk, J. Schmid, and E. Boe<rger. *Java and the Java Virtual Machine*. Springer, 2001.

[39] S. D. Stoller. Model-Checking Multi-threaded Distributed Java Programs. In *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 224–244. Springer, 2000.

[40] Java Grande Forum Benchmark Suite. http://www.epcc.ed.ac.uk/javagrande/threads/contents.html.

[41] SUN. http://www.sun.com.

[42] A. S. Tanenbaum. *Modern operating systems*. Prentice Hall, 1992.

[43] Visual Threads. http://h18000.www1.hp.com/products/software/visualthreads/index.html.

[44] valgrind. http://developer.kde.org/ sewardj/.

[45] Verisoft. http://www.bell-labs.com/project/verisoft/.

[46] vmware. http://www.vmware.com.