Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Instrumenting Java Bytecode to Replay Execution Traces of Multithreaded Programs

Diploma Thesis

Marcel Christian Baur
baur@adbw.ch

Formal Methods Group, Computer Systems Institute,
Swiss Federal Institute of Technology (ETH Zürich)

Zürich, April 9, 2003

Abstract

Replay of program executions, that lead to a failure, can help developers to find software faults. While replay of deterministic executions of sequential programs can be easily achieved with existing debuggers, little support exists for deterministic replay of a multi-threaded program.

In Java, scheduling of threads depends on the algorithm implemented in the underlying virtual machine. A multi-threaded program is executed according to one of many possible schedules.

In this thesis a conceptual model to replay a specific thread schedule of a multi-threaded Java program is developed. A generic representation of an execution trace that contains all information for a deterministic replay of a given thread schedule was devised. Deterministic replay is achieved by modifying the bytecode of the original program such that only one thread at a time may runs according to the specified schedule. This allows deterministic replay on any virtual machine that conforms to the specification by Sun.

The model is implemented in **jreplay**, a software utility to enforce a given thread schedule in a compiled Java program. The application is implemented within the **JNuke** framework for checking Java bytecode. For this purpose, the framework was extended with a class writer and a bytecode instrumentation facility.

A number of experiments have been performed to validate the model and measure its efficiency.

Acknowledgements

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

My assistant *Viktor Schuppan* has contributed countless valuable ideas and without his help, I would probably not have gotten so far. Thank you for supervising my thesis.

Cyrille Artho was never tired to point out new errors in my source code and always gave useful feedback when asked for tiny details of the **JNuke** framework.

Pascal Eugster gave helpful advice how to use his virtual machine.

Christoph Kiefer proof-read some chapters during the last two weeks and suggested some minor changes to the final layout of the report.

Last but not least I would like to thank my girlfriend *Rahel Stalder* for the great support during the four months I was working on this thesis.

Table of Contents

Table of Contents 1

1 Introduction 2

1.1 Structure of this report . . . 3

2 Background and Preliminaries 5

2.1 Multithreading in Java . . . 5

2.2 The JNuke Framework . . . 7

2.3 Java class file format . . . 7

2.4 Summary . . . 9

3 Related Work 10

3.1 Deterministic replay . . . 10

3.2 Java bytecode instrumentation . . . 12

3.3 Random scheduling testing . . . 14

3.4 Summary . . . 15

4 Problem Description 16

4.1 Replay invariant . . . 16

4.2 Thread control . . . 17

4.3 Measuring correctness and efficiency of the future solution . . . 18

4.4 Summary . . . 19

5 Deterministic Replay 20

5.1 Methods to block and unblock a thread . . . 20

5.2 Transferring control between threads . . . 21

5.3 Creation of new thread objects . . . 23

5.4 Thread termination . . . 23

5.5 Implied thread changes . . . 24

5.6 Summary . . . 27

6 Instrumented Bytecode Transformations **28**

6.1 Rule 1: Initialization of the replayer 28

6.2 Rule 2: Instrumenting explicit thread changes 29

6.3 Rule 3: Registering new thread objects 30

6.4 Rule 4: Retaining the replay invariant for new threads 31

6.5 Rule 5: Terminating threads 32

6.6 Rule 6: Object.wait 32

6.7 Rule 7: Object.notify and notifyAll 33

6.8 Rule 8: Thread.interrupt 33

6.9 Rule 9: Thread.join 34

6.10 Summary 34

7 The Instrumentation Facility **35**

7.1 Design of the JNuke instrumentation facility 35

7.2 A two phase protocol for instrumenting bytecode 37

7.3 Selected problems of bytecode instrumentation 39

7.4 Summary 41

8 The Class File Writer **42**

8.1 Class writing 43

8.2 Extensions to the JNuke framework 46

8.3 Using the class writer object 47

8.4 Summary 48

9 Experiments **49**

9.1 Overhead of instrumented thread changes 50

9.2 Race condition 50

9.3 Deadlock 51

9.4 wait / notify 51

9.5 suspend / resume 51

9.6 Summary of test results 52

10 Evaluation and Discussion **53**

10.1 Summary 53

10.2 Known limitations 53

10.3 Future work 54

11 Conclusions **56**

A Usage of the jreplay Application **57**

B Layout of a Thread Replay Schedule File **60**

C Replayer **62**

C.1 Classes 62

C.2 Interface to the replayer 63

D Uninstrumented Java Example Programs **65**

D.1 closed loop 65

D.2 r1 66

D.3 deadlock 67

D.4 waitnotify 69

D.5 suspresume 71

E Programming Interface to Implemented Objects **73**

E.1 JNukeConstantPool 75

E.2 JNukeAttribute 84

E.3 JNukeAttributeBuilder 90

E.4 JNukeMFinfo 96

E.5 JNukeMFInfoFactory 101

E.6 JNukeClassWriter 105

E.7 JNukeBCPatchMap 110

E.8 JNukeBCPatch 113

E.9 JNukeInstrRule 116

E.10 JNukeInstrument 120

E.11 JNukeEvalFactory 128

E.12 JNukeExecFactory 129

E.13 JNukeTaggedSet 130

E.14 JNukeThreadChange 135

E.15 JNukeScheduleFileParser 142

E.16 JNukeReplayFacility 144

List of Figures **148**

Bibliography **151**

Chapter 1

Introduction

Although checking algorithms have been devised to discover potential faults in computer programs, repeated execution of an application is still a state-of-the-art debugging technique. A trace that leads to a failure is usually taken and executed step-by-step to understand the error and locate the fault. For this purpose, non-determinism must be eliminated from the program. For sequential programs, the main source of nondeterminism is input, which can be easily reproduced. In Java, multithreading adds another source of non-determinism: the order how threads are scheduled by the virtual machine. While some methods have been proposed to solve this problem (see Ch. 3), none has found wide-spread use in existing debugging tools. Thus, as concurrency related faults in multi-threaded programs may not show up every time an application is executed, debugging such applications with conventional debuggers is difficult.

Once sufficient conditions to reproduce a concurrency-related fault are known, a Java program can be modified in such a way that it deterministically exhibits this particular fault every time it is executed. A given thread schedule can be enforced by inserting calls to a replayer class that is dynamically linked to the erroneous Java program.

In this work, an application called **jreplay** was devised that takes the compiled class files of a Java application as input and instruments them. Calls to a replayer class are inserted at all locations where thread changes occurred according to a previously captured execution trace. Capturing of such thread schedules is not considered in this report. Instead, the list of thread changes to be instrumented is obtained from a text file. An increasing number of software products allow for formal verification and systematic testing of computer programs [1, App. B]. Many of these tools can produce executions traces that lead to a concurrency-related failure.

When the instrumented program is run, the replayer reads the same file with the thread schedule and enforces it by blocking and unblocking threads at runtime. The replayer therefore performs deterministic scheduling of the execution trace.

Representation of a thread replay schedule

The text file contains a sequential list of locations where thread changes occurred in the original program. A location is specified by combining class name, method index, and bytecode offset of the instruction that triggered the thread change. Each location is attributed with a unique identifier of the thread that should cease to run. Furthermore, an iteration index defines the number of successive invocations of the instruction – since the last thread change occurred – before the thread switch is accomplished.

Conceptual Model for Deterministic Replay

The virtual machine running the program can be forced to schedule threads in a specific order by blocking all threads except the current one given by the execution trace. During the replay of an instrumented program, a lock object is assigned to each thread. Threads are blocked and unblocked using these locks. To transfer control from one thread to another, the replayer first unblocks the next thread in the schedule and then blocks the current thread.

To differentiate threads, the replayer assigns a unique identifier to each thread that directly corresponds with the thread identifiers in the text files containing the replay schedule. Creation of new thread objects is instrumented so they are registered with the replayer at runtime. Instrumentation is also necessary at exit points where execution of a thread terminates.

During replay, some calls to methods in the `Thread` and `Object` classes lead to thread changes. While the actual thread change is performed as described, the calls to these methods need to be instrumented to avoid side-effects. Calls to `Object.wait` in the original program are substituted with calls to replayer methods that temporarily replace the lock object associated with a thread. Corresponding calls to `notify` or `notifyAll` can be removed in favor of instrumented thread changes. When the thread returns from `wait`, the lock object is reset to the default. Calls to `Thread.interrupt` are instrumented, such that the thread is interrupted later, when it is unblocked during a transfer. Finally, invocations of `Thread.join` need to be removed to avoid deadlocks.

Limitations of the model

Due to time constraints, the suggested model for replay has some limitations. It does not support applications that use thread groups or certain deprecated methods of the Java foundation classes. The replay model can be extended to support these features.

1.1 Structure of this report

The next chapter introduces multitasking and multithreading, explains some aspects of debugging nondeterministic programs and gives a short presentation of the **JNuke** framework on which the **jreplay** application is based. Chapter 3 provides an overview of related work and existing state-of-the-art solutions in the areas of deterministic replay and Java bytecode instrumentation. Chapter 4 defines the problem, lists necessary requirements for deterministic replay and establishes criteria for correctness and efficiency for the application to be developed.

Chapter 5 describes a conceptual model for deterministic replay and Chapter 6 presents the necessary bytecode transformations. Chapter 7 deals with the implementation details of a new bytecode instrumentation facility and Chapter 8 explains how the **JNuke** framework for checking Java bytecode was extended with a generic class file writer.

These two extensions to the framework were eventually used to implement the replay concept in the **jreplay** application. The results of some experiments with this application are presented in Chapter 9. Chapter 10 debates advantages and disadvantages of the replay model. It also presents some ideas for future work. Chapter 11 summarizes the efforts.

Usage of this utility is described in Appendix A. The format of text files containing the thread schedules is explained in Appendix B. The replayer classes are presented in Appendix C. Source code of the programs used in the experiments can be found in Ap-

pendix D. The report is closed with a reference in Appendix E that contains the programming interface to all implemented **JNuke** framework extensions.

Chapter 2

Background and Preliminaries

2.1 Multithreading in Java

2.1.1 Multitasking and Multithreading

Multitasking is a mode of operation that provides for the concurrent performance or interleaved execution of two or more computer tasks [35, p.750]. A *thread* is an execution context that is independently scheduled but shares a single address space with other threads [35, p.755]. The key difference between multitasking and multithreading lies in *memory protection*. While multiple threads within the same application may access shared resources, a process may not access variables of another process unless is uses an operating system interface specifically designed for this purpose.

A program is *nondeterministic*, if – for a given input – there may be situations where an arbitrary programming statement is succeeded by one of multiple follow-up states [24, Def. 4-3]. Programs with multiple threads are inherently nondeterministic [1]. Even with the same input, some faults may not appear every time an application is executed.

Sources of non-determinism in programs frequently are concurrent or parallel access to shared resources, asynchronous interrupts, user or file input, dependence on external conditions such as time, the state of underlying hardware, and the state of other applications.

Typical concurrency faults in multi-threaded applications include:

race conditions
: A *race condition* is a situation in which multiple threads access and manipulate shared data with the outcome dependent on the relative timing of the threads [35, p.752].

deadlocks
: A *deadlock* [9] is an impasse that occurs when multiple threads are waiting for the availability of a resource that will not become available because it is being held by another thread that is in a similar wait state [35, p.745].

livelocks
: A *livelock* is a condition in which two or more threads continuously change their state in response to changes in the other thread(s) without doing any useful work. A livelock is similar to a deadlock in that no progress is made but differs in that neither process is blocked or waiting for anything [35, p.748].

Java is a programming languages that has excellent built-in support for threads. Each Java program consists of at least one execution thread, usually called *main thread*, but the Java Virtual Machine (JVM) allows an arbitrary number of concurrently running threads.

2.1.2 Java Threading Models

Java threads are scheduled by the Java Virtual Machine. A threading model describes the implemented technique how threads are scheduled in a particular virtual machine. Unfortunately, the threading model depends on the JVM implementation and/or the host operating system, as it is not part of the Java language specification.

Virtual machines with *native threads* use the thread system of the underlying host operating system. *User threads* are based on a user-level thread implementation [34, Sec. 4.5.1]. Most virtual machine implementations use *Green Threads* that are optimized user threads that allow scheduling of different instruction streams [43, Sec. 4.2].

2.1.3 Creation of new threads

In Java, every thread is an object. In order for an object to become a thread, it must either extend the `java.lang.Thread` class or implement the `java.lang.Runnable` interface. As the `Thread` class itself implements the `Runnable` interface, it is sufficient to say that every object implementing the `Runnable` interface is a thread. A thread is started when its `start` method is executed.

2.1.4 Thread synchronization

Java offers a concept called *monitors* to prevent that two threads access the same resource at the same time. A monitor is a programming language construct providing abstract data types and mutually exclusive access to a set of procedures [35, p.749]. In Java, a statement block, method or class can be declared `synchronized`. Java associates a lock and a wait set with every object [18, Sec. 17.13/14]. Let `o` be an object. When entering a section that is synchronized on `o`, the current thread tries to acquire the lock ("enters the monitor") for `o`. If another thread already holds the lock, then the current thread is suspended until the thread releases the lock ("exits the monitor") by leaving the synchronized section.

By calling `o.wait` the current thread temporarily releases the locks it holds on `o` and is added to the wait set of `o`. It is suspended until another thread calls `o.notify`, `o.notifyAll` or if an optional specified amount of time has elapsed. `wait` can be useful if the current thread is waiting for a certain condition that can only be met by another thread that needs access to the monitor. When the waiting thread resumes execution, the locks are automatically reacquired. As it is not guaranteed that the condition has been met, `wait` is often called within a `while` loop.

Note that if `t` is a `Thread` object, then calling `t.wait` does not necessarily suspend the execution of `t`. Instead, the *current thread* is suspended until another thread calls `t.notify` or `t.notifyAll`. Only in the case that `t` is the current thread, calling `t.wait` is equivalent to `this.wait` and `t` is suspended by itself.

`o.notifyAll` wakes up all threads in the wait set of `o`. It is used when it cannot be guaranteed that each thread in the wait set of `o` can continue execution.

2.2 The JNuke Framework

The Formal Methods Group has developed a framework called **JNuke** for checking Java bytecode [15]. The framework is written in the C programming language. It has a loader that is capable of parsing compiled Java classes. Each loaded class is represented in memory using a `JNukeClass` descriptor. For each field and method it contains a reference to a corresponding **JNuke** descriptor object. Each method descriptor stores bytecode instructions and exception handlers.

The loader can also transform a class. The `JNukeBCT` interface describes such a bytecode transformation. One implementation of this interface is `JNukeBCTrans`. It can transform Java bytecode into two optimized, but functionally equivalent representations, called *abstract bytecode* and *register based bytecode*. Register bytecode inlines subroutines and eliminates the operand stack by replacing stack operations with register operations [13].

The **JNuke** framework is equipped with a simple interpreter for register based bytecode [16], that was recently rewritten into a full-featured virtual machine [14]. The virtual machine now includes a rollback mechanism inspired by Rivet [6] that can be used for systematic testing of multithreaded Java programs. In the future, this virtual machine could be used to find thread schedules that lead to concurrency faults. These execution traces can then serve as input for the **jreplay** application described in Appendix A.

2.3 Java class file format

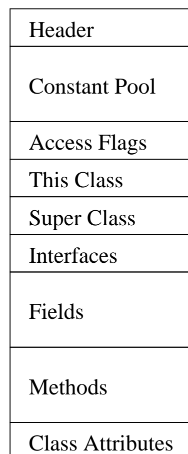
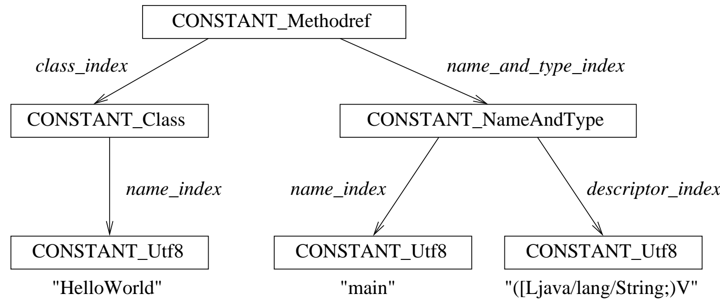


Figure 2.1: Structure of a Java class file

As Java programs were designed to run on different computer architectures, the JVM specification provides a universal class file format [28, Ch. 4], shown in Figure 2.1. Java class files use a big-endian notation (high byte first) for all integer values exceeding the size of a single byte.

The Java class file starts with a small *header* that consists of a two-byte magic value and file format version information.

A *constant pool* that contains numbered entries of variable type and size follows directly after the header. The first item in the constant pool has index 1, subsequent entries have higher indices. Figure 2.2 shows how these indices can be used to create tree structures. The type of each entry is defined using a tag byte [28, Table 4.3].



idx	tag byte	value
1	CONSTANT_Utf8	"HelloWorld"
2	CONSTANT_Utf8	"([Ljava/lang/String;)V"
3	CONSTANT_Utf8	"main"
4	CONSTANT_Class	class_index=1
5	CONSTANT_NameAndType	name_index=3, descriptor_index=2
6	CONSTANT_Methodref	class_index=4, name_and_type_index=5

Figure 2.2: Storing hierarchical information about a method in the constant pool of a class file

The middle part of a class file consists of *Access Flags* for the class contained in the file. The *This Class* field refers to a constant pool entry of type `CONSTANT_Class` defining the class in the file. If the class directly extends `java.lang.Object`, the *Super Class* field has value zero. Otherwise, the entry refers to a constant pool entry describing the direct superclass.

The *Interface* table contains a list of interfaces the class implements. Again, each implemented interface is represented by an appropriate index to a `CONSTANT_Class` entry in the constant pool.

Information about the fields and methods of a class are stored in `method_info` and `field_info` data structures. They are hierarchically built from attributes. An attribute is a named data buffer of variable size. Figure 2.3 shows a list of predefined attributes that has to be recognized by every Java virtual machine [28].

Code	Contains operand stack and local variable array size limits, the byte code stream of a method, and an exception table. If the class was compiled with debug information, it may also contain the sub-attributes **LineNumberTable** (source code line numbers) and **LocalVariableTable** (names of local variables)
ConstantValue	Used for final fields
Deprecated	Indicates that the item should no longer be used
Exceptions	List of exceptions a method may throw
InnerClasses	Inner classes of the class file
SourceFile	Source file name of the class
Synthetic	Indicates that a structure was generated by the compiler.

Figure 2.3: Standardized attributes of a Java class file

2.4 Summary

Execution of multi-threaded programs is nondeterministic as it depends on the scheduler of the JVM running the program. Unfortunate scheduling can lead to concurrency faults in badly written programs.

The Java language has excellent support for threads. It allows synchronization of threads through monitors. Method `wait` can be used to temporarily release a lock. A waiting thread is resumed when another thread calls `notify` or `notifyAll`.

Java classes are stored in standardized class files. Hierarchical structures can be stored in a constant pool. Information about methods and fields are stored in `method_info` and `field_info` data structures that are built from named data buffers of variable size, called attributes.

The class loader in the **JNuke** framework provides an interface that allows to implement arbitrary bytecode transformations.

Chapter 3

Related Work

3.1 Deterministic replay

Debugging tools for sequential programs have been optimized in the past. Little support exists for debugging parallel programs because of non-determinism introduced by the thread scheduler. One approach that allows using the same set of debugging tools for both sequential and parallel programs is to *capture* a history of events during a record phase by monitoring the program. The history of events can then be used to control re-execution in a conventional debugger for the purpose of *replay* [29].

A particular problem of this approach lies in the precise definition of what and how much information should be captured. Suggested events include shared memory accesses, thread change information, operating system signals, and interprocess communication. Another problem is that traces of a program execution frequently become very large. The amount of information to be captured depends on how the event history is going to be used [29, Sec. 2.2]. A capture system should only store the minimally information required for replay. Several approaches have been made to optimize the size of traces [32]. Transformations can also be applied to an event history in order to reduce its size [29, Sec. 2.4].

Approaches to capture a thread schedule for later replay fall into two classes: The relative number of instructions between two thread changes can be counted or the order how shared resources are accessed can be recorded. An optimal adaptive tracing technique for the second method is presented in [32]. It requires information whether a particular variable is private or shared.

All systems for deterministic replay described in this section assume a single-process environment. Replay of distributed parallel systems is not considered in this report, but is covered in [31] and [23].

3.1.1 Leblanc, Mellor-Crummey: InstantReplay

In 1987, LeBlanc and Mellor-Crummey developed *InstantReplay* [26], a piece of software that records the relative order in which successive operations are performed on objects. A version number is associated with each shared variable and by tracking which thread accesses which version of a resource, an event history may be obtained. They assume that shared variables are protected by multiple-reader single-writer locks for efficiency reasons. Although their model is a simplification of the real world, their work has set the base for

future capture/replay system. It is considered the origin of the "replay" terminology [41, footnote 1 p.45].

3.1.2 Tai, Carver, Obaid: Replay of synchronization sequences in Ada

For introducing concurrency into a program, the Ada language offers the concept of an *Ada task* object derived from a task type, which acts as a template. As Ada provides no conventional synchronization – such as semaphores or monitors – for controlling access to shared variables, it is assumed that access to shared variables is always protected [41, p.47].

The authors present a language-based approach for the deterministic replay of concurrent programs written in Ada [41, pp.45]. Their solution is language-based in the sense that it works completely on the source level of a program. One reason for this decision was to avoid portability problems. Working directly on the Ada source code also removes the need of system-dependent debugging tools.

Although a tool is introduced that instruments a concurrent Ada program such that synchronization operations are collected, the main topic of their paper is deterministic replay of *synchronization sequences*, i.e., total or partial orderings of all synchronization events of the tasks in an Ada program. A synchronization sequence provides the minimum amount of information necessary for deterministic execution. For the purpose of replay, a new Ada task is instrumented into the program that controls execution by synchronizing the original tasks.

In their paper *TDCAda*, an environment for testing and debugging concurrent Ada programs, is presented. The framework is written in C and serves similar purposes for Ada source programs as our **JNuke** framework does for compiled Java classes.

3.1.3 Russinovich and Cogwell: Extensions to the Mach OS

Russinovich and Cogwell [33] present an algorithm for efficient replay of concurrent shared-memory applications. Their technique is based on counting the number of instructions between two thread changes.

In their approach, a register of the processor is reserved for later use when the program to be replayed is being compiled. An instrumentation setup written in the Awk programming language is used to realize an instruction counter in the reserved register.

By replacing the Mach system libraries with an instrumented version, the scheduling of applications can be monitored or recreated. For initialization a call to a special method is inserted before the main method of the program is executed. The approach in this thesis uses a similar call to set up the replay system at run time.

Their approach has some disadvantages. First, it is heavily based on the Mach 3.0 operating system and its modified system libraries. Their solution can only be applied to operating systems whose source code is available. Second, the need to reserve a processor register for later use as an instruction counter introduces a dependency to a particular feature of a compiler. Not all compilers may provide a possibility that allows reservation of a specific register. Finally, the proposed method does not work on systems with more than one processor or in systems that take input from external processes.

3.1.4 Choi and Srinivasan: DejaVu

In their paper *Deterministic Replay of Java Multi-threaded Applications* [7], J. Choi and H. Srinivasan present a record/replay tool called *DejaVu* that is able to capture a thread schedule of a Java program and allows deterministic replay.

Their solution is independent of the underlying operating system. It is based, however, on a modified Java virtual machine [7, Sec. 5], which makes it basically impossible to analyze the behavior of a program under a specific thread schedule in a third party Java debugger.

It was briefly considered by the authors to modify the application byte code instead of the virtual machine. This is the approach taken in this thesis. Their decision to modify the virtual machine was based on the claim that the introduced instrumentation overhead would have been a lot higher. This observation is contradictory to the results obtained in Chapter 9.

Another difference between their approach and the solution proposed in this thesis lies in the notation of the thread schedule to be replayed. Their notation of a *logical thread schedule interval* is based on counting the relative number of *critical events* (shared variable accesses and synchronization events) between two thread changes [7, Sec. 2].

The method in this report is to use absolute coordinates of thread changes supplemented by an iteration index. This decision was influenced by the observation that thread schedules with absolute positions are easier to read for humans, especially in large programs.

3.2 Java bytecode instrumentation

Several projects deal with instrumenting bytecode. For the rest of this chapter, some existing work in this area will be presented. The list is not intended to be exhaustive.

3.2.1 Java Object Instrumentation Environment (JOIE)

Most operating systems and programming environments support late binding, i.e., dynamic linking of library routines at load time of a binary executable. In Java locating and verifying additional software units is done by the class loader. The *Java Object Instrumentation Environment* (JOIE) [10] is a toolkit to augment class loaders with load-time transformation dynamic behavior, developed at Duke University.

Load-time transformation extends the concept of dynamic linking, in which the loader is responsible not only for locating the class, but for transforming it in user-defined ways. While late binding of libraries does not alter the semantics of a program, the JOIE class loader allows the installation of bytecode transformers. A transformer can insert or remove bytecode instruction and generally alter the class file to be loaded. The JOIE toolkit therefore allows dynamically instrumenting Java class files at runtime.

3.2.2 Bytecode instrumentation Tool (BIT)

The *Bytecode Instrumentation Tool (BIT)* [27] is a collection of Java classes that allows to insert calls to analysis methods anywhere in the bytecode, such that information can be extracted from the user program while it is being executed.

BIT was successfully used to rearrange procedures and to reorganize data stored in Java class files. An application called *ProfBuilder* was built around BIT that allows for rapid construction of different types of Java profilers.

3.2.3 JavaClass API / Bytecode Engineering Library (BCEL)

The *JavaClass API* was designed as a general purpose tool for bytecode engineering. It was later renamed to *Bytecode Engineering Library (BCEL)* [12].

BCEL was designed to model bytecode in a completely object-oriented way by mapping each part of a class file to a corresponding object. Applications can use BCEL to parse class files using the visitor design pattern [17]. Particular bytecode instructions may be inserted or deleted by using instruction lists. Searching for instructions in such lists can be done using a powerful language based on regular expressions. Applying changes to existing class files can be realized using generic objects.

Efficient bytecode transformations can be realized by using *compound instructions* – substitute for a whole set of instructions of the same category. For example, an artificial `push` instruction can be used to push arbitrary integer values to the operand stack. The framework then automatically uses the most compact instruction.

Transparent runtime instrumentation can be done by using custom class loaders, as in JOIE. With the aid of runtime reflection, i.e., meta-level programming, the bytecode of a method can be reloaded at run time.

The *Bytecode Engineering Library* library was used at a very early stage of this project. Parts of the design of the constant pool object and the usage of compound instructions were adapted in the instrumentation facility presented in Chapter 7.

3.2.4 Barat

Barat [3, 5] is a front-end for Java capable of generating type-annotated syntax trees from Java source files. These can then be traversed using the visitor design pattern to reorganize code at the source level. It also allows parsing of comments in the source code.

Barat does not decompile bytecode, but the modified source code can be easily recompiled allowing compile-time reflection of Java programs.

Barat is based on *Poor Man's Genericity for Java* [4] and is available under the BSD License. It was named after the western part of the Java island.

3.2.5 JTrek

JTrek developed at Digital Equipment Corporation [1] consists of the *Trek* class library, that provides features to examine and modify Java class files. It also includes a set of console applications based on the *Trek* library.

JTrek allows the writing of three types of programs [11]: *Coding-time applications* parse a class file and generate custom reports. *Run-time applications* can be used to instrument a Java program. *Tuning applications* modify the bytecode of a Java program so it runs faster, e.g., by removing debugging assertions.

[1] Digital has meanwhile merged with Compaq and Hewlett-Packard

3.2.6 CFParse

CFParse [19] is a minimal low-level API developed by IBM to interactively edit the attributes and bytecode of Java class files. It has been designed to be type-safe and to reflect the underlying physical structure of a class file as much as possible.

3.2.7 Jikes ByteCode Toolkit

The *Jikes ByteCode Toolkit* [20] tries to maintain an abstract object-oriented representation by hiding the physical details of a Java class file. References to the constant pool are immediately resolved and they are transparently regenerated when saving the modified class file back to disk. For the representation of constant values, the toolkit automatically chooses the optimal instruction. A peep-hole code optimizer is also included. Object relationships allow to detect the instruction where a specific method is invoked, or to detect the set of instructions where a given field is accessed. The internal toolkit classes may be extended by user classes. This allows adding new features or modifying certain aspects of the *Jikes* toolkit.

The toolkit comes with the *javaSpy* application that instruments an application so its execution can be observed.

3.2.8 Jasmin Assembler

Jasmin [30] is a Java Assembler. It takes as input textual representations of Java bytecode programs written in a simple assembler-like proprietary syntax. Although it does not directly allow changing existing bytecode programs through instrumentation, it may be useful for writing custom class files from scratch.

As Sun Microsystems never released a standard for Java bytecode assembler programs, sources written for *Jasmin* cannot be reused in other applications. The previously mentioned *BCEL* project comes with a *JasminVisitor* sample application that can decompile Java class files into the assembler source file format required by *Jasmin*.

3.3 Random scheduling testing

3.3.1 rstest

To verify correctness of a Java program, calls to a randomized scheduling function can be inserted at selected points. Once the program was instrumented, the transformed program is executed repeatedly to test it.

In the case of *rstest* [36], instrumentation is done using the *BCEL* library. The locations where calls are inserted include all synchronization operations as well as object and array access instructions. Static analysis is used to avoid instrumenting calls for accessed objects that have not yet escaped the thread that created them. Storage locations can be classified as unshared, protected or unprotected. This allows to reduce the number of operations that need to be instrumented.

The scheduling function either does nothing or causes context switches using the methods `Thread.sleep` or `Thread.yield`. The choice is blindly made using a pseudo-random number generator.

3.4 Summary

Thread schedules can be captured by monitoring access to every shared resource. Tracing which thread accesses which successive version of a variable allows to compile an event history. Another possibility is to count the relative number of instructions between thread changes. This requires to instrument a program counter.

The presented solutions for deterministic replay either depend on modified system libraries or the virtual machine. Complementing bytecode with synchronization methods offered by the Java language has the advantage that replay can take place on every JVM conforming to the standard by Sun.

Several generic frameworks for instrumenting Java bytecode exists. While some toolkits were designed to closely reflect the physical structure of a class file, other applications perform exactly the opposite and try to abstract as much as possible. Inserting calls to a scheduling function allows randomized testing of multi-threaded programs.

Chapter 4

Problem Description

The goal of this thesis is to develop a tool for deterministic replay of multithreaded Java programs. A concept to modify a Java program to enforce a given schedule on any virtual machine complying to the JVM specifications by Sun should be devised. This has the advantage that any debugging tool can be used to locate faults in the program. The **JNuke** framework should be extended with an instrumentation facility and a class writer that automatically perform the requested modification.

4.1 Replay invariant

A possible solution to enforce a given thread schedule cannot rely on the thread support of the underlying operating system or the particular JVM implementation running the program. The only way is to override the JVM scheduler by forcing it to exactly replay the scheduling given in the execution trace. This can be achieved by using the synchronization mechanisms offered by the Java language to block and unblock threads. The program is thus transformed from a nondeterministic multi-threaded program into a deterministic sequential program. The sequential program is basically multi-threaded, but all threads except the current thread in the replay schedule are blocked by the instrumented replay mechanism.

From the point of view of the JVM Scheduler, there must always be at most one thread that can be chosen for execution when replaying the instrumented program. If a thread switch should not occur at the current program location, all other threads must stay in a blocked state. This is our replay invariant. If the JVM scheduler can choose among two or more threads, the transformed program is no longer single-threaded and deterministic replay may fail. If on the other hand there is no thread ready to be scheduled, the program to be replayed will enter a deadlock situation. It is thus mandatory for the replayer to handle all situations where threads are created or destroyed as well as contexts in which a transfer of control flow to another thread naturally occurs or should artificially be inserted according to the external schedule.

4.2 Thread control

4.2.1 Controlling transfer between threads

A method needs to be devised that transfers control from the current thread to the next thread in the replay schedule.

Figure 4.1 shows an example involving two threads. In the beginning, only Thread 1 is running due to the replay invariant. If a thread change should occur according to the thread schedule to be replayed, the transfer method of the replayer is invoked because of an instrumented call (A). After the replay mechanism has determined the next thread in the schedule (B), the actual transfer is achieved by unblocking the next thread in the replay schedule and blocking the current thread (C). In the example, Thread 2 is scheduled next and is free to run while Thread 1 remains blocked (D).

Assume now that control should be transferred back to the first thread. The instrumentation facility has inserted a call to the transfer method of the replayer at the appropriate location in the code of Thread 2 (E). The replayer again queries the replay schedule to determine the next thread (F). The current thread is blocked and the original thread is resumed (G). Transfer in both directions can be achieved in the same way.

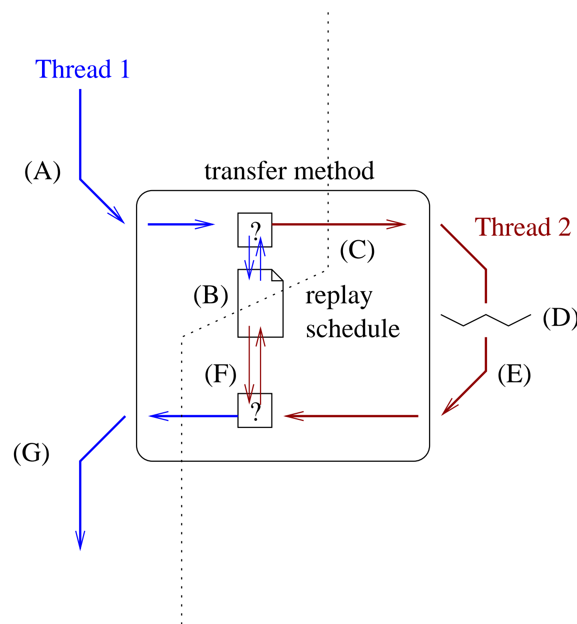


Figure 4.1: Thread switch

4.2.2 Thread schedule

The thread schedule must be unambiguous, concise and complete. Replay will fail if the exact positions where a thread change occurred cannot be determined or if some thread changes are omitted in the thread schedule.

The number of threads in an application is a dynamic property. Creation of new threads must be detected, so they can be blocked until the moment they are scheduled for the first time. The replayer must unblock the next thread in the schedule immediately before the current thread dies, otherwise a deadlock occurs.

The virtual machine automatically assigns a name to each thread. This name can be modified by the program. To differentiate threads, the replayer must therefore assign a unique identifier to each thread.

4.2.3 Class Files

A Java application may consist of multiple class files. The instrumentation tool must be able to detect and instrument all relevant classes. Each class file should be instrumented at most once. Missing class files or native methods cannot be instrumented.

New threads are not only created in the class that contains the `main` method of the application. Rather, method calls and field accesses need to be followed transitively. Inheritance and static features need to be considered as well.

4.3 Measuring correctness and efficiency of the future solution

4.3.1 Criteria for correctness

Instrumented programs should behave like the uninstrumented versions and follow precisely the schedule given. Side-effects of the replay mechanism should be avoided. Ideally, they should exhibit the same faults when they are replayed. A correct implementation further complies with the following terms:

- Class files generated by the instrumentation utility must be formally correct, in particular they may not be rejected by the Java verifier.
- By inserting instructions into the byte stream, the absolute address of other instructions are shifted. As a result, targets of branch instructions may change and need to be updated. The instrumentation facility must analyze and possibly update the arguments of all instructions that take relative offsets as arguments. Branch targets may no longer be addressable in narrow format instructions, if too many instructions are inserted between the instruction and the branch target.
- Replay of the instrumented program should be transparent to the user. Line number information in the instrumented class must be retained such that debuggers can map the instrumented bytecode to the original uninstrumented source program.

4.3.2 Criteria for efficiency

Both instrumentation of the bytecode and replay should be as fast as possible. Only necessary calls to the replay mechanism should be inserted. The program to be replayed must be reinstrumented whenever the replay schedule is changed.

Obsolete bytecode should be removed and not overwritten with `nop` instructions. The most compact operation should be chosen to push constant values to the stack. Depending on the actual relative offset, narrow formats of branch instructions should be taken, if possible.

4.4 Summary

A multi-threaded program can be transformed into an equivalent program where only one thread is ready to run at a time. The JVM then automatically schedules this thread. The other threads are blocked using thread synchronization methods offered by Java. Control can be transferred from one thread to another by first unblocking the next thread in the replay schedule and then blocking the current thread.

During replay, new threads must be blocked until they are scheduled for the first time. When a thread dies, it must not be blocked during its last control transfer, otherwise a deadlock occurs.

The utility to be developed must assure that all necessary classes of an application are transitively instrumented. Generated class files must be correct and both the utility and the resulting code should be as efficient as possible.

Chapter 5

Deterministic Replay

This chapter presents a conceptual model for deterministic replay. The necessary transformations on the bytecode level are explained in Chapter 6. Section 5.1 presents mechanisms for blocking and unblocking threads offered by Java. Section 5.2 shows how a transfer from the current thread to the next thread in the replay schedule can be achieved using `wait` and `notifyAll`. Section 5.3 and 5.4 deal with situations where new threads are created or existing threads end. Finally, some more substitutes for statements that imply a thread switch are considered in section 5.5, to avoid side-effects during replay.

5.1 Methods to block and unblock a thread

The replayer needs to block and unblock a specific thread, depending on the thread schedule to be enforced. Java offers several ways to temporarily suspend and resume the execution of a thread [22, p.218]. In the proposed solution, each thread is associated with a unique private lock object. A thread is blocked an unblocked by calling `wait` and `notifyAll` on that object.

When `wait` is executed, the current thread is blocked. If present, the next thread will be scheduled for execution. The choice of the next thread depends on the implementation of the scheduler. With the exception of Realtime Java [42], no direct interface to the scheduler is currently provided. When only the next thread in the replay schedule is ready for execution, the scheduler will choose it when the current thread is blocked.

Before explaining how control can be transferred from one thread to another using the pairing of `wait` and `notifyAll`, let us briefly discuss some of the other possibilities for blocking a thread.

Thread.yield and Thread.sleep

Method `yield` causes the current thread to temporarily pause and allows other threads with the same priority to execute. It cannot be used to pass control among threads in applications where threads have different priorities [37].

A thread can be put to sleep for a set amount of time using the `sleep` method. The argument to `sleep` specifies only a lower bound to the time after which execution of the sleeping thread will automatically be resumed. In addition, the exact amount of time a thread has to be blocked depends on the machine the program is replayed on.

Blocking a thread can be realized either by doing nothing, by calling `yield` or by calling `sleep` in a closed loop (busy waiting, [35, p.744]).

Thread.suspend

Execution of a thread can be paused using `Thread.suspend`. A corresponding call to `Thread.resume` will unblock the thread. `suspend` and `resume` are deadlock-prone and have been deprecated [40].

Thread.join

If `t` is a thread, calling `t.join` suspends the current thread until `t` dies. The only way to resume such a blocked thread is to interrupt it.

5.2 Transferring control between threads

Methods `wait` and `notifyAll` can be used to implement block and unblock operations that suspend and resume execution of a thread. Figure 5.1 shows how execution of Thread t_1 is passed to the replayer due to an instrumented call to a transfer routine `sync`. An explicit control transfer is realized by unblocking the next thread in the schedule followed by blocking the current thread. By blocking the current thread, the JVM scheduler chooses the next available thread for execution. As t_2 has been previously unblocked and all other threads are blocked due to the replay invariant, the application continues with thread t_2.

A direct implication of this design is that threads are always unblocked by another thread and always blocked by themselves. In this example, t_2 is unblocked by t_1, that then blocks itself. The next thread must be unblocked before the current thread is blocked, otherwise a deadlock will occur.

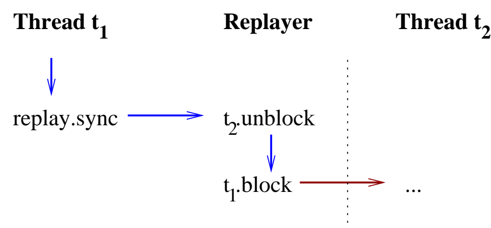


Figure 5.1: Control flow in a successful transfer from thread t_1 to t_2

5.2.1 Implementing `block`

Unblocking t_2 before blocking t_1 temporarily violates the replay invariant as shown in Figure 5.2. After unblocking t_2 (A), the JVM scheduler can immediately choose it for execution before t_1 is blocked (B). The second thread is then free to run (C). If control is transferred back to the first thread, t_1 is unblocked (D) and t_2 blocks itself. Note that t_1 did not yet have the opportunity to block itself. The `unblock` call (D) therefore has no effect other than clearing the `blocked` flag of t_1. Once the interrupted transfer (B) is resumed, t_1 immediately blocks itself. A deadlock occurs (B/E).

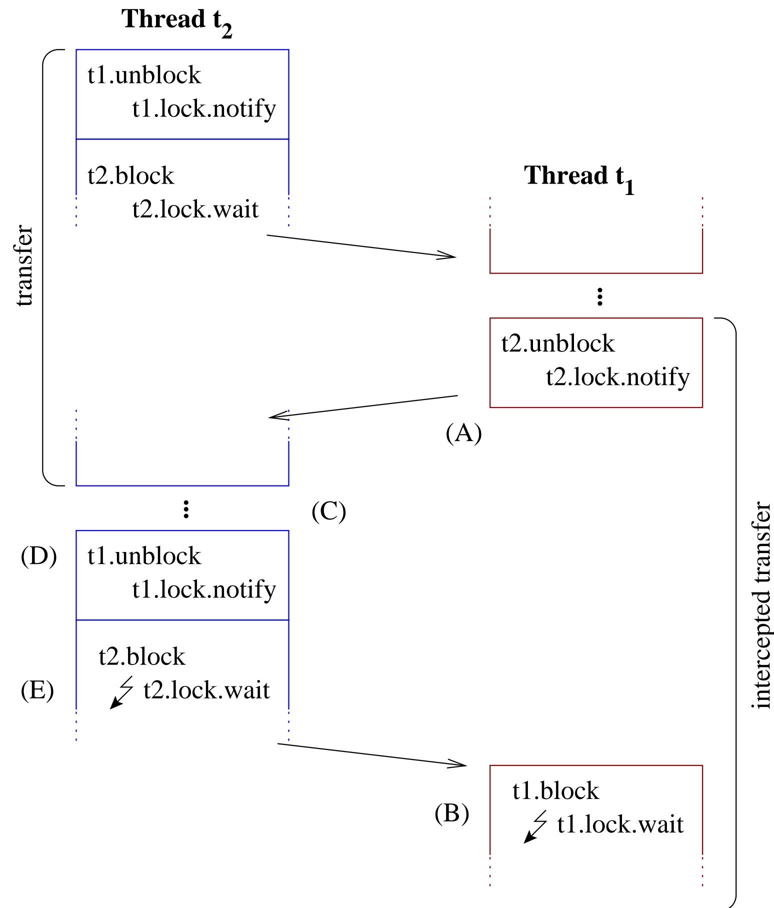


Figure 5.2: Control transfer from Thread t_1 to Thread t_2 intercepted by the JVM scheduler

For this reason, a boolean variable `blocked` keeps track of the last lock operation. The `blocked` variable is used as a binary semaphore [25, Sec. 3.4]. The `block` method examines its `blocked` flag once it has entered the monitor. This flag will be set, except in the case where the transfer method was intercepted by the JVM scheduler. In that case a call to the `block` method has been superseded by a call to `unblock`. the call to `block` is obsolete, and the `wait` is not performed. The `blocked` flag is restored for the next control transfer involving this thread. Figure 5.3 shows the Java code implementing the `block` operation.

```
1   public void block() {
2       synchronized(lock) {
3           while (blocked) {
4               try {
5                  lock.wait();
6               } catch (InterruptedException e) {
7                  System.out.println(e.getMessage());
8               }
9           }
10          blocked = true;
11      }
12  }
```

Figure 5.3: Implementation of a method to block a thread

5.2.2 Implementing `unblock`

Implementation of the corresponding `unblock` operation is shown in Figure 5.4. The boolean variable `blocked` is set back to false to indicate the last operation and a `notifyAll` on the lock of the thread is sent.

As every thread is waiting on its own lock, there can be at most one object waiting in the monitor listening to the notify event. Calling `notify` would therefore be sufficient and instead of a `while` loop, an `if` statement could be used in Figure 5.3. The reason for choosing `while` and `notifyAll` will be explained later in Section 5.5.2.

```
1   public void unblock() {
2       synchronized(lock) {
3           blocked = false;
4           lock.notifyAll();
5       }
6   }
```

Figure 5.4: Implementation of a method to unblock a thread

5.3 Creation of new thread objects

The replayer maintains a data structure for every thread. This data structure contains a unique identifier, a reference to the `Thread` object it refers to, and the previously discussed `lock` object for the thread. As soon as a new thread is created, the replayer will create a data structure for it. The main thread of an application receives the value zero as unique identifier. The first thread to be created receives an identifier of one, subsequent threads receive higher identifiers.

If a thread is started, the thread object has already been registered with the replayer when its instance was created. If the `start` method of a thread object is never called, the memory required for one data structure is wasted.

When a new thread becomes ready to run after its `start` method was invoked, it must be immediately blocked, otherwise the scheduler might choose it for execution. The instrumentation utility inserts a call to a replayer method at the very beginning of each `run` method. The replayer immediately gains control over new threads independently of when and where a thread is started. The replayer can then decide depending on the thread replay schedule whether it should block the new thread and transfer control back to the intercepted thread or if it should let the new thread continue execution.

5.4 Thread termination

If a thread finishes its `run` method, it terminates and the JVM scheduler implicitly chooses the next ready thread for execution. This thread change will be instrumented by a transfer operation at the thread exit point. This will block the current thread before it can execute its last instruction. As the thread died in the original program, it will never be rescheduled in the instrumented program and is trapped in its blocked state. Applications waiting until all their threads died would never terminate when replayed.

The replayer must therefore be notified to release the current thread in the schedule before the instrumented thread switch occurs. This can be achieved by inserting an additional unblock operation directly before the instrumented thread switch. By clearing the `blocked` flag, the transfer method is instructed not to block the ending thread and it may die.

Uncaught exceptions

An application may abruptly terminate when an error or runtime exception [18, Sec. 11.3.1] occurs that is not caught. If other threads are still runnable, then this thread change must appear in the schedule during instrumentation, otherwise a deadlock will occur during replay. The mechanism used for capturing the execution trace to be replayed must therefore log this thread change.

Daemon threads

A thread may be declared a daemon thread by calling `Thread.setDaemon(true)` before the thread is started. If only daemon threads remain, the JVM will terminate the program. During replay, no side-effects occur when instrumenting applications that use daemon threads.

VM shutdown with registered hooks

By calling `System.exit`, `Runtime.exit`, or `Runtime.halt`, the shutdown sequence of the Java virtual machine is initiated. Prior to terminating the JVM, registered shutdown hooks will be started. Shutdown hooks are initialized thread object that have not yet been started. Furthermore, the thread must be registered using the `addShutdownHook` method. For the replayer, shutdown hooks are normal threads, that are just started at the latest possible point in a program.

5.5 Implied thread changes

Some elements of the Java language, e.g., statements that suspend the execution of a thread, may imply a thread switch. In the case that a multi-threaded application calls `Thread.sleep(500)`, the current thread might be suspended and another thread may be scheduled instead. After 500 ms have passed, the original thread can continue. In this example, calling `sleep` triggered two thread changes. If a single-threaded application calls `Thread.sleep(500)` to pause execution for 500 ms, no thread switch will occur as there are no other threads. The replayer has no possibility to determine whether a particular call to `sleep` should trigger a thread change at runtime. If a thread change occurs, it must know which thread has to be scheduled next. For these reasons, implied thread changes must be explicitly recorded in the replay schedule.

If a thread change appears in the thread replay schedule, a transfer is instrumented into the program. As a direct consequence, the thread change is no longer implied, but turned into an explicit thread switch. If the instrumented program is now replayed, the thread change may occur twice – once because of the instrumented thread switch and the second time by replaying the original instruction. During instrumentation, the original statement that triggered the implied thread change must be replaced. The most common instructions that imply thread changes are discussed below.

5.5.1 Object.wait

Method `wait` comes in three forms shown in Figure 5.5, each with a different number of parameters. Consider the case where there is only one thread or all other threads in the original program cannot be scheduled for some reason. If `wait` is called with no

parameter, no thread change occurs and the program will deadlock. If `wait` is called with a timeout parameter, the thread that called `wait` is continued after the specified time has passed. Again, a thread change cannot occur.

```
void wait()
void wait(long millis)
void wait(long millis, int nanos)
```

Figure 5.5: Three ways to call Object.wait

The idea of the transformation on the source level is given in Figure 5.6. If a thread change occurred at some stage in the original program, it will be enforced with a call to the transfer method that is inserted precisely between two instrumented calls to `setLock` and `resetLock`. These two methods are used to temporarily change the lock associated with the thread. Let `o` be any object. A call to `o.wait` is removed and the transfer method temporarily waits on `o` instead of the default lock associated with the thread. The original lock is automatically restored after the thread is continued.

If on the other hand no thread change was taken, the call to `sync` is omitted. To simplify the implementation, the calls to `setLock` and `resetLock` are nevertheless instrumented.

Original program		Instrumented program
synchronized (o) {		synchronized (o) {
// ... 1 ...		// ... 1 ...
		replay.setLock(o);
o.wait();	⇒	**replay.sync(…);**
		replay.resetLock();
// ... 2 ...		// ... 2 ...
}		}

Figure 5.6: Equivalent transformation of `wait` at the source level

As the original call to `wait` is removed, the system will no longer pause for an optionally specified time. This side-effect could be avoided by inserting a call to `Thread.sleep`.

5.5.2 Object.notify and notifyAll

A call to `o.notify` wakes up one thread that is waiting in the monitor of `o`. If there are multiple threads in the wait set of `o`, only one of them is awakened. As `notify` is called within a synchronized section, the awakened thread will not be able to proceed until the notifying thread releases its current lock. A switch to the waiting thread therefore usually occurs when the notifying thread leaves the synchronized section.

If a thread change occurs, it is reflected in the thread replay schedule and a transfer to the correct thread will be instrumented by inserting a call to `sync` at the corresponding location.

The previously discussed instrumentation for `wait` temporarily sets the lock object to the original monitor in the uninstrumented program. The inserted call to `sync` blocks on the original object instead of the default lock associated with the thread. When a thread is continued because another thread called the transfer method replacing the `notify`, then the lock object in the lock object associated with the thread is still the temporary lock. The corresponding transfer will therefore unblock the old thread by calling `notify` on the original lock object.

If two threads wait on the same object, they are also waiting on the same lock. The `unblock` method presented in Figure 5.4 therefore uses `notifyAll` to wake up one of the waiting threads. Only the thread to be resumed will have its `blocked` flag cleared. The other threads waiting on the same lock go back to sleep as the `wait` is called within a loop.

A call to `o.notifyAll` wakes up all threads that are waiting on the monitor of `o`. For deterministic replay, there is no difference between `notify` and `notifyAll`.

5.5.3 Thread.interrupt

Let `t` be a thread that is blocked because it called `Object.wait`. If another thread calls `t.interrupt`, an `InterruptedException` is thrown. By replacing every call to `t.interrupt` with `replay.interruptSelf(t)`, the `interruptSelf` flag associated with thread `t` is set. The extended implementation of `block` shown in Figure 5.7 checks a flag to determine whether such an interruption should be rethrown.

A thread can also be interrupted while it is suspended due to a call to `Thread.join` or `Thread.sleep`. While comparing the current lock and the original lock of the thread reveals that a thread was suspended in a `wait`, detecting if a thread returns from a `join` or `sleep` is currently not possible. The substitute would be the same as for `wait`.

If the thread was not suspended, then the normal invocation of `interrupt` would set the interrupted flag of the thread. To simulate this behavior, the `block` method performs a substitute call to `interrupt`.

```
1   public void block() {
2       synchronized(lock) {
3           while (blocked) {
4               try {
5                    lock.wait();
6               } catch (InterruptedException e) {
7                    System.out.println(e.getMessage());
8               }
9           }
10          blocked = true;
11          if (lock != original_lock) {
12              /* thread exits from Object.wait */
13              throw new InterruptedException();
14          } else {
15              Thread.currentThread().interrupt();
16          }
17      }
18  }
```

Figure 5.7: Extended version of the `block` method

5.5.4 Thread.sleep and Thread.yield

Calling `sleep` or `yield` may result in a thread change. Again, if a thread change was discovered during the capture phase, it will be contained in the thread replay schedule and a transfer will be instrumented. If no thread change was taken, calling `sleep` or `yield` implies no thread switch takes place during replay as all other threads are blocked. These calls therefore do not need to be removed.

5.5.5 Thread.join

`Thread.join` comes in three forms like `Object.wait`. Consider the case of `t.join` where the current thread waits until thread `t` dies. The normal case is that the current thread is suspended, and a thread change occurs. This thread change is contained in the thread replay schedule. A transfer is instrumented into the program. As a result, the call to `join` must be removed during instrumentation. If `t` is not running, e.g., because it has not yet been started, no thread change occurs.

When `t` dies or the optional timeout has been reached, control is passed back using another transfer. If `t` is interrupted as another thread calls `t.interrupt`, then the situation is as described in Section 5.5.3.

5.6 Summary

The proposed replay mechanism assigns a lock object to each thread. A thread is blocked by calling `wait` within a loop on this lock object and unblocked by executing a corresponding `notifyAll`. A binary semaphore is used to prevent deadlocks in the control transfer method due to interception by the JVM scheduler.

To prevent unwanted execution of new threads, they are immediately blocked. The transfer method warrants that the additional instrumented block is ignored if the thread change is intended. To avoid that ending threads remain blocked, they are unblocked directly before their final thread change. If `o` is an object, calls to `o.wait` are replaced by two replayer calls that assign `o` as temporarily lock for the transfer method. As the transfer method itself performs the necessary actions, the corresponding `notify` and `notifyAll` instructions can be removed.

Invocations of `Thread.interrupt` are replaced with a method that sets a flag associated with the unblock method in the replayer. When the thread is resumed, the interruption is replayed. As explicit thread changes are always instrumented, calls to `Thread.join` need to be removed. To avoid side-effects, calls to `Thread.sleep` and `yield` can be left in the original program. They cannot imply a thread change during replay as all other threads are blocked due to the replay invariant.

Chapter 6

Instrumented Bytecode Transformations

In this chapter, the replay concept proposed in Chapter 5 is now mapped to nine instrumentation rules, described in the following sections. The instrumentation rules are implemented in the **jreplay** application described in Appendix A using the instrumentation facility presented in the next chapter.

Instrumentation is performed on Java bytecode, not on abstract bytecode. It was briefly considered to perform instrumentation at the abstract bytecode level and then transforming the program back to Java bytecode before writing the class files to disk. While a conversion to and from abstract bytecode would result in a semantically equal program, such a transformation would have too much of an impact on the structure of a program.

All necessary type data, e.g., class names, is looked up in the constant pool during instrumentation. The system therefore works independent of the Java reflection API.

6.1 Rule 1: Initialization of the replayer

At the begin of the `main` method of a class, an `invokestatic` call to `replay.init` is inserted to initialize the replayer when the instrumented program is executed. Figure 6.1 shows the changes applied at bytecode level. The function takes the name of the instrumented class as an argument to derive the name of the schedule file. For this purpose, the class name is looked up in the constant pool during instrumentation and an `ldc` instruction is added that pushes this value to the operand stack. The constant pool index `#1` depends on the instrumented class file and will be different in real examples.

Original Bytecode				Instrumented Bytecode
0	...	⇒	0	**ldc #1 < String *classname* >**
			2	**invokestatic #2 <void replay.init (String)>**
			5	...

Figure 6.1: Bytecode transformations of `main` method to initialize the replayer

As `init` is called as first instruction within the `main` method of an application, no other thread except the main thread is running. Method `init` registers this main thread by

adding it to the `taskPool` data structure. It also assigns zero as unique identifier of the thread. Finally, the thread replay schedule is read from the same execution trace file that was used to instrument the class files. The schedule is kept in memory until the application to be replayed terminates.

6.2 Rule 2: Instrumenting explicit thread changes

Original Bytecode				Instrumented Bytecode
i	...	$\Rightarrow$	i	**ldc #1 <String *classname* >**
			$i+2$	**push *method_index***
			$i+3$	**push *bytecode_loc***
			$i+4$	**invokestatic #2 < void replay.sync(String,int,int) >**
			$i+7$	...

Figure 6.2: Bytecode transformation to force an explicit thread switch

Figure 6.2 shows how a call to the `replay.sync` method is inserted at all locations whenever a thread change is necessary according to the thread replay schedule. In addition to class name and method index, the absolute address of the uninstrumented bytecode is given as a parameter.

The current thread has to be temporarily suspended. The next thread that should be unblocked is obtained from the thread schedule by examining the thread schedule and resolving the given index in the `taskPool` data structure. Figure 6.3 shows the pseudo-code of `sync`.

```
1   public void sync(String classname, int method_index, int loc) {
2       Thread current = Thread.currentThread();
3       next = getNextInSchedule();
4
5       if (! classname.equals(next.classname)
6           || (method_index != next.method_index)
7           || (loc != next.location)) {
8               /* skip wrong location */
9               return;
10      }
11
12      /* now: correct absolute location */
13      if (next.moreIterations()) {
14          /* more iterations of current location */
15          next.decreaseIterationCounter();
16          return;
17      }
18
19      /* now: correct absolute location and iteration step */
20      dropCondition();
21
22      if (noMoreConditions() {
23          /* schedule exhausted */
24          unblockAllThreads()
25      }
26
27      transfer(next);
28  }
```

Figure 6.3: Pseudo-code of the `replay.sync` transfer method

When the replay schedule has been processed, the current implementation resumes normal execution of the program by unblocking all threads. Another possibility would be to call a user function or abort replay.

6.3 Rule 3: Registering new thread objects

The order in which new threads are registered with the replayer is important, as their index in the thread pool is linked with the thread index in the thread switch rules in the file describing the replay schedule. New thread objects are therefore registered with the replayer immediately after their instance initializer was executed.

Creation of new object instances is done in two steps for all objects, independently of whether they are to become a thread or not. On bytecode level, object instances are created using the `new` instruction, followed by a call to their instance initialization method `<init>` using the `invokespecial` opcode [28, Sec. 7.8]. The `new` instruction does not completely create a new instance unless the instance initialiser method has been invoked. New threads therefore cannot be registered with the replayer before the `invokespecial` call. As `invokespecial` consumes one object reference from the stack, the reference to the new object is duplicated beforehand on the stack.

The `new` instruction takes an index into the constant pool as bytecode argument to determine the type of the new object. Analyzing this constant pool entry during instrumentation allows to detect creation of new threads. If the newly created object does not implement the `Runnable` interface, then it will never become a thread object and instrumentation is not necessary.

Figure 6.4 shows how a call to `replay.registerThread` is inserted directly after the call to the instance initializer. The thread object to be registered is passed as an argument to that function. To obtain the argument for the `invokestatic` method call, a `dup` command is inserted at location *k* directly before the call to the instance initialization method `<init>`. This reference is consumed by the `invokespecial` instruction, but the original reference obtained by `dup` at location *i+3* is still on the stack and will be used as argument to the `invokestatic` function call.

	Original Bytecode			**Instrumented Bytecode**
	…			…
i	new #1<*object type*>		*i*	new #1 <*object type*>
i+3	dup		*i+3*	dup
	…	⇒		…
	…			…
k	invokespecial #2 <init>		*k*	**dup**
k+3	…		*k+1*	invokespecial #2 <init>
			k+4	**invokestatic #3 <registerThread>**
			k+7	…

Figure 6.4: Bytecode transformations to register a new thread object

At the time of instrumentation, the class name of the object to be created can be obtained from the constant pool when the instrumentation facility runs across a `new` instruction. As only thread objects need to be registered with the replayer, the following algorithm is used to analyze the class name:

1. Within the `java` name space hierarchy, only the `java.lang.Runnable` interface and the `java.lang.Thread` class are potential threads. If the class name directly equals one of these two names, the `new` instruction needs to be instrumented.
2. Some standardized classes, e.g., classes from the `javax.swing` package, cannot be instrumented, as the class files contain native calls or are hidden within the JVM implementation. If the class name can be found in a given list of such classes, the `new` instruction does not need to be instrumented.
3. If the class was already loaded once at an earlier stage of instrumentation, a cached entry exists in the **JNuke** class pool. The entry can be used to obtain all implemented interfaces and the inheritance chain of extended super classes. The `new` instruction needs to be instrumented if any implemented interface or super class implements the `Runnable` interface.
4. If no entry in the class pool is available for the mentioned class, the instrumentation rule tries to locate and load the class from disk. The class descriptor is then analyzed as in step 3. If the class file can not be located, the instrumentation facility tries the alternative directories in the class path. If loading the class file still fails, a warning message is printed out. It is assumed that the class does not implement the `Runnable` interface. In the case that a class file may not be located but implements the `Runnable` interface, deterministic replay of the application will fail.

6.4 Rule 4: Retaining the replay invariant for new threads

Figure 6.5 shows how a call to `replay.blockThis` is inserted at the beginning of every `run` method to prevent an unwanted thread switch once a thread is started. When the JVM thread scheduler unintentionally switches to the newly created thread, the thread is marked as blocked and the thread will immediately suspend itself.

If the new thread is not scheduled for execution, `blockThis` is not called. If at a later point, an explicit thread change to the thread is performed using an instrumented transfer, the `unblock` operation during the transfer will unset the `blocked` flag of the thread. When the `blockThis` method is invoked, the call has therefore no effect.

Original Bytecode			Instrumented Bytecode
0 ...	⇒	0	**invokestatic #1 <void replay.blockThis() >**
		3	...

Figure 6.5: Bytecode transformations of `run` methods to retain the replay invariant for newly created threads

6.5 Rule 5: Terminating threads

The `run` method is required by the `Runnable` interface to have the return type `void`, therefore all `vreturn` instructions are potential method exit points.

A call to `replay.sync` will be inserted at the exit point where the thread has left its `run` method as termination always implies a thread change. During replay, this enforced thread change makes it impossible for a thread to die. The terminating thread must be unblocked before control is transferred to the next thread. This can be achieved by inserting a call to `replay.terminate` directly before every `vreturn` instruction in the `run` method, as shown in Figure 6.6.

Original Bytecode			Instrumented Bytecode	
i	vreturn	⇒	*i*	**invokestatic #1 <void replay.terminate() >**
			i+3	ldc # < String *classname* >
			i+6	push *method_index* >
			i+7	push *bytecode_loc*>
			i+8	invokestatic < void replay.sync (String, int, int) >
			i+11	vreturn

Figure 6.6: Bytecode transformation to release a terminating thread

6.6 Rule 6: Object.wait

Figure 6.7 shows how the `replay.sync` instrumented by transformation rule 2 is complemented with two calls to `replay.setLock` and `replay.resetLock` as described in Section 5.5.1. The actual call to `wait` and potential parameters are removed. The argument to `setLock` is the argument of the original call to `invokevirtual` instruction.

Method `wait` can be called with a time limit. If no other thread is ready to run in the original program, execution will resume after the time has elapsed. In this case, no thread change will occur and rule 2 will not instrument a thread change. During replay, the call to `wait` is removed. The instrumented program passes through the `setLock` and `resetLock` instructions without delay. The side effect could be prevented by inserting a call to `sleep`.

Original Bytecode			Instrumented Bytecode	
i	invokevirtual #1 <wait>	⇒	*i*	**invokestatic <replay.setLock(Object) >**
i+3	. . .		*i+3*	ldc # < String *classname* >
			i+6	push < *method_index* >
			i+7	push < *bytecode_loc*>
			i+8	invokestatic <replay.sync(String, int, int) >
			i+11	**invokestatic <replay.resetLock() >**
			i+14	. . .

Figure 6.7: Bytecode transformation to instrument `Object.wait`

6.7 Rule 7: Object.notify and notifyAll

As with `wait`, the call is removed. If an implicit thread change took place during the capture, a call to `sync` is inserted by rule 2.

`Object.notifyAll` acts like `Object.notify`, except that all threads waiting on the monitor of this object are woken up. During replay, the thread schedule is known and `notifyAll` is treated exactly like `notify`.

	Original Bytecode			Instrumented Bytecode
i	invokevirtual #1 <notify>	⇒	*i*	ldc #1 <String *classname*>
i+3	…		*i+1*	push *method_index*
			i+4	push *bytecode_loc*
			i+5	invokestatic #2 <void sync>
			i+6	…

Figure 6.8: Bytecode transformation to instrument `Object.notify`

6.8 Rule 8: Thread.interrupt

Every call to `Thread.interrupt` is replaced by a call to `replay.interruptSelf` as shown in Figure 6.9, that sets a flag for the specified thread in the replayer.

	Original Bytecode			Instrumented Bytecode
	…			…
i	invokevirtual #1 <interrupt>	⇒	*i*	**invokestatic #2 < interruptSelf(Thread) >**
	…			…

Figure 6.9: Bytecode transformation to instrument `Thread.interrupt`

6.9 Rule 9: Thread.join

Figure 6.10 presents how calls to `Thread.join` are removed. If a thread change occurred, rule 2 instruments an explicit thread switch by inserting a call to `sync`. If arguments were supplied to `join`, they are removed from the operand stack by inserting appropriate `pop` and `pop2` instructions.

	Original Bytecode			Instrumented Bytecode
	...			...
i	invokestatic #1 <Thread.join>	⇒	*i*	ldc #2 <String *classname*>
i+3	...		*i+2*	push *method_index*
			i+5	push *bytecode_loc*
			i+6	invokestatic #3 <void sync>
			i+9	...

Figure 6.10: Bytecode transformation to instrument `Thread.join`

6.10 Summary

Figure 6.11 summarizes the instrumentation rules presented in this chapter. Five rules insert calls to replayer methods. Calls to `Thread.join`, `Object.notify` and `notifyAll` and their arguments are removed. `Thread.interrupt` and `Object.wait` are replaced with calls to the replayer.

Rule	Location	Instrumentation
1	Begin of `main` method	Insert call to `replay.init`
2	Explicit thread change locations	Insert call to `replay.sync`
3	Creation of new thread objects	Insert call to `replay.registerThread`
4	Begin of `run` method	Insert call to `replay.blockThis`
5	`vreturn` instructions in `run` methods	Invoke `replay.terminate`
6	`Object.wait`	Remove call and parameters, temporarily overwrite lock
7	`Object.notify` and `notifyAll`	Remove call
8	`Thread.interrupt`	Replace by call to `replay.interruptSelf`
9	`Thread.join`	Remove calls and parameters

Figure 6.11: Summary of instrumented bytecode transformations

Chapter 7

The Instrumentation Facility

7.1 Design of the JNuke instrumentation facility

As part of this thesis, the **JNuke** framework has been extended with an instrumentation facility. Instrumenting Java bytecode is the task of patching the code in a compiled class file. The input file is read into memory using the class loader. The compiled code in the byte stream is parsed, and control flow is intercepted at various points by inserting and/or removing bytecode operations at distinct positions, as required.

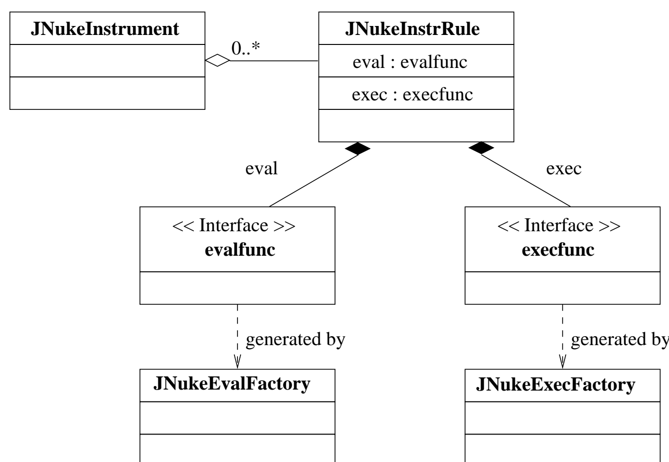


Figure 7.1: Layout of the **JNuke** instrumentation facility

Figure 7.1 shows the objects of the new instrumentation facility. A `JNukeInstrument` object acts as a container for one or more instrumentation rules. Each instrumentation rule is an object `JNukeInstrRule` that defines where and how the bytecode of a method should be instrumented. It stores function pointers to two methods `eval` and `exec` for this purpose. The functions can be obtained from two libraries `JNukeEvalFactory` and `JNukeExecFactory`. Single rules can easily be enabled or disabled, depending on whether they are added to `JNukeInstrument` or not. The class file writer described in the next chapter may be used to write the instrumented class file back to disk.

7.1.1 Instrumentation model

`JNukeInstrument`, described in Appendix E.10, acts as a container of instrumentation rules by providing methods to add and remove instrumentation rules. As the set of instrumentation rules is ordered, there is an implicit deterministic order of how the changes are applied to the bytecode.

The object implements a per-method instrumentation facility operating on what is henceforth called a ”working method”. All interaction with the instrumentation facility is done using the `JNukeInstrument` object. Iterators are used to process each rule with every method of a class file. Figure 7.2 shows the pseudo-code for instrumentation.

```
1   void instrument(class c) {
2           for each method m in c {
3                   for each rule r in rules {
4                           for each bytecode bc {
5                                   context = (c,m,bc,rules,...);
6                                   if (r.eval(context)) {
7                                           r.exec(context);
8                                   }
9                           }
10                  }
11          }
12  }
```

Figure 7.2: Pseudo-code for instrumentation of a class in memory

During instrumentation, the object maintains an *instrumentation context* that is passed to the rules. Among other information, this context includes the current set of rules, the current class and method being processed, the current position in the bytecode as well as the constant pool of the class file. A rule is then free to use this information to perform its work.

7.1.2 Layout of instrumentation rules

A `JNukeInstrRule` object described in Appendix E.9 embodies a single instrumentation rule. A rule consists of a globally unique descriptive name, and two functions `exec` and `eval`. The first function applies the rule, i.e., patches the bytecode at the current instrumentation context. The second function decides whether instrumentation is necessary at the current bytecode location. The `exec` function specifying how the bytecode should be modified is executed only if the `eval` function call evaluates to true.

This two-part design of an instrumentation rule allows to reuse individual functions. To name an example, it is possible to write an `atEndOfMethod` evaluation function that returns true if the instrumentation context is at the last instruction of the method. This location evaluation function can then be used with any `exec` function.

Instrumentation does not always need to be deterministic. To implement randomized scheduling or automated testing, generic probabilistic location evaluation functions could rely on an internal or external random number generator. Rules based on such functions would allow to instrument bytecode with a given probability.

As the instrumentation setup uses a safe vector iterator, it is possible to alter the set of rules at runtime, although this feature is yet untested and therefore not recommended. Both `exec` and `eval` function could alter the set of rules. Possible applications of this scenario would be “one-shot rules“, i.e., rules that remove themselves from the instrumentation facility after they have been executed. Removing instrumentation rules at runtime may have a positive impact on instrumentation performance.

By using a plug-in functionality, it may also be possible to reuse the instrumentation facility for different kinds of bytecode. As described in Section 2.2, **JNuke** is capable of handling alternative kinds of bytecode. A different set of rules could be developed for register-based and/or abstract bytecode.

All currently implemented `exec` and `eval` functions were combined into two libraries (see Appendices E.11 and E.12). These libraries may be used and extended by future applications.

7.2 A two phase protocol for instrumenting bytecode

One of the main difficulties of instrumenting Java bytecode is the maintenance of instruction offsets. By inserting and/or removing bytecode instructions, the offsets of the remaining instructions in the bytecode stream usually change. Figure 7.3a shows the bytecode of the `closedloop` example program, whose source code is given in Figure D.1. Arguments to branch instructions must be updated after changes to the bytecode are made. For example, the target 8 of the `goto` command at offset 2 in the uninstrumented program needs to be changed to 21, as shown in Figure 7.3b. The changed instructions are highlighted in a bold font.

	a) Original bytecode		**b) Instrumented bytecode**
0	iconst_0	0	**ldc #16 <String ”closedloop”>**
1	istore_1	2	**invokestatic #26 <void init(String)>**
2	goto 8	5	iconst_0
5	iinc 1 1	6	istore_1
8	iload_1	7	goto **21**
9	ldc #2 <Integer 1000000000>	10	iinc 1 1
11	if_icmplt 5	13	**ldc #16 <String ”closedloop”>**
14	vreturn	15	**iconst_1**
		16	**bipush 8**
		18	**invokestatic #22 <void sync(String, int, int)>**
		21	iload_1
		22	ldc #2 <Integer 1000000000>
		24	if_icmplt **10**
		27	vreturn

Figure 7.3: Instrumenting the `closedloop` example program

When bytecode is instrumented at multiple positions, complexity of the necessary bytecode updates increases. A possible solution would be to perform the necessary changes after every insertion or deletion of a bytecode instruction. This approach has a severe drawback: If a rule inserts multiple instructions at the same time, committing the necessary changes to the bytecode stream might shift the insertion point for the second instruction. It would be necessary to determine the new insertion location for the second instruction. Updating the bytecode after every change is also very inefficient, as previously applied changes are frequently overwritten.

The implemented solution therefore follows a different approach. The code of every method is processed in two steps. In the first phase, the changes to be applied to the bytecode are collected from the registered rules and stored for later use. It is important to understand that no changes to the bytecode are made in the first phase. In the second phase, the previously registered changes are applied to the bytecode in a single pass. The two phases are now discussed in detail.

7.2.1 Phase 1: Registration of intended modifications

During the first phase, rules may register their intention to modify the bytecode of a single method. The instrumentation facility provides two methods `insertByteCode` and `removeByteCode` for this purpose. By calling these functions the instructions to be inserted are stored in an “insertion wait list“ together with a bytecode position specifying their insertion location. The bytecode offset of instructions to be removed is stored in a corresponding “deletion wait list“.

Figure 7.4 shows the insertion wait-list for the previous example. Note that subsequent instructions at the same location must be inserted in reverse order. As no instructions are removed, the deletion wait-list is empty in this example.

Ofs	Instruction
0	invokestatic #26 <void init(String)>
0	ldc #16 <String ”closedloop”>
8	invokestatic #22 <void sync(String int, int)>
8	bipush_8
8	iconst_1
8	ldc #16 <String ”closedloop”>

Figure 7.4: Insertion wait list for the `closedloop` example program

7.2.2 Phase 2: Applying the previously registered modifications

In the second phase, the previously registered changes are applied to the bytecode. A bytecode address transition map is kept in memory for the current method. This map stores all original locations and their corresponding bytecode position after the changes have been applied to the bytecode. Figure 7.5 shows the transition map for the example program in Figure 7.3.

orig.		instr.	orig.		instr.	orig.		instr.
0	↔	5	5	↔	10	11	↔	24
1	↔	6	8	↔	21	14	↔	27
2	↔	7	9	↔	22			

Figure 7.5: Transition map for the `closedloop` example program

The map is bidirectional, i.e., it allows queries to be performed in both directions. The implementation of this map is based on the existing `JNukeOffsetTrans` object that is used within the **JNuke** framework to store changes of bytecode offsets while inlining `jsr` instructions.

The second phase consists of six steps:

1. Changes registered in the patch map are committed by calling the method `JNukeBCPatchMap_commit`. This will produce the final bidirectional map allowing original bytecode addresses to be translated to the instrumented method and vice versa.
2. The deletion wait list is searched for the instruction with the highest offset. This instruction is both removed from the bytecode stream and the deletion wait list. The step is repeated until the deletion wait list is empty.

3. The list of instructions to be inserted is searched for the lowest address. If there are multiple instructions with the same address, then the original order how they were added to the insertion wait list will be retained. These instructions are then inserted into the bytecode vector of the method. Processed entries are removed from the insertion wait-list.
4. If there are more instructions to be inserted, step 3 is repeated as often as necessary.
5. The bytecode stream is parsed for branch instructions. The arguments of these instructions are updated using the map derived in step 1. The padding bytes of instructions that require word-alignment, e.g., `tableswitch` and `lookupswitch`, are adjusted at the same time.
6. The exception handler table of the method is traversed using an iterator. All intervals and handler addresses are updated. Finally, the transition map is cleared.

7.3 Selected problems of bytecode instrumentation

While modifying bytecode as described in phase 2 works with most examples, some specific problem situations may arise. In the following paragraphs, selected problems are presented together with the corresponding solutions.

7.3.1 Large offsets in absolute branches

Figure 7.6a shows the bytecode of a method. By inserting instructions between the `goto` instruction at offset 2 and the absolute branch target at offset 32768, the branch target of the `goto` instruction needs to be updated. The `goto` instruction uses only two bytes to encode the relative offset of the branch target. If too many instructions are inserted, an overflow can occur.

a)			b)	
0	iconst_0		0	iconst_0
1	istore_1		1	istore_1
2	goto 32768	⇒	2	**goto_w 32770**
...	...		...	...
32768	iload_1		**32770**	iload_1

Figure 7.6: Example bytecode with a large absolute branch offset and instrumented substitute

Figure 7.6b explains how an absolute branch instruction can be expanded into its wide instruction format. In this example, the `goto` instruction is substituted with a `goto_w` command that takes four bytes to encode the relative offset to the branch target. As a direct consequence, instructions following the absolute branch will be shifted by two bytes. The new branch target is `32770` plus the size of the inserted instructions.

A similar substitute can be applied for all absolute branch instructions, i.e., a `jsr_w` opcode is used in case of a `jsr` instruction. As two bytes are inserted, steps 5 and 6 of the algorithm described in Section 7.2.2 need to be repeated for each overflow. This is inefficient, but should not occur often.

7.3.2 Large offsets in relative branches

Consider the Java bytecode of the method given in Figure 7.7a. By inserting some instructions in the area between the `ifne` and the `iload_1` instruction, the jump target of the conditional branch at offset 3 is shifted and must be updated. When inserting an unfortunate number of instructions, the distance to the jump target at `32772` will eventually increase to a point where it is no longer addressable by the `ifne` instruction. In the previous section, we faced a similar problem. The previous solution was to expand the narrow bytecode instruction into its wide counterpart. Unfortunately, there are no wide versions to `if` instructions in the bytecode instruction set [28, Chapter 6].

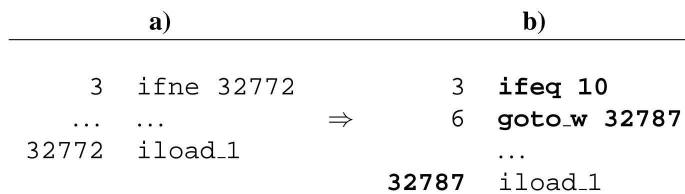


Figure 7.7: Example bytecode with a large relative branch offset

A different solution that is implemented in this instrumentation facility is shown in Figure 7.7b. It consists of two steps.

First, the original conditional branch instruction is negated. In our example, the `ifne` instruction at offset 3 in the original program is replaced with an `ifeq` statement. The target offset of the conditional branch instruction is set to a relative value of seven. This offset points to the instruction following the `goto_w` instruction.

Second, as wide variants for unconditional branch instructions exist, a `goto_w` instruction is inserted directly after the conditional branch statement. The argument of the `goto_w` instruction is the updated target offset of the original `ifne` command. The program resulting from this transformation is semantically equal to the original one. Again, steps 5 and 6 of the presented algorithm need to be re-executed.

7.3.3 Operand stack depth

Every method has its own operand stack. The maximum depth of this operand stack is specified in the `Code` attribute of each method and may not be exceeded at any point during execution of the method. Otherwise, the class is rejected by the Java bytecode verifier. During instrumentation, instructions that push arguments to the stack are inserted into the program, e.g., to pass arguments to an instrumented function call.

Due to time constraints, only a simple heuristic was implemented. If the maximal stack size property of a method where instructions are inserted lies below a *critical stack size*, the *MaxStack* property of the method is raised as a precaution. Raising the allowed stack size limit is trivial to implement and has little influence on the performance. The stack size limit property is not changed for methods where no instructions are inserted. A setting of `10` for the critical stack size has worked very well in all performed experiments. It is important to note that this heuristic may fail.

A more advanced solution would be to repeatedly raise *MaxStack* every time a push instruction is inserted. This approach gives an upper bound on the number of items on the operand stack that would be inefficient, but always correct.

The best solution would be to determine the stack height at each instruction by static analysis. Within the **JNuke** framework, the `JNukeSimpleStack` object already implements a similar algorithm for abstract bytecode.

7.3.4 Very large methods

The code size of a method is currently limited to 65535 bytes due to static constraints of a Java class file [28, 4.8.1]. Although such large methods are very rare in practice, this restriction can be violated by adding instructions to the code attribute of a method. Unfortunately, there is no solution conforming to the virtual machine specification other than rewriting the program to be instrumented, i.e., by splitting the large method into multiple parts. If the allowed size of a code section is exceeded during instrumentation, a warning message is printed to inform the user that the generated class file violates the aforementioned constraint.

7.4 Summary

A Java bytecode instrumentation facility was added to the **JNuke** framework. Rules are registered with the facility that decide where and how bytecode should be transformed. Instrumentation is performed in two steps. First, each rule can request insertion or deletion of bytecode instructions. Second, a bidirectional address transition map is built from the collected requests and the changes are applied to the bytecode.

Inserting instructions may have the side-effect that certain branch targets are no longer addressable, which requires additional modifications to the bytecode. Implemented solutions are expanding narrow instructions to their wide counterparts and complementing bytecode with wide absolute branches.

Due to time constraints, only a simple heuristic was implemented to detect the maximal size of the operand stack after instrumentation. This heuristic should be replaced with a sound algorithm as used in Java bytecode verifiers.

Chapter 8

The Class File Writer

Java stores its code in standardized class files [28, Ch.4], class files are built hierarchically from various parts. Symbolic information and constants are stored in a constant pool. Complete description of the class or interface fields and methods reside in `field_info` and `method_info` arrays, respectively.

This chapter describes how the **JNuke** framework was extended with a possibility to write new Java class files to disk. The class writer takes an existing memory representation of a class descriptor object as input. The generated class files have to follow the structural constraints described in the Java virtual machine specification [28, Ch. 4], otherwise they will be rejected by the verifier when executed.

Section 8.1 presents new classes added to the framework. Section 8.2 deals with some details regarding modifications applied to the existing framework in order to obtain an optimal class file writer. Finally, Section 6.3 shows the usage of the class writer object in an application.

Class loading in JNuke

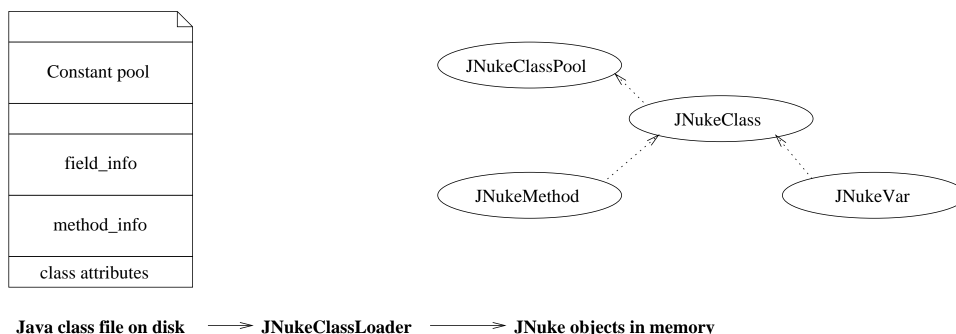


Figure 8.1: Loading Java class files in **JNuke**

Figure 8.1 schematically shows how a class file is loaded within the framework. **JNuke** uses its own objects for internal representations of classes, methods and fields. These objects are called `JNukeClass`, `JNukeMethod`, and `JNukeVar`. A global constant pool shared by all classes is maintained by the `JNukeClassPool` object. The `JNukeClassLoader` can be used to read a class file from disk and create internal representation of its contents.

8.1 Class writing

A `JNukeClassWriter` drives the class writing process. It delegates the tasks of creating the more complex data structures to the new objects shown in Figure 8.2. These objects are discussed in more detail in the following sections. The `JNukeConstantPool` object is responsible for maintenance of the constant pool. The `JNukeAttributeBuilder` object is used to build new attributes from **JNuke** descriptor objects. `JNukeAttribute` objects are used to prepare the writing of class attributes. Instances of `JNukeAttribute` are used by the `JNukeMFInfoFactory` to create representations of the `method_info` and `field_info` areas of class files.

The values for the remaining fields in a class file are easy to fill in. After the memory images are collected from the other objects, appropriate methods are used to write everything to disk.

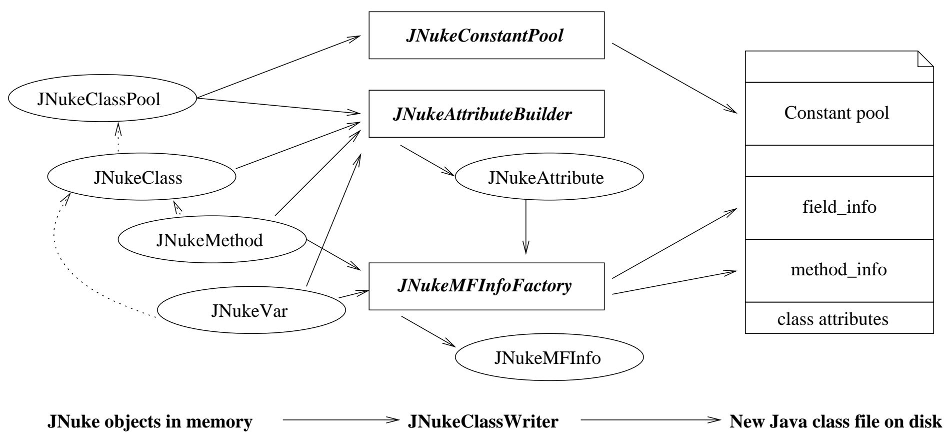


Figure 8.2: Objects involved in writing new Java class files

8.1.1 Constant pool

To simplify the maintenance of a class file constant pool, a `JNukeConstantPool` class was devised (see Appendix E.1. The constant pool object is an array of variable size. A tag is associated with each entry to specify its type [28, Table 4.3].

The component was designed to easily allow the addition of new items. Duplicate entries in the constant pool are avoided by scanning the constant pool before a new entry is added. If an identical item is already contained in the constant pool, no modifications are made. For the purpose of writing a class file, a method was implemented that returns the exact representation of an entry as required in a Java class file constant pool.

Many Java bytecode instructions, such as `ldc` or `invokevirtual`, refer to the constant pool using indices. When creating the constant pool from scratch or by modifying an existing constant pool these indices may change. To avoid having to update all the indices in the byte code, the current class writer is capable of reusing the constant pool obtained by the class loader.

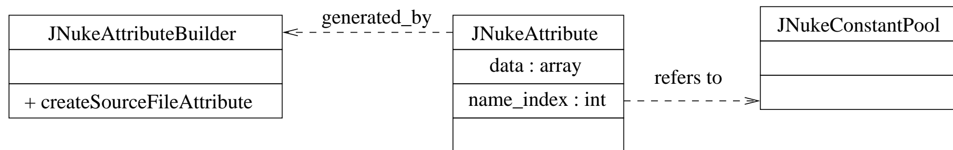


Figure 8.3: Attribute objects

8.1.2 Attributes

The attribute object shown in Figure 8.3 is essentially a data buffer of variable size. Each attribute has an index into the constant pool that refers to its name.

Initially, attributes were implemented using simple C structs. This was a poor choice because of the following reasons:

- Structs are suited for compound data structures of fixed size whose element offsets never change. In contrast, real attributes have variable length. Adding new items to the fields within an attribute was difficult to implement.
- Individual elements of an inhomogeneous struct are word-aligned by the compiler, introducing memory gaps that makes it impossible to write a struct to disk at once. A workaround for this problem would have been to use *packed structs*, but the way a struct is attributed as packed depends on the compiler. The required conditional definitions unnecessarily increases the complexity of the source code. Some compiler may not support packed structs at all.

Appendix E.2 describes a `JNukeAttribute` object that uses a buffer of variable length. Generic methods to append data to the buffer and to overwrite existing values are available that automatically keep track of the actual buffer size and perform boundary checks. The attribute object also provides methods to set and query the *NameIndex* property.

As the *Code* attribute of a method may contain further attributes, a possibility to nest attributes was added.

8.1.3 Building attributes from templates

A recurring task during class writing is to take the internal representation of a descriptor object and turn it into a corresponding attribute. A generic attribute builder has been devised for this purpose. For example, this builder object can extract the data contained in a particular `JNukeMethod` descriptor object and use it to create a new *Code* attribute. The `JNukeAttributeBuilder` object described in Appendix E.3 offers methods to create *SourceFile*, *InnerClasses*, *LineNumberTable*, *LocalVariableTable*, *Deprecated*, *Exceptions*, *Code*, and *ConstantValue* attributes.

Every class writer instance has an attribute builder object associated with it. This attribute builder uses the same constant pool as the class writer.

8.1.4 Methods and fields

The `method_info` and `field_info` data structures [28, Sec. 4.5/6] share similar fields. Representation of the two structures has therefore been combined into a single `JNukeMFInfo` object, described in Appendix E.4.

The JNukeMFInfo object has methods to set and query the integer properties *AccessFlags*, *NameIndex*, and *DescriptorIndex*. Apart from that, a method_info or field_info data structure consists of zero or more attributes. The JNukeMFInfo object therefore acts as a vector for a variable number of JNukeAttribute objects. The number of attributes in the attribute table can be obtained by counting the elements of the vector.

8.1.5 Method and field information factory

A single JNukeMFInfo object contains either information about a particular method or a particular field. To create new object instances with data based on existing descriptor objects, a JNukeMFInfoFactory factory object was devised. As the attribute builder may create new instances of the JNukeAttribute object, the JNukeMFInfoFactory can use an existing JNukeMethod or JNukeVar descriptor object and create a new instance of a JNukeMFInfo object from it.

Generating Method information

To generate a new JNukeMFInfo object representing the method_info data structure described in [28, Sec. 4.6], a JNukeMethod descriptor object is passed to the method JNukeMFInfoFactory_createMethodInfo. This builder method creates a new JNukeMFInfo object and sets the *access flags*, *name index*, and *descriptor index* properties according to the current settings in the method descriptor.

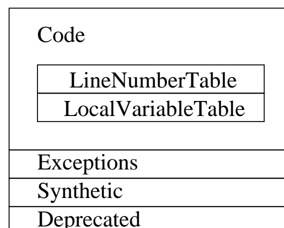


Figure 8.4: Attributes in a JNukeMFInfo object for a method_info item

If the method is neither abstract nor native, a Code attribute [28, Sec. 4.7.3] is created by the attribute builder and added to the JNukeMFInfo container. The attribute builder object will automatically add LineNumberTable and/or LocalVariableTable subattributes, depending on the JNukeMethod descriptor template.

Exceptions [28, Sec. 4.7.4], *Synthetic* [28, Sec. 4.7.6], and *Deprecated* [28, Sec. 4.7.10] attributes are created as required, as shown in Figure 8.4.

Generating field information

Generation of field information is very similar to the creation of method information described in the previous paragraph. First, a JNukeMFInfo object instance is created and its *access flags*, *name index*, and *descriptor index* properties are set. Second, if the final bit of the field descriptor object is set, the attribute builder object is used to create a ConstantValue attribute [28, Sec. 4.7.2]. Last but not least, if the variable descriptor is tagged as *synthetic* and/or *deprecated*, the respective attributes are created and added to the JNukeMFInfo object.

8.2 Extensions to the JNuke framework

In order to optimize the resulting class files, some extensions to the **JNuke** framework had to be made. Some of the bigger changes consisted in adding support for *Deprecated* attributes to the class loader, modifying descriptor objects and extending the floating point wrapper objects to handle the special values infinity and NaN.

8.2.1 Support for missing attributes in the class loader

Java offers a consistent way based on *javadoc* [38] comments to prevent the usage of fields, methods, and/or classes that should no longer be used. A compiler will warn programmers that they are using superseded items when it encounters a corresponding *Deprecated* attribute in a class file [28, Sec. 4.7.10].

The corresponding descriptor objects were extended with a `deprecated` bit and appropriate methods for setting and querying the flag. The parser in the class loader was extended to handle *Deprecated* attributes, to avoid the loss of this information when instrumenting class files. To achieve a complete coverage of all access flags defined in the Java virtual machine specification [28], the missing flags were added to descriptor objects. Finally, the `JNukeMethod` descriptor object was extended with appropriate methods to modify and query the list of exceptions a method may throw.

8.2.2 Modifications to Floating Point objects

One of the design goals of the **JNuke** framework is that it should run on as many platforms as possible. Because the size of floating point variables differs between computer architectures, the framework uses uniform wrapper objects that try to hide the platform-dependent issues by providing a uniform programming interface. In Java, the treatment of the special values *positive infinity*, *negative infinity* and *NaN* is different from the IEEE 754-based standard implemented in most computer hardware. Support for these special values was missing in the framework.

Support for NaN values

An exception is thrown by the central processing unit whenever an integer value is divided by zero, as the result is mathematically not defined. The IEEE754 standard for binary floating point arithmetic demands that a status flag in the floating point register is set when a similar situation occurs with floating point values [21, Sec. 7, p.14]. The operating system provides subroutines to query, set, or clear these flags. Instead of setting a condition flag, the system may raise a floating point exception[1].

A special symbolic value “not a number“ (NaN) is assigned to floating point variables whenever the result of a mathematical operation is no longer defined [21, Section 2, p.8]. An example for such a floating point operation is the division of the floating point value 0.0 by itself [21, 7.1(4), p.14].

To determine whether the value of a floating point variable is no longer defined, most C language environments provide a function `isnan`, that returns a boolean value. On other systems, two versions of the function, called `isnand` and `isnanf` are offered instead. The first is intended for variables of type `double`, the latter for `floats`. Beside different

[1] `SIGFPE` signal on Unix [2, Sec. 7.2.1, p.203]

naming of some mathematical functions, the names of the libraries and header include files containing the functions depend on the platform, too[2].

Fortunately, the floating point standard allows a trivial way to detect whether a value equals to NaN [8]. By definition, a comparison of NaN with another value always fails [21, 5.7 p.12]. This implies that comparison of NaN with itself fails too. Instead of using the `isnan` function of the underlying operating system libraries, a simple comparison is used to implement two functions `JNukeFloat_isNaN` and `JNukeDouble_isNaN`.

The *DEC Alpha* architecture differs in the treatment of NaN floating point values, as a `SIGFPE` signal is thrown whenever a NaN value is assigned to a floating point variable. If the signal is not caught, applications are terminated by the operating system whenever an application is using NaN floating point values. A POSIX signal handler was implemented to circumvent this problem.

Support for infinite values

Two distinct special values represent positive and negative infinity in Java class files, unlike most floating point unit implementations in computers that allow arbitrary values within certain ranges.

A similar property as for NaN values is used to detect infinite values: two positive or negative infinite values are always equal. A comparison of a variable with positive infinity therefore evaluates to true if and only if its value is positive infinity.

To allow querying whether a particular floating point object has infinity as its value, two methods `JNukeFloat_isInf` and `JNukeDouble_isInf` were implemented, inspired by – but independent of – the `isinf` BSD library function.

Obtaining the floating point representation for Java class files

Methods were added to the `JNukeFloat` and `JNukeDouble` objects to convert their actual floating point value into the representation used in the constant pool of Java class files.

To obtain the binary representation of a floating point value, its value is first compared to the three special values mentioned in the previous sections. In case of a regular floating point value, the IEEE 754 representation is derived by computing the sign bit, exponent and mantissa, as described in [28, Sect 4.4.4/5]

8.3 Using the class writer object

Interaction with the class writer is limited to the `JNukeClassWriter` object, that is described in Appendix E.6. Writing a single Java class file is done in seven steps:

1. A new instance of a class writer object is created.
2. The descriptor object of the class to be written is registered with the class writer. As the name of a class file equals the name of the class it contains, setting the class descriptor will also automatically set the name of the new class file.
3. Items may be added to the constant pool using the provided methods, or an existing constant pool returned by the class loader may be used as a template.

[2] Solaris defines some math functions like `isinf` in `sunmath.h` instead of `math.h`, to name an example.

4. Facultatively, the output directory may be set explicitly. If this step is omitted, class files will be written to the default directory called `instr`. The output directory must exist, otherwise no class files are written. If a class file with the same name as the class file to be written already exists in the output directory, it will be overwritten.
5. The major and minor version[28, Section 4.1] of the resulting class file may be set. If the call to the corresponding interface functions is omitted, reasonable default values will be substituted[3].
6. Actual writing of the class file is done by calling method `writeClass`.
7. The class writer instance is destroyed and memory is released.

See Figure E.2 in Appendix E.6 for sample program code.

8.4 Summary

A possibility to write new class files to disk was added to the **JNuke** framework. The class writer delegates the more complex task of building the necessary representations for the constant pool and the method/field information tables to appropriate objects. An attribute builder was devised that can take existing descriptor objects and create the necessary class file attributes from it.

Support for special floating point values was added to the framework. Minor extensions were applied to the class loader and the descriptor objects to obtain optimal results.

[3]The implementation uses 45 as major version and 3 as minor version, as these versions are currently used by most Java compilers. The default values may be adapted whenever a new version of the Java platform with new features is released.

Chapter 9

Experiments

In this chapter, some experiments with the **jreplay** application are described. It implements the bytecode transformations presented in Chapter 6. Usage of the application is explained in Appendix A.

To verify that the replay mechanism is independent of the underlying virtual machine, the experiments were tested on the virtual machines shown in Figure 9.1.

VM Version (Build)	Architecture / OS
J2RE SE (1.3.1_03-b03)	x86 Linux 2.4.18-10
J2RE SE (1.4.0-b92)	x86 Linux 2.4.18-10
Solaris VM (1.2.2_10)	SunOS 5.8
J2RE SE (1.3.1_01)	SunOS 5.8
J2RE SE (1.4.0-rc-b91)	SunOS 5.8
J2RE SE (1.4.1-b21)	SunOS 5.8
J2RE SE (1.3.1-1)	alpha Linux 2.4.9-40

Figure 9.1: Virtual machines used to test the experiments

The thread schedules being replayed were written by hand. It was tried to capture thread schedules using a custom virtual machine written by Pascal Eugster as part of his thesis [14]. The JVM is built on the **JNuke** framework and allows to execute runtime verification algorithms to find thread schedules that lead to concurrency faults. Although a prototype **jcapture** utility was implemented, no working thread schedule could yet be captured. While instrumentation works on Java Bytecode, the JVM and the runtime verification algorithm were designed to work on abstract bytecode. Due to time constraints, the **jcapture** application does not yet perform the reverse mapping. Thread schedules obtained by **jcapture** can therefore not directly be used with the replay system presented in this thesis. Furthermore, the JVM currently only supports a subset of the Java foundation classes. Some real-world examples relying on currently missing classes such as `StreamTokenizer` or `InputStreamReader` could not be executed on the virtual machine.

For these reasons, only simple examples are presented in this chapter. They serve as a proof that the concept presented in this report actually works for a number of cases. Once a system to capture execution traces is implemented, thread replay schedules leading to concurrency errors in programs can be automatically obtained.

9.1 Overhead of instrumented thread changes

The program in Figure D.1 was used to measure the overhead of explicit thread changes, The bytecode of this program is shown in Figure 7.3a. A total number of three experiments were made with this program. In each experiment, the empty body of the loop was replaced with a call to a different method. The measured execution times are shown in Figure 9.2.

Description	Measured time	Factor
Uninstrumented program	10.5 sec	1.00
`Thread.yield`	161.6 sec	15.39
`Thread.sleep(0)`	685.7 sec	65.30
`replay.sync`	525.6 sec	50.06

Figure 9.2: Measured execution times for the `closedloop` example program

First, the comment in the program was replaced with a call to `Thread.yield`. The program took approximately fifteen times longer than without the call.

Second, `Thread.sleep(0)` was called inside the while loop. As `sleep` may throw an `InterruptedException`, it was called within a `try`...`catch` block. Adding the exception handler significantly slows down the program. `sleep` takes a parameter on the operand stack, which adds an additional overhead. The measured execution time suggests that it is about 65 times slower than the uninstrumented version.

Finally, the execution time of one billion instrumented thread changes were measured by inserting a call to `replay.sync`. The instrumented program was replaying the schedule shown in Figure D.2. As the program consists of only one thread, no change between threads occurs. This example measures only the overhead of parsing the thread replay schedule, the cost of calls to the method `replay.sync` and the overhead introduced by comparing absolute bytecode locations. Compared to the other function calls, `sync` introduces a moderate overhead of roughly 50.

To put things into perspective, it is important to note that the `closedloop` example program is worst case as it basically consists only of instrumented thread changes. Few real applications consist of one billion thread changes.

9.2 Race condition

The `r1` program given in Appendix D.2 suffers from an inherent race condition. Two threads `t1` and `t2` each try to assign a different value to a shared variable `x`. Initially, `x` has the value zero. If `t1` is scheduled before `t2`, the value of `x` is `2` at the end of the program. If `t2` is scheduled before `t1`, the value of `x` finally equals to `1`.

Two thread schedules shown in Figure D.4 and D.5 were devised that may be used for replay of these scenarios. The program acts as a proof that side effects in programs due to inherent race conditions can be deterministically replayed if an appropriate thread replay schedule is available. The running time of this example is too short to be used as a performance benchmark.

9.3 Deadlock

A classical deadlock situation occurs when two threads `t1`, `t2` try to acquire the locks for two objects **a**, **b** in different order. Such a situation is shown in Figure D.8. Unfortunate scheduling of the threads may cause a thread change from `t1` to `t2` at the very moment where `t1` just acquired the lock for object **a**. If `t1` is suspended, `t2` is scheduled next. While `t2` succeeds in acquiring the lock for object **b**, it fails in getting the lock for object **a**, as this lock is already held by thread `t1`. Thread `t2` is therefore suspended and control flow is transferred back to thread `t1`, which now tries to acquire the lock for object `b`. This lock is already held by thread `t2`. We now face the situation where both threads mutually wait for the other thread to release the lock they do not have acquired yet.

If the thread schedule given in Figure D.6 is replayed, the program runs into a deadlock. This example demonstrates that deadlocks in multi-threaded programs can be replayed deterministically. Again, the running time of this example is too short to be used as a performance benchmark.

9.4 wait / notify

The `waitnotify` example given in Figure D.4 has two threads that share a variable `counter`. The initial value of this variable is zero. The first thread increases the counter by two and suspends itself by using `wait`. Control is then transferred to the second thread that increases the counter by seven. The final result of the variable depends on how the JVM scheduler reacts to the `notify` sent by the second thread. This program was implemented to verify the correctness of the substitutes for `wait` and `notify`.

Three thread replay schedules are presented in Figures D.10 - D.12. Each schedule leads to a different value of the `counter` variable at the end of the program. Performance results were measured for 2216 thread changes using schedule A (see Figure D.10).

9.5 suspend / resume

A test program similar to `waitnotify` was devised to test the substitutes for `suspend` and `resume`. The source is given in D.13. The program performs 20’000 thread changes using the thread replay schedule in Figure D.14.

9.6 Summary of test results

Figure 9.3 shows the results of the experiments. The first line lists the measured times for the execution of the uninstrumented programs, including the delay for JVM startup. The running times for the `r1` and `deadlock` examples were omitted as they are too little to be taken as a reference. The last two lines list the number of instructions and the overall bytecode size of the uninstrumented program. The net change due to instrumentation is given in parentheses. The measured times generally were very short. Nevertheless, the results serve as a base for the following observations:

- The examples show that scenarios involving concurrency faults and thread changes can be successfully replayed.
- During startup, the replayer reads the replay schedule to be enforced at runtime. With increasing number of thread changes, parsing of the text files slows down program start.
- Although the measured times are slightly different with every execution, the overall comparative overhead ratio between the instrumented and the original program remains the same when the examples are repeatedly executed.

		r1	**deadlock**	**waitnotify**	**suspresume**
	Running time of uninstrumented program in seconds	—	—	1.00	5.50
–	Time to initialize JVM	—	—	0.30	0.30
=	Effective running time	—	—	0.70	5.20
	Running time of instrumented program in seconds	—	—	3.30	22.20
–	Time to initialize JVM & replayer	—	—	2.40	15.90
=	Effective time for replay	—	—	0.90	6.30
	Estimated replay overhead	—	—	1.28	1.21
	Number of classes	3	3	3	3
	Instructions inserted / deleted	31 / 1	18 / 1	42 / 3	34 / 4
	Bytes inserted / deleted	64 / 3	41 / 3	81 / 9	69 / 12
	Number of instructions	770 (+30)	1463 (+17)	1694 (+39)	1221 (+30)
	Overall bytecode size	1705 (+61)	2629 (+38)	3652 (+72)	2585 (+57)

Figure 9.3: Results of the simple experiments

All results in Figure 9.3 were obtained using the same computer. The machine was equipped with a 930 MHz Intel Pentium III Coppermine processor and 512 MB of memory. The machine was running version 7.3 (Valhalla) of the Red-Hat operating system with Linux kernel version 2.4.18-10. The example programs were compiled and executed using version 1.3.1_03 of the Java 2 Standard Edition (SE) compiler by Sun Microsystems, Inc.

Chapter 10

Evaluation and Discussion

10.1 Summary

It was shown using simple examples that race conditions and deadlock situations can be replayed using appropriate thread replay schedules.

Substitutes for the most important thread changes in programs are instrumented according to the concept presented in chapter 5. The **jreplay** application can handle calls to `Thread.sleep`, `yield`, `join`, `suspend`, and `resume`. It knows how to deal with `Object.wait`, `notify`, and `notifyAll`. With the exception of suspended threads in `Thread.join` or `Thread.sleep`, thread changes due to calls to `Thread.interrupt` are also correctly instrumented. The overhead introduced through instrumenting thread changes is moderate.

The results from the examples suggest that using the proposed concept, it is possible to deterministically replay regular non-deterministic Java programs on any conventional virtual machine. The system has not yet been tested with real applications, as there is currently no way to automatically capture execution traces. Note that development of such a tool was not part of the problem description.

10.2 Known limitations

Support for thread changes in native methods is missing.

Class files of an applications that are dynamically loaded using static calls or the method `Class.forName` are not yet automatically discovered during instrumentation. The transitive mechanism, however, detects creation of new object instances. Super classes that pass features to the application are also discovered.

With the current implementation, a deadlock may occur during replay if a thread terminates because of an error or an uncaught unchecked exception, as discussed in Section 5.4. During instrumentation, the `main` and `run` methods can be replaced by wrapper methods. A new `main` method would call the original method and catch any exception to unblock the next thread during replay, as shown in Figure 10.1.

The Java archive (jar) file format allows combining multiple files in a single archive file. The archive is compressed and allows easy deployment of applications over the internet.

```
1  static void main(String argv[]) {
2          try {
3                  original_main(argv);
4          } catch (Throwable t) {
5                  replay.terminate();
6          }
7  }
```

Figure 10.1: Wrapper method to prevent deadlocks during replay due to uncaught exceptions in an application

jreplay can currently not handle jar files. The archives of applications that need to be instrumented must be unpacked first.

Capturing thread schedules from non-terminating programs is a challenge, as the resulting schedule has infinite size. Partial recording – e.g., of the first n thread changes – or detecting loops in the thread schedule are possible ways to simplify the problem. The format of execution traces could be extended with commands `label` and `goto` for this purpose.

10.3 Future work

10.3.1 Large scale tests

As soon as thread replay schedules can be automatically captured, real-world tests should be made with large applications.

10.3.2 Operand stack depth

The heuristic described in 7.3.3 to update the maximum operand stack depth of an instrumented method is weak and should be replaced with a more sound approach. Although it works in most cases, an algorithm similar to the one used in a bytecode verifier should be implemented.

10.3.3 Support for thread groups

Thread groups allow to affect the state of a set of threads with a single call. Thread groups are organized by names. The top level thread group available in every Java application is called `main`. Each thread group may contain threads and other thread groups. Using Thread groups therefore allows organizing threads of an application in a hierarchical fashion. A thread cannot access information in thread groups other than its own group.

`ThreadGroup.interrupt` invokes the `interrupt` method on all the threads in a particular thread group and in all of its subgroups. The methods `suspend`, `resume`, `destroy`, and `stop` of the `ThreadGroup` class work correspondingly.

The reason why deterministic replay of applications with thread groups is currently not supported, is that class `ThreadGroup` is internal to most JVM implementations and therefore can not be instrumented.

One possibility for deterministically replaying applications using thread groups would be to instrument all method calls to the `ThreadGroup` class. For example, calls to `ThreadGroup.suspend` can be replaced by `ThreadGroupWrapper.suspend` in the instrumented program. The class `ThreadGroupWrapper` can be provided by the

instrumentation facility. Its implementation would iterate over all members of the thread group and perform the substitute corresponding to the method for each thread.

10.3.4 Support for applets and servlets

The **jreplay** application was designed to instrument stand-alone Java programs. These applications have a `main` method that can be instrumented to initialize the replayer. No effort was made to support other forms of Java binaries, such as Java applets or Java servlets. The reason why applets and servlets are not supported is that they do not have a traditional `main` method.

An *applet* is a small Java application that is embedded in a web page. Applets are classes that extend `java.applet.Applet`. When the web page is viewed with a Java-enabled browser, the applet is executed in a restricted Java virtual machine implemented within the web browser. Threads are frequently used in applets for animations and for asynchronously downloading data from the server in the background. To initialize the replayer, a static initializer could be added to the applet class for this purpose.

Servlets are Java programs running on a web server to dynamically build web pages. Servlets extend the `javax.servlet.http.HttpServlet` interface. A static initializer could be added or the methods `doGet`, `doPost` implemented in most servlets could alternatively be complemented with a call to initialize the replayer.

10.3.5 Support for Thread.stop

The method `Thread.stop` is deprecated, and it is unlikely that calls to this method will ever be encountered. Due to time constraints, no substitute was implemented.

Stopping a thread is inherently unsafe, as it causes the thread to unlock all its monitors, resulting in unpredictable behavior if one of the previously protected objects was in an inconsistent state [39]. If a thread is stopped during a capture phase, a context switch to the next thread is made. This context switch appears in the thread schedule and a call to `sync` is instrumented.

A stopped thread may be restarted using `start`. When a resurrected thread is rescheduled, a thread change is again logged in the thread replay schedule during replay.

To add support for `stop` methods, it is suggested that a similar substitute as used for `suspend` and `resume` is implemented. The actual calls to `stop` can be removed as the captured thread switch is already enforced.

Chapter 11

Conclusions

A conceptual model for deterministic replay of multi-threaded Java applications was developed. The necessary bytecode transformations for instrumentation were presented. Calls to an intermediary replayer are inserted into the program only where thread changes occurred in the original program. The replayer then performs scheduling according to this execution trace by blocking and unblocking threads at runtime. Existing calls that imply a thread change are replaced or removed to avoid side-effects during replay. The model can handle all thread-related events with the exception of thread groups. A generic representation for an execution trace has been devised. Thread schedules conforming to this standard are read both at the time of instrumentation and during replay.

The possibility to write class files, a facility for instrumenting byte code, and a parser to read text files with thread schedules to be replayed have been implemented. The existing class loader was extended to support more class file attributes. The floating point wrapper objects were improved by adding support for special values. Two libraries with a broad variety of instrumentation functions were added to the **JNuke** framework.

A **jreplay** application using all these extensions to enforce a given thread schedule in a program was developed. It allows automatic instrumentation of multi-threaded programs by detecting and instrumenting the class files of an application.

Simple experiments were made to verify the replay model. The required thread schedules were manually produced. The results show that instrumenting Java bytecode allows deterministic replay on any virtual machine conforming to the standards by Sun, including third party debuggers whose source code is not available. This is an advantage compared to other replay solutions that rely on customized virtual machines or particular library substitutes.

Using bytecode instrumentation to turn nondeterministic into deterministic programs can help reducing the required number of executions to fix concurrency-related errors in multi-threaded programs.

Appendix A

Usage of the jreplay Application

This chapter gives a brief introduction how the **jreplay** application is used from the perspective of a future user. This command line utility instruments class files using the transformations presented in chapter 6.

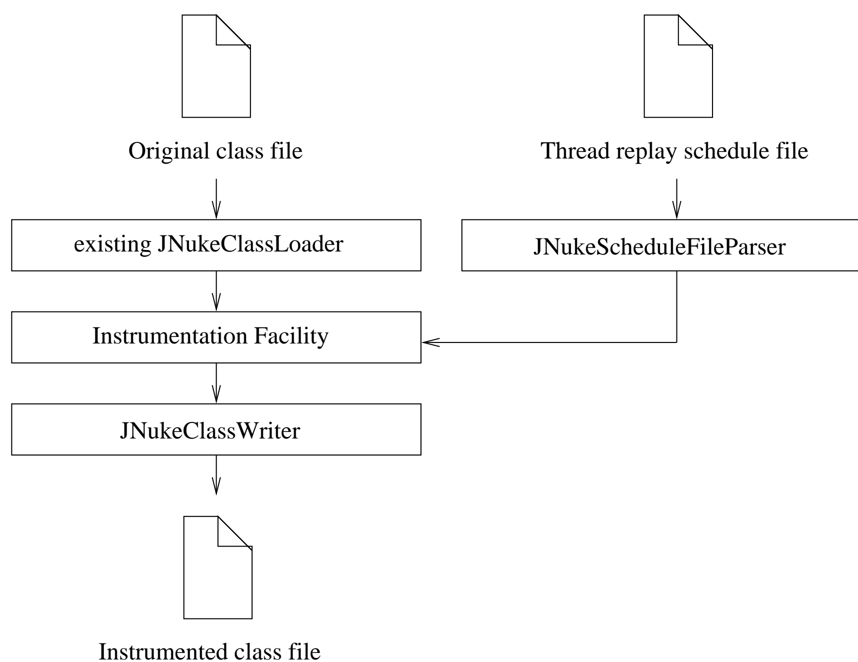


Figure A.1: Overview of the **jreplay** application layout

Figure A.1 gives an overview of the **jreplay** program. The original class file is read into memory using the existing **JNuke** class loader. The absolute locations where thread changes occur are extracted from the thread replay schedule that is parsed into a vector of `JNukeThreadChange` objects using the objects described in Appendix E.15 and E.14.

The instrumentation facility described in chapter 7 is used to modify the bytecode in memory. Finally, the instrumented class file is written back to disk using the class file writer presented in chapter 8.

Simple example

The most simple way to invoke the **jreplay** utility, is to use the following command line syntax:

```
jreplay filename.cx
```

The thread schedule file `filename.cx` will be parsed and the classes mentioned in the file will be instrumented. The file must follow the format specification given in Appendix B. Instrumented class files will be written to the default output directory called `instr`.

Specifying an initial class file

If the class file with the `main` function driving the application indirectly create threads, then the name of the class is not mentioned in the thread schedule file. In this particular case, the class file will not be instrumented. It is possible to specify the name of an initial class on the command line. The parameter must appear after the name of the thread schedule file, as shown in the following example:

```
jreplay filename.cx classname
```

Note that the name of the class is expected, not the name of the class file. The `.class` file extension may therefore not appear on the command line. If `classname` appears in the file describing the thread schedule, the additional parameter is ignored. Note that only one name of a class may be specified on the command line. Additional classes must appear in the thread schedule file when a thread change to them should be performed and/or they are automatically detected by the instrumentation facility when new instance objects are created. As only one class file can contain the `main` method, It does not make sense to specify multiple class files.

Verbose output

By adding the `--verbose` parameter, **jreplay** will produce output with additional information.

```
jreplay filename.cx --verbose
```

Specifying an alternate output directory

Optionally, an output directory may be set by using the `--dir` parameter. Both relative path settings or absolute paths are allowed. To write the instrumented class files into the parent directory, the command line parameter `--dir=..` may be used.

```
jreplay filename.cx --dir=..
```

Specifying a class path

If a class file to be analyzed and/or instrumented cannot be found in the current directory, **jreplay** additionally searches in directories specified by the `CLASSPATH` environment variable. Directories must be separated by the platform-dependent path separator token.

The `--classpath` parameter may be used to add directories to the class path. More than one class path parameters may appear on a command line. Note that **jreplay** always prints a warning message if it is unable to locate a class file.

```
export CLASSPATH=/home/joe:/opt/javaclasses
jreplay filename.cx --classpath=/pub/foo:/opt/bar --classpath=/mnt/java
```

If different class files with the same name exist in multiple directories, the order in which directories are specified is important. In the above example, **jreplay** would try to locate a class files in the following order:

current directory, `/home/joe`, `/opt/javaclasses`, `/pub/foo`, `/opt/bar`, `/mnt/java`.

Appendix B

Layout of a Thread Replay Schedule File

Execution traces are stored in text files with suffix `.cx` and read both during instrumentation and when initializing replay. The name of such files are determined at run time based on the name of the instrumented class file.

Empty lines and lines beginning with a hash (#) character are treated as comments and ignored. Every other line specifies the necessary circumstances under which a thread change occurs. The format of a non-comment line is given in Figure B.1.

<command> <thread index> <class name> <method index> <bcloc> <iter>

Figure B.1: Syntax of a single line defining a thread change in the replay schedule file

The meaning of the items is as follows:

command
: Currently, the `switch` command must be used. In the future, more commands may be supported.

thread index
: This integer value specifies the thread index. The main thread of an application has thread index 0. The value is successively increased with the creation of new thread objects.

class name
: This token specifies the name of the class where the thread change occurred.

method index
: This integer value specifies the method index in the class file. As Java supports polymorphic methods, i.e., methods with the same name but different signatures, the method name may not uniquely describe the position where the thread change occurred in the class file. The first method in a class file has index zero, the second method has an index value of one. Negative values are considered to be a syntax error.

bcloc
: This item specifies the exact bytecode location within the specified method where the thread change occurred. It must be a positive integer value that specifies the start of an instruction. Negative values are considered to be a syntax error.

iter

This token counts the number of successive invocations of the bytecode instruction. A value of 1 states that a thread change should occur the first time the instruction is executed. A value of 2 states that the first pass is ignored and that the thread change is taken the second time directly before the instruction is executed. The value may not equal to zero, as the thread change would obviously never be taken. Negative values are considered a syntax error.

Every line of the file describes the necessary circumstances, under which an explicit thread switch must be enforced. During replay, thread changes of the replay schedule are stored in a *queue*. As soon as the full rule is matched, a thread change is performed and the first element in the queue is dropped. The next thread in the schedule is given by the next line in the thread schedule file.

Example

Consider the very simple example file given in Figure B.2. The execution trace consists of only three thread changes.

The first thread change occurs if thread `0` – the main thread – executes the bytecode instruction at location 12 of the second method of class `SimpleExample` for the first time. The **jreplay** will insert a call to the replayer at this location. During replay, the replayer will perform a thread change to thread `1` as this is the next thread in the replay schedule on the next line.

The second thread change will be instrumented at location 37 of the first method of the inner class `SimpleExample$inner1`. This time, the thread change should occur at the third time the instruction is executed since the last thread change occurs. For the first two calls, the replayer decrements a counter for this purpose, but a thread change will not occur. The third time the instruction is executed, the thread change is enforced.

According to the third and last line, the thread scheduled next is again the main thread. When the instruction at bytecode location 90 is reached, deterministic replay will end and all blocked threads will be released. The last line in a thread schedule usually marks the exit point of a program.

```
switch   0   SimpleExample          2   12   1
switch   1   SimpleExample$inner1   1   37   3
switch   0   SimpleExample          2   90   1
```

Figure B.2: Example thread schedule file

You can find more examples in Appendix D.

Appendix C

Replayer

The **jreplay** application adds various calls to a replayer to its input program. Section C.1 presents the Java classes the replayer consists of and Section C.2 summarizes all public replayer methods.

C.1 Classes

The replayer consists of five Java classes:

replay
: The actual replayer. Interaction between the instrumented program and the replay mechanism is done through this class, the other classes are for internal use by the replayer only. There may be at most one replayer present. The replayer is therefore implemented as a static class. It has no explicit constructor, but uses an initializer method.

task
: A wrapper class for an object of type `Thread` providing methods to block and unblock a task.

taskPool
: Maintains an ordered list of `task` objects. The replayer has a single instance of a `taskPool` object that mirrors the set of threads in the application to be replayed. The replayer can register new threads with the `taskPool` by using the method `registerThread`. The transfer method can be used to transfer control to a particular thread.

condition
: A `condition` contains the necessary information to perform a single thread switch.

schedule
: This class maintains the complete replay schedule in a vector of `condition` objects. An external file representation of a schedule may be parsed using the method `readFromFile`. This object also provides methods to query whether the schedule is empty (`hasNext`), to get the next object in the schedule (`firstElement`) and to drop the first element in the queue (`dropFirst`) when the corresponding task switch was performed.

C.2 Interface to the replayer

Figure C.1 lists the interface methods to the replayer. These methods are intended to be invoked by the replayed class. The calls are inserted during instrumentation.

```
public static void init();
public static void registerThread(Thread t);
public static void sync(String classname, int methidx, int loc);
public static void terminate(int loc);
public static void blockThis(Thread t);
public static void interruptSelf(Thread t);
public static void setLock(Object o);
public static void resetLock();
```

Figure C.1: Programming interface to the replayer

The public methods are now described in more detail. Entries for all replayer methods have to be inserted once into the constant pool during instrumentation of a class file.

replay.init

A call to `init` initializes the replayer at runtime. The instrumentation facility inserts a call to this method as first instruction of a `main` method of an application. The thread schedule to be replayed is read from the text file and the main thread of the program is registered with the `taskPool` data structure.

If the schedule file cannot be found in the current directory, it is searched in the parent directory. As instrumented class files are by default written to a direct subdirectory, this allows automated testing of the bytecode transformations. If the replay schedule file cannot be found in either the current directory or the parent directory, an error message is printed and replay is aborted. Replay is also aborted if a syntax error is encountered in the thread schedule file.

replay.registerThread

A call to this method is used to register a new thread object with the replayer. The thread object to be registered is passed as first parameter. A task object is created and added to the `taskPool` vector. The number of the slot in the vector determines its unique task identifier.

replay.sync

To determine at runtime, whether an explicit thread change should be performed, a call to the `sync` method is inserted at the relevant locations. When a thread change should occur during replay, the next thread in the replay schedule is automatically determined and unblocked by calling `notify` on the current lock object of the task object associated with the next thread. The current thread is blocked by calling `wait` on the current lock object of the task object associated with the current thread.

To notify the replayer about the current bytecode location at runtime, three parameters are passed to the function. The first parameter contains the name of the class where the call to sync occurred as a String. As Java supports polymorphic methods that have the same name but different signatures, the name of the method does not uniquely describe a method. The second parameter therefore contains the method index in the class. A value of zero is

passed for the first method in the class file. Subsequent methods have higher indices. As third and last parameter, the absolute bytecode within said method and said class is passed. The parameter reflects the byte code location within the original program.

replay.terminate

`terminate` will perform an additional `unblock` operation on the current thread in the schedule. A call to this method is inserted before a thread dies. This method does not expect any parameters, as the current thread can be determined at runtime.

replay.blockThis

`blockThis` blocks the current thread and is inserted as first instruction of the `run` method within runnable classes. The thread to be blocked is automatically determined by the replayer.

replay.interruptSelf

Calls to `Thread.interrupt` are replaced with a call to `replay.interruptSelf`. The thread to be interrupted is expected as first argument.

replay.setLock

This method is used to implement the `Object.wait` substitute and temporarily sets the lock of the task object associated with the current thread.

replay.resetLock

Calling this method resets the lock of the task object associated with the current thread to the default lock. This method is used to instrument `Object.wait`.

Appendix D

Uninstrumented Java Example Programs

This chapter features some small Java programs that are interesting enough to show certain aspects of concurrent programming. These programs are referenced in the text. Note that all programs are given in uninstrumented Java source code.

D.1 closed loop

```
1  public class closedloop {
2     public static void main(String argv[]) {
3         int i;
4         for (i=0; i<1000000000; i++) {
5             /* do nothing */
6         }
7     }
8  }
```

Figure D.1: Example program to measure the overhead of instrumented explicit thread changes

Thread replay schedule

```
switch   0   closedloop   1   8   1000000000
```

Figure D.2: Thread replay schedule for example `closedloop`

D.2 r1

`r1.java` is a program whose main thread creates two threads `t1` and `t2`. The two threads share a common variable `x`. Access to this variable is not synchronized. This is a simple example of a *race condition*. When the program terminates, the variable `x` is assigned the value given by the thread that is scheduled last. When both threads are terminated, the main thread resumes and the value of `x` is printed on standard output.

```
1  public class r1 {
2
3      public static volatile int x = 0;
4
5      public static void main(String argv[]) {
6          t1 T1 = new t1();
7          t2 T2 = new t2();
8          T1.start();
9          T2.start();
10         while (T1.isAlive() || T2.isAlive()) {
11             try {
12                 Thread.currentThread().sleep(500);
13             } catch (InterruptedException e) {
14                 System.out.println(e.getMessage());
15             }
16         }
17         System.out.println("x is " + x);
18     }
19 }
20
21 class t1 extends Thread {
22     public void run() {
23         System.out.println("t1.begin");
24         r1.x = 1;
25         System.out.println("t1.end");
26     }
27 }
28
29 class t2 extends Thread {
30     public void run() {
31         System.out.println("t2.begin");
32         r1.x = 2;
33         System.out.println("t2.end");
34     }
35 }
```

Figure D.3: Source code of the r1 example program

Figures D.4 and D.5 show two possible thread replay schedules.

Thread replay schedule A

Thread schedule A shown in Figure D.4 executes thread `t1` before thread `t2`. The resulting value of the variable `x` is `2`.

```
switch  0  r1  1  24  1
switch  1  t1  1   3  1
switch  1  t1  1  15  1
switch  2  t2  1   5  1
switch  2  t2  1  17  1
```

Figure D.4: Thread replay schedule A for example `r1`

Thread replay schedule B

Thread schedule B shown in Figure D.5 executes thread `t2` before thread `t1`. The resulting value of the variable `x` is `1`.

```
switch   0   r1   1   24   1
switch   2   t2   1    5   1
switch   2   t2   1   17   1
switch   1   t1   1    3   1
switch   1   t1   1   15   1
```

Figure D.5: Thread replay schedule B for example `r1`

D.3 deadlock

As the name suggests, a deadlock may occur when the program `deadlock.java` given in Figure D.8 is executed. The main thread creates two objects `a` and `b`, as well as two threads `t1` and `t2`. Both threads enter into monitors of both variables, but in different order.

If execution of thread `t1` is intercepted by the JVM scheduler when it holds both locks, a deadlock occurs when thread `t2` enters the monitor on line 39

A good rule of thumb to prevent deadlocks in the first place is to lock variables always in the same order. This rule is obviously violated here. If the monitors were entered in the same order, thread `t2` would have been suspended as `t1` is already in the monitor. Execution would proceed with thread `t2`, who could release both monitors and `t1` could then enter its monitor without the possibility of a deadlock.

Thread replay schedule A

The thread schedule shown in Figure D.6 leads to a deadlock.

```
switch   0   deadlock       2   48   1
switch   1   deadlock$D1    1   20   1
switch   2   deadlock$D2    2   17   1
switch   2   deadlock$D2    2   35   1
```

Figure D.6: Thread replay schedule A for the `deadlock` example

Thread replay schedule B

The thread schedule shown in Figure D.7 executes the first thread before the second thread, therefore no deadlock occurs.

```
switch   0   deadlock       2   48   1
switch   1   deadlock$D1    1   50   1
switch   2   deadlock$D2    1   66   1
switch   0   deadlock       1   12   2
```

Figure D.7: Thread replay schedule B for the `deadlock` example

```
1   public class deadlock {
2       private Object a, b;
3
4       public static void main(String[] argv) {
5           deadlock d = new deadlock();
6           d.bootstrap();
7       }
8
9       public void bootstrap() {
10          D1 t1;
11          D2 t2;
12
13          a = new Object();
14          b = new Object();
15
16          t1 = new D1();
17          t1.start();
18          t2 = new D2();
19          t2.start();
20
21          while (t1.isAlive() || t2.isAlive()) {
22              Thread.currentThread().yield();
23          }
24      }
25
26      public class D1 extends Thread {
27          public void run() {
28              synchronized (a) {
29                  synchronized(b) {
30                  }
31              }
32              System.out.println("D1.end");
33          }
34      }
35
36      public class D2 extends Thread {
37          public void run() {
38              System.out.println("x");
39              synchronized (b) {
40                  System.out.println("y");
41                  synchronized(a) {
42                  }
43              }
44              System.out.println("D2.end");
45          }
46      }
47  }
```

Figure D.8: Source code of the deadlock example program

D.4 waitnotify

```
1  public class waitnotify {
2      private static Thread t1, t2;
3      public static int counter;
4      final static int limit = 10000;
5
6      public static void main(String[] argv) {
7          waitnotify me = new waitnotify();
8          me.bootstrap();
9      }
10
11     public void bootstrap() {
12         counter = 0;
13         t1 = new Twait1();
14         t2 = new Twait2();
15         t1.start();
16         t2.start();
17         while (t1.isAlive() || t2.isAlive()) {
18         }
19         System.out.println("\nFinal value is " + counter);
20     }
21
22     class Twait1 extends Thread {
23         public void run() {
24             while (counter < limit){
25                 synchronized(t1) {
26                     System.out.print("A" + counter + " ");
27                     counter += 2;
28                     try {
29                         t1.wait();
30                     } catch (InterruptedException e) {
31                         System.out.println(e.getMessage());
32                     }
33                 }
34             }
35             System.out.println("End t1");
36         }
37     }
38
39     class Twait2 extends Thread {
40         public void run() {
41             while (counter < limit) {
42                 System.out.print("B" + counter + " ");
43                 synchronized(t1) {
44                     t1.notify();
45                 }
46                 counter += 7;
47             }
48             synchronized(t1) {
49                 t1.notify();
50             }
51             System.out.println("End t2");
52         }
53     }
54 }
```

Figure D.9: Source code of the `waitnotify` example program

Thread replay schedule A

```
switch   0   waitnotify          2    38   1
switch   1   waitnotify$Twait1   1    52   1
switch   2   waitnotify$Twait2   1    44   1
switch   1   waitnotify$Twait1   1    52   1
switch   2   waitnotify$Twait2   1    44   1
switch   1   waitnotify$Twait1   1    52   1
switch   2   waitnotify$Twait2   1    44   1
switch   1   waitnotify$Twait1   1    52   1
...      ... ...                 ...  ...  ...
switch   2   waitnotify$Twait2   1    44   1
switch   1   waitnotify$Twait1   1    52   1
switch   2   waitnotify$Twait2   1    44   1
switch   1   waitnotify$Twait1   1    52   1
switch   2   waitnotify$Twait2   1    44   1
switch   1   waitnotify$Twait1   1    52   1
switch   2   waitnotify$Twait2   1    44   1
switch   1   waitnotify$Twait1   1    52   1
switch   2   waitnotify$Twait2   1    44   1
switch   1   waitnotify$Twait1   1    52   1
switch   2   waitnotify$Twait2   1    44   1
switch   1   waitnotify$Twait1   1    52   1
switch   2   waitnotify$Twait2   1    44   1
switch   1   waitnotify$Twait1   1    52   1
switch   2   waitnotify$Twait2   1    44   1
switch   1   waitnotify$Twait1   1    96   1
switch   2   waitnotify$Twait2   1   105   1
switch   0   waitnotify          2    38   1
```

Figure D.10: Thread replay schedule A for the `waitnotify` example

Thread replay schedule B

```
switch   0   waitnotify          2    38      1
switch   1   waitnotify$Twait1   1    52      1
switch   2   waitnotify$Twait2   1    44   1429
switch   1   waitnotify$Twait1   1    96      1
switch   2   waitnotify$Twait2   1   105      1
switch   0   waitnotify          2    38      1
```

Figure D.11: Thread replay schedule B for the `waitnotify` example

Thread replay schedule C

```
switch   0   waitnotify          2    38      1
switch   1   waitnotify$Twait1   1    52   4999
switch   2   waitnotify$Twait2   1   105      1
switch   1   waitnotify$Twait1   1    96      1
switch   0   waitnotify          2    38      1
```

Figure D.12: Thread replay schedule C for the `waitnotify` example

D.5 suspresume

The suspend/resume program example repeatedly performs thread changes between two threads `sr1` and `sr2` by suspending and resuming the first thread.

```
1   public class suspresume {
2
3       private Thread t1, t2;
4       private int counter;
5
6       public static void main(String[] argv) {
7           suspresume me = new suspresume();
8           me.bootstrap();
9       }
10
11      public void bootstrap() {
12          counter = 10000;
13          t1 = new sr1();
14          t2 = new sr2();
15          t1.start();
16          t2.start();
17
18          while (counter>0) {
19              try {
20                  Thread.currentThread().sleep(500);
21              } catch (InterruptedException e) {
22                  System.out.println(e.getMessage());
23              }
24          }
25          System.out.println("\ndone.");
26      }
27
28      public class sr1 extends Thread {
29          public void run() {
30              while (counter>0) {
31                  System.out.println(counter);
32                  counter--;
33                  t1.suspend();
34              }
35          }
36      }
37
38      public class sr2 extends Thread {
39          public void run() {
40              while (counter>0) {
41                  System.out.print("B");
42                  t1.resume();
43              }
44              t1.resume();
45          }
46      }
47  }
```

Figure D.13: Example program for suspend resume

Thread replay schedule

```
switch   0   suspresume       2   45   1
switch   1   suspresume$sr1   1   31   1
switch   2   suspresume$sr2   1   18   1
switch   1   suspresume$sr1   1   31   1
switch   2   suspresume$sr2   1   18   1
switch   1   suspresume$sr1   1   31   1
switch   2   suspresume$sr2   1   18   1
switch   1   suspresume$sr1   1   31   1
switch   2   suspresume$sr2   1   18   1
switch   1   suspresume$sr1   1   31   1
switch   1   suspresume$sr1   1   31   1
switch   2   suspresume$sr2   1   18   1
switch   1   suspresume$sr1   1   31   1
switch   2   suspresume$sr2   1   18   1
switch   1   suspresume$sr1   1   31   1
...     ...  ...             ...  ...  ...
switch   1   suspresume$sr1   1   31   1
switch   2   suspresume$sr2   1   18   1
switch   1   suspresume$sr1   1   31   1
switch   2   suspresume$sr2   1   18   1
switch   1   suspresume$sr1   1   31   1
switch   2   suspresume$sr2   1   38   1
switch   1   suspresume$sr1   1   44   1
```

Figure D.14: Thread replay schedule for `suspresume`

Appendix E

Programming Interface to Implemented Objects

As part of this thesis, the **JNuke** framework for checking Java bytecode [15] has been extended with various new objects. Figure E.1 shows an UML diagram of the implemented objects that are presented in this chapter.

A new `JNukeInstrBCT` class loader transformation can instrument a Java class file and write the results back to disk. It uses two objects `JNukeInstrument` and `JNukeClassWriter` for this purpose.

`JNukeInstrument` is a generic facility for instrumenting Java bytecode in memory. Its implementation is described in Chapter 7. Instrumentation is done using a set of rules. Each rule is represented as a `JNukeInstrRule` object and contains two functions `eval` and `exec`. The functions can be generated using two libraries `JNukeEvalFactory` and `JNukeExecFactory`. A `JNukeReplayFacility` object was developed, that provides the necessary instrumentation rules to achieve deterministic replay. The thread schedule to be replayed is stored in a vector of `JNukeThreadChange` objects obtained from a text file using a `JNukeScheduleFileParser`. A `JNukeConstantPool` object allows modifying the constant pool obtained from the class loader at runtime.

Information about each method presented in the following sections consists of the following parts:

Signature
: The signature of the method is presented as it would appear in a typical header include file used in the C programming language. To enhance readability, the signature is spread over multiple lines and a vertical bar was added to distinguish it from the rest of the text.

Description
: A brief synopsis of the method is presented. Important points and possible side-effects are discussed and if applicable, cross references to similar methods are given.

Parameters
: All method parameters are listed and explained. Where it is not clear from the context, information about the expected type was added.

Return value
: If the return type of the function differs from `void`, possible return values of the method are presented.

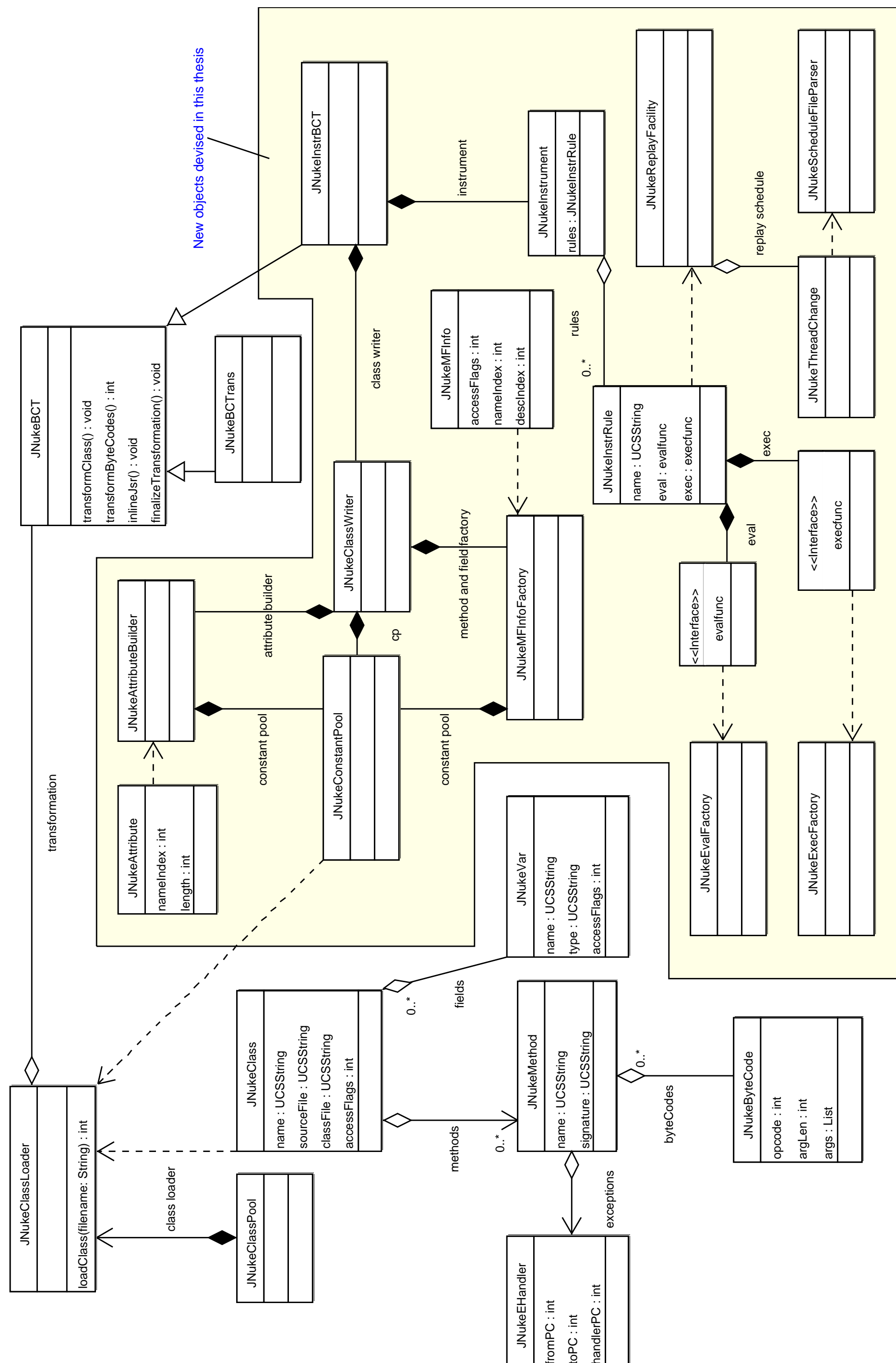


Figure E.1: UML diagram of **JNuke** objects relevant for this report

E.1 JNukeConstantPool

JNukeConstantPool is an object that has been written to modify a constant pool of a class file.

The representations of the constant pool in the class loader and in the class writer are slightly different. The class loader returns the constant pool as a an array of variable size where each item in the array refers to a struct containing the data of the corresponding constant pool entry. As the first item in the constant pool has index 1, the array element with index zero is never used. The JNukeConstantPool does not waste this entry. For historical reasons, entries of type long and double take up two entries in the constant pool [28, 4.4.5]. The second entry must not be used for compatibility reasons. The class loader simply skips over these entries. As the JNukeConstantPool object allows deleting of entries, it assigns a special *reserved* tag. By deleting an entry that takes up two slots in the constant pool, both entries are deleted.

These differences require a transformation of the constant pool when it is assigned to the class writer in terms of the very first element and elements that occupy multiple entries, such as long and double values. The necessary changes are automatically performed when a constant pool obtained by the class loader is registered with the class writer using the setConstantPool method.

E.1.1 Constructor

```
JNukeObj *
JNukeConstantPool_new (
  JNukeMem * mem
)
```

Description

The constructor is called to obtain a new object instance.

Parameters

mem
: A memory manager object of type JNukeMem. It is being used to allocate memory for the new object instance.

Return value

A pointer to a new instance of a JNukeConstantPool object.

E.1.2 JNukeConstantPool_addClass

```
int
JNukeConstantPool_addClass(
  JNukeObj * this,
  const int nameidx
)
```

Description

Add an entry of type CONSTANT_Class to a constant pool and return the index of the element. If there is already an identical copy of the element, the constant pool is not modified and the existing element is returned instead. The memory required is allocated using the memory manager associated with the constant pool object.

Parameters

`this`
A `JNukeConstantPool` object reference to the constant pool, to which the entry should be added.

`nameidx`
The parameter to the `CONSTANT_Class` entry as described in [28, 4.4.1].

Return value

This function returns the index of the element in the constant pool.

E.1.3 JNukeConstantPool_addFieldRef

```
int
JNukeConstantPool_addFieldRef(
  JNukeObj * this,
  const int classidx,
  const int natidx
)
```

Description

Add an entry of type `CONSTANT_FieldRef` [28, 4.4.2] to a constant pool and return the index of the element. If there is already an identical copy of the element, no item is added to the constant pool and the existing element is returned instead. The memory required is allocated using the memory manager associated with the constant pool object.

Parameters

`this`
A `JNukeConstantPool` object reference to the constant pool, to which the entry should be added.

`classidx`
Index into the constant pool declaring the name of the class, to whom this field belongs.

`natidx`
Index into the constant pool declaring the `CONSTANT_NameAndType` [28, 4.4.6] descriptor of the field.

Return value

Index of the element in the constant pool.

E.1.4 JNukeConstantPool_addMethodRef

```
int
JNukeConstantPool_addMethodRef(
  JNukeObj * this,
  const int classidx,
  const int natidx
)
```

Description

Add an entry of type `CONSTANT_MethodRef` [28, 4.4.2] to a constant pool. This method otherwise behaves similar as `JNukeConstantPool_addFieldRef`.

Parameters

`this`
: a `JNukeConstantPool` object reference to the constant pool, to which the entry should be added.

`classidx`
: Index into the constant pool declaring the name of the class to whom this method belongs.

`natidx`
: Index into the constant pool declaring the `CONSTANT_NameAndType` [28, 4.4.6] descriptor of this method.

Return value

Index of the entry in the constant pool.

E.1.5 JNukeConstantPool_addInterfaceMethod

```
cp_info *
JNukeConstantPool_addInterfaceMethod(
  JNukeObj * this,
  const int classidx,
  const int natidx
)
```

Description

Add an entry of type `CONSTANT_InterfaceMethodRef` [28, 4.4.2] to the constant pool. This method otherwise behaves similar as `JNukeConstantPool_addFieldRef`.

Parameters

`this`
: a `JNukeConstantPool` object reference to the constant pool, where the entry should be added

`classidx`
: Index into the constant pool declaring the name of the class to whom this interface method reference belongs.

`natidx`
: Index into the constant pool declaring the `CONSTANT_NameAndType` [28, 4.4.6] entry associated with this element.

Return value

Constant pool index of the new entry.

E.1.6 JNukeConstantPool_addString

```
int
JNukeConstantPool_addString(
  JNukeObj * this,
  const int utf8idx
)
```

Description

Add an entry of type `CONSTANT_String` to the constant pool. This method otherwise behaves similar as `JNukeConstantPool_addFieldRef`.

Parameters

`this`
: a `JNukeConstantPool` object reference to the constant pool, where the entry should be added

`utf8idx`
: Index into the constant pool stating the `CONSTANT_Utf8` [28, 4.4.7] entry with the actual string contents.

Return value

Constant pool index of the new entry.

E.1.7 JNukeConstantPool_addInteger

```
int
JNukeConstantPool_addInteger(
  JNukeObj * this,
  const int value
)
```

Description

Add a new entry of type `CONSTANT_Integer` to the constant pool. This method otherwise behaves similar as `JNukeConstantPool_addFieldRef`.

Parameters

`this`
: a `JNukeConstantPool` object reference to the constant pool, where the entry should be added

`value`
: The integer value of the entry to be added.

Return value

Constant pool index of the new entry.

E.1.8 JNukeConstantPool_addFloat

```
int
JNukeConstantPool_addFloat(
  JNukeObj * this,
  const float value
)
```

Description

Add an entry of type `CONSTANT_Float` [28, 4.4.4] to the constant pool. This method otherwise behaves as `JNukeConstantPool_addFieldRef`.

Parameters

`this`
: a `JNukeConstantPool` object reference to the constant pool, where the entry should be added

`value`
: The float value of the entry to be added

Return value

Constant pool index of the new entry.

E.1.9 JNukeConstantPool_addLong

```
int
JNukeConstantPool_addLong(
  JNukeObj * this,
  const long value
)
```

Description

Add an entry of type `CONSTANT_Long` [28, 4.4.5] to the constant pool. This method otherwise behaves as `JNukeConstantPool_addFieldRef`.

Parameters

`this`
: a `JNukeConstantPool` object reference to the constant pool, where the entry should be added.

`value`
: The long value of the entry to be added.

Return value

Constant pool index of the new entry.

E.1.10 JNukeConstantPool_addDouble

```
cp_info *
JNukeConstantPool_addDouble(
  JNukeObj * this,
  const double value
)
```

Description

Add an entry of type `CONSTANT_Double` [28, 4.4.5] to the constant pool. This method otherwise behaves as `JNukeConstantPool_addFieldRef`.

Parameters

`this`
: a `JNukeConstantPool` object reference to the constant pool, where the entry should be added.

`value`
: The double value to be added.

Return value

Constant pool index of the new entry.

E.1.11 JNukeConstantPool_addNameAndType

```
int
JNukeConstantPool_addNameAndType(
  JNukeObj * this,
  const int nameidx,
  const int descidx
)
```

Description

Add an entry of type `CONSTANT_NameAndType` [28, 4.4.6] to the constant pool. This method otherwise behaves as `JNukeConstantPool_addFieldRef`.

Parameters

`this`
: a `JNukeConstantPool` object reference to the constant pool, where the entry should be added

`nameidx`
: Index of the constant pool entry referring to the name of this `CONSTANT_NameAndType` entry.

`descidx`
: Index of the constant pool entry referring to the descriptor of this `CONSTANT_NameAndType` entry.

Return value

Constant pool index of the entry to be added.

E.1.12 JNukeConstantPool_addUtf8

```
cp_info *
JNukeConstantPool_addUtf8(
  JNukeObj * this,
  const JNukeObj * str
)
```

Description

Add an entry of type `CONSTANT_Utf8` [28, 4.4.7] to the constant pool. This method otherwise behaves as `JNukeConstantPool_addFieldRef`.

Parameters

`this`
: a `JNukeConstantPool` object reference to the constant pool, where the entry should be added

`str`
: Reference to a `UCSString` object with the string to be added.

Return value

Constant pool index of the entry to be added.

E.1.13 JNukeConstantPool_addNone

```
cp_info *
JNukeConstantPool_addNone(
  JNukeObj * this )
```

Description

This method adds a special entry of type `CONSTANT_None` to the constant pool. This method is for internal use only and should never be called.

Parameters

`this`
: Reference to a constant pool object

Return value

A pointer to the new constant pool entry.

E.1.14 JNukeConstantPool_addReserved

```
cp_info *
JNukeConstantPool_addReserved(
  JNukeObj * this
)
```

Description

Add an entry of type `CONSTANT_Reserved` to the constant pool. This method is for internal use only and should never be called.

Parameters

`this`
: Reference to a constant pool object.

Return value

E.1.15 JNukeConstantPool_count

```
int
JNukeConstantPool_count(
  JNukeObj * this
)
```

Description

Count the number of elements in a constant pool.

Parameters

`this`
: a `JNukeConstantPool` object reference to the constant pool, whose elements should be counted

Return value

The number of elements in the constant pool. An empty constant pool consists of zero elements. The maximum number of elements is currently limited by the JVM specification.

E.1.16 JNukeConstantPool_removeElementAt

```
void
JNukeConstantPool_removeElementAt(
  JNukeObj * this,
  const int index
)
```

Description

Remove an element with a given index from the constant pool. The index of the entries following the item to be removed will be decreased. Byte code referencing the constant pool to be modified must therefore be updated.

Parameters

this
Reference to the constant pool object where an element should be removed.

index
Index of the element to be removed.

Return value

None.

E.1.17 JNukeConstantPool_elementAt

```
cp_info *
JNukeConstantPool_elementAt(
  JNukeObj * this,
  const int index
)
```

Description

Return element with a given index. The element is returned as a `struct cp_info *`, defined in `java.h`.

Parameters

this
a `JNukeConstantPool` object reference to the constant pool

index
Index of the item to be returned.

Return value

A pointer to the desired element. If the object does not exist in the pool, the behavior is not specified.

E.1.18 JNukeConstantPool_find

```
int
JNukeConstantPool_find(
  JNukeObj * this,
  const cp_info * elem
)
```

Description

Locate a particular element in the constant pool. If there is no entry matching the pointer value, `-1` is returned.

Parameters

`this`
: a `JNukeConstantPool` object reference to the constant pool

`elem`
: Pointer to be searched in the constant pool.

Return value

This function returns the element index.

E.1.19 JNukeConstantPool_findReverse

```
int
JNukeConstantPool_findReverse(
  JNukeObj * this,
  const cp_info * elem
)
```

Description

This function is similar as `JNukeConstantPool_find`, except that it searches backward, starting with the last item of the constant pool.

Parameters

`this`
: a `JNukeConstantPool` object reference to the constant pool

`elem`
: Pointer to be searched in the constant pool.

Return value

This function returns the index of the element matching the search template in the constant pool. If the element could not be matched, the value `-1` is returned.

E.2 JNukeAttribute

An attribute is essentially a data buffer of variable size that has an index into a constant pool associated with it. Furthermore, attributes may be nested, i.e., it is possible to create a hierarchy made of sub- and super-attributes. Every attribute is therefore also a container of zero or more subattributes.

E.2.1 Constructor

```
JNukeObj *
JNukeAttribute_new(
  JNukeObj * mem
)
```

Description

The constructor is called to obtain a new `JNukeAttribute` object instance.

Parameters

mem
: A memory manager object of type `JNukeMem`. It is being used to allocate memory for the new object instance.

Return value

A pointer to a new instance of a `JNukeAttribute` object.

E.2.2 JNukeAttribute_getLength

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

Description

Get the length of the data area in bytes. The length is always positive. A value of zero denotes that there is no data associated with this attribute.

Parameters

this
: Reference to an object of type `JNukeAttribute` whose data area length should be returned.

Return value

The length of the data area in bytes.

E.2.3 JNukeAttribute_getData

```
char *
JNukeAttribute_getData (
  const JNukeObj * this
)
```

Description

Get a pointer to the data area associated with this attribute.

Parameters

this
Reference to the attribute object whose data area should be returned.

Return value

A pointer to the data area associated with this attribute.

E.2.4 JNukeAttribute_setData

```
void
JNukeAttribute_setData (
  JNukeObj * this,
  char *data,
  const int size
)
```

Description

Set the data area associated with this attribute.

Parameters

this
Reference to the attribute object whose data should be set.

data
Pointer to the data area. The data area must be have at least `size` bytes.

size
Size of the data area. May not be negative.

Return value

None.

E.2.5 JNukeAttribute_getNameIndex

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

Description

Obtain the current value of the name index property of this attribute. The name index is an index of an item in a constant pool defining the name of the attribute. Use `JNukeAttribute_setNameIndex` to set the name index.

Parameters

`this`
Reference to the attribute object whose name index should be returned.

Return value

The name index associated with this attribute.

E.2.6 JNukeAttribute_setNameIndex

```
void
JNukeAttribute_setNameIndex (
  JNukeObj * this,
  const int idx
)
```

Description

Set name index associated with this attribute.The `JNukeAttribute_getNameIndex` function may be used to query the current value of this property.

Parameters

`this`
Reference to the attribute object whose name index should be modified.

`idx`
New value for the name index property. The value must be greater than zero. Furthermore, the constant pool entry is expected to be of type `CONSTANT_Utf8` [28, 4.4.7].

Return value

None.

E.2.7 JNukeAttribute_append

```
void
JNukeAttribute_append (
  JNukeObj * this,
  const char *data,
  int size
)
```

Description

Append zero or more bytes at the end of the data area associated with this attribute. The necessary memory is obtained by reallocating the data buffer. As a result, previously obtained pointers to the data area using the `JNukeAttribute_getData` may be invalidated. The `JNukeAttribute_write` function may be used to overwrite values anywhere in the data buffer associated with this object.

Parameters

this
Reference to the attribute object where the data should be appended.

data
Pointer to the data that should be appended.

size
Number of bytes to be appended. The value must be positive. If `size` equals zero, the statement has no effect.

Return value

None.

E.2.8 JNukeAttribute_write

```
void
JNukeAttribute_write (
  JNukeObj * this,
  int ofs,
  const char *data,
  int size
)
```

Description

Partially or completely overwrite data in an attribute object. This method may only be used to overwrite data that has previously been added to the attribute. The function `JNukeAttribute_append` may be used for this purpose.

Parameters

this
Reference to the attribute object whose data should be partially or completely overwritten.

ofs
Start position of where data should be copied to. A value of zero denotes the first data byte in the attribute, i.e., the beginning. `ofs` may not be negative.

data
Pointer to the data who acts as source for the write process. The data area must have at least `ofs+size` bytes.

size
Number of bytes to be copied. `size` must be a positive integer. If `size` equals zero, then the statement has no effect.

Return value

None.

E.2.9 JNukeAttribute_isEmpty

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

Description

Query whether the data area associated with this attribute has one or more bytes. A call to this function is semantically equivalent to the expression `JNukeAttribute_getLength(this) == 0`.

Parameters

this
Reference to the attribute whose data area size should be examined.

Return value

This function returns `1` if the length of the data area equals zero. It returns `0` in all other cases.

E.2.10 JNukeAttribute_addSubAttribute

```
void
JNukeAttribute_addSubAttribute (
  JNukeObj * this,
  JNukeObj * sub
)
```

Description

Add a subattribute to an attribute.

Parameters

this
Reference to the attribute object who should act as a container for a new subattribute.

sub
Attribute object to be added as a subattribute.

Return value

None.

E.2.11 JNukeAttribute_getSubAttributes

```
JNukeObj *
JNukeAttribute_getSubAttributes (
  const JNukeObj * this
)
```

Description

Return a vector with all subattributes. If there are no subattributes associated with the attribute, then an empty vector will be returned.

Parameters

this
Attribute whose subattributes should be returned.

Return value

A reference to a `JNukeVector` containing the subattributes.

E.2.12 JNukeAttribute_countSubAttributes

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

Description

Returns the number of subattributes associated with this attribute. If the attribute has no subattributes, the return value is zero.

Parameters

`this`
Attribute whose subattributes should be counted.

Return value

The number of subattributes.

E.3 JNukeAttributeBuilder

The `JNukeAttributeBuilder` is an object capable of creating memory images of attributes as used in Java Class Files. The attributes are created using **JNuke** -internal objects, such as method or class descriptors, as templates. To name an example, `JNukeAttributeBuilder_createSourceFileAttribute` takes a class descriptor object as parameter and returns a new `JNukeAttribute` object containing a memory dump of a `SourceFile` attribute in its data area. The builder design pattern was adopted from [17, Chapter 3, p.97]

E.3.1 Constructor

```
JNukeObj *
JNukeAttributeBuilder_new(
  JNukeObj * mem
)
```

Description

The constructor is called to obtain a new object instance.

Parameters

mem
: A memory manager object of type `JNukeMem`. It is being used to allocate memory for the new object instance.

Return value

A pointer to a new instance of an attribute builder object.

E.3.2 JNukeAttributeBuilder_setConstantPool

```
void
JNukeAttributeBuilder_setConstantPool (
  JNukeObj * this,
  const JNukeObj * cp
)
```

Description

Set constant pool associated with the attribute builder. The new constant pool is expected to be a reference to an object of type `JNukeConstantPool`. When creating new attribute objects using the builder methods, the constant pool specified using this method will be extended with new entries.

The function `JNukeConstantPool_getConstantPool` may be used to get the current setting of this property.

Parameters

this
: Reference to the attribute builder object whose constant pool should be set.

cp
: Reference to the new `JNukeConstantPool`

Return value

None.

E.3.3 JNukeAttributeBuilder_getConstantPool

```
JNukeObj *
JNukeAttributeBuilder_getConstantPool (
  const JNukeObj * this
)
```

Description

Obtain constant pool object associated with this attribute builder. The constant pool may be set using the `JNukeConstantPool_setConstantPool` function.

Parameters

this
: Reference to an object of type `JNukeAttributeBuilder` whose current constant pool should be returned.

Return value

The constant pool object associated with the attribute builder, or `NULL` if the constant pool is not set.

E.3.4 JNukeAttributeBuilder_createSourceFileAttribute

```
JNukeObj *
JNukeAttributeBuilder_createSourceFileAttribute (
  JNukeObj * this,
  const JNukeObj * class
)
```

Description

Create a source file attribute, as described in [28, Section 4.7.7].

Parameters

this
: A `JNukeAttribute` object reference to the attribute builder that should create the attribute.

class
: Reference to an object of type `JNukeClass` to be used as a template.

Return value

Reference to a new `JNukeAttribute` object instance.

E.3.5 JNukeAttributeBuilder_createInnerClassesAttribute

```
JNukeObj *
JNukeAttributeBuilder_createInnerClassesAttribute (
  JNukeObj * this
)
```

Description

Create an attribute for inner classes, as described in [28, Section 4.7.5].

Parameters

this
A JNukeAttribute object reference to the attribute builder that should create the attribute

Return value

Reference to a new JNukeAttribute object instance.

E.3.6 JNukeAttributeBuilder_createSyntheticAttribute

```
JNukeObj *
JNukeAttributeBuilder_createSyntheticAttribute (
  JNukeObj * this
)
```

Description

Create a synthetic attribute, as described in [28, Section 4.7.6].

Parameters

this
A JNukeAttribute object reference to the attribute builder that should create the attribute.

Return value

Reference to a new JNukeAttribute object instance.

E.3.7 JNukeAttributeBuilder_createLineNumberTableAttribute

```
JNukeObj *
JNukeAttributeBuilder_createLineNumberTableAttribute(
  JNukeObj * this,
  const JNukeObj * method
)
```

Description

Create a line number table attribute, as described in [28, Section 4.7.8].

Parameters

this
A `JNukeAttribute` object reference to the attribute builder that should create the attribute.

method
Reference to an object of type `JNukeMethodDesc` to be used as a template.

Return value

Reference to a new `JNukeAttribute` object instance.

E.3.8 JNukeAttributeBuilder_createLocalVariableTable

```
JNukeObj *
JNukeAttributeBuilder_createLocalVariableTable(
  JNukeObj * this,
  const JNukeObj * method
)
```

Description

Create a local variable table attribute, as described in [28, Section 4.7.9].

Parameters

this
A `JNukeAttribute` object reference to the attribute builder that should create the attribute.

method
Reference to an object of type `JNukeMethodDesc` to be used as a template.

Return value

Reference to a new `JNukeAttribute` object instance.

E.3.9 JNukeAttributeBuilder_createDeprecatedAttribute

```
JNukeObj *
JNukeAttributeBuilder_createDeprecatedAttribute (
  JNukeObj * this
)
```

Description

Create a deprecated attribute, as described in [28, Section 4.7.10]. The same attribute is used for classes, methods, and fields, therefore no template is used.

Parameters

this
A `JNukeAttribute` object reference to the attribute builder that should create the attribute.

Return value

Reference to a new `JNukeAttribute` object instance.

E.3.10 JNukeAttributeBuilder_createExceptionsAttribute

```
JNukeObj *
JNukeAttributeBuilder_createExceptionsAttribute (
  JNukeObj * this,
  const JNukeObj * method
)
```

Description

Create an exceptions attribute, as described in [28, Section 4.7.4].

Parameters

this
A `JNukeAttribute` object reference to the attribute builder that should create the attribute.

method

Return value

Reference to a new `JNukeAttribute` object instance.

E.3.11 JNukeAttributeBuilder_createCodeAttribute

```
JNukeObj *
JNukeAttributeBuilder_createCodeAttribute (
  JNukeObj * this,
  const JNukeObj * method
)
```

Description

Create a code attribute, as described in [28, Section 4.7.3].

Parameters

this
A `JNukeAttribute` object reference to the attribute builder that should create the attribute.

method
Reference to an object of type `JNukeMethodDesc` to be used as a template.

Return value

Reference to a new `JNukeAttribute` object instance.

E.3.12 JNukeAttributeBuilder_createConstantValueAttribute

```
JNukeObj *
JNukeAttributeBuilder_createConstantValueAttribute (
  JNukeObj * this,
  const JNukeObj * field
)
```

Description

Create a constant value attribute, as described in [28, Section 4.7.2].

Parameters

this
: A `JNukeAttribute` object reference to the attribute builder that should create the attribute.

field
: Field descriptor of type `JNukeVar` to be used as template for the attribute

Return value

A new attribute object of type `JNukeAttribute`.

E.4 JNukeMFinfo

An object of type JNukeMFInfo is a shared container for both method_info [28, 4.6] and field_info [28, 4.5] regions of a class file. An object of type JNukeMFInfo stores zero or more attributes with a couple of integer fields.

E.4.1 Constructor

```
JNukeObj *
JNukeMFInfo_new (
  JNukeMem * mem
)
```

Description

The constructor is called to create a new instance of an object of type JNukeMFInfo. The memory required for creating the instance is drawn from the memory manager mem. Note that the JNukeMFInfoFactory object provides two methods to create new instances of type JNukeMFInfo using templates.

Parameters

mem
A reference to a memory manager object

Return value

A new instance of a JNukeMFInfo object.

E.4.2 JNukeMFInfo_getAccessFlags

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

Description

Get the current setting for the access flags property of a JNukeMFInfo object. Access flags are explained in [28, Table 4.5] for the method_info data structure, and in [28, Table 4.4] for the field_info data structure. The flags may be set by calling JNukeMFInfo_setAccessFlags.

Parameters

this
Reference to the JNukeMFInfo object whose access flags should be returned.

Return value

This function returns an integer value representing the current access flag setting of the info data structure.

E.4.3 JNukeMFInfo_setAccessFlags

```
void
JNukeMFInfo_setAccessFlags (
  JNukeObj * this,
  const int flags
)
```

Description

Set access flags of a `JNukeMFInfo` object. The current setting of the flags may be queried using the `JNukeMFInfo_getAccessFlags` function.

Parameters

this
Reference to a `JNukeMFInfo` object whose access flags should be set.

flags
The new value for the access flags. Individual flags can be added using logical OR.

Return value

None.

E.4.4 JNukeMFInfo_getNameIndex

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

Description

Get method or field name index of a `JNukeMFInfo` object. This property is an index into the constant pool, referencing an element of type `CONSTANT_UTF8` [28, 4.4.7]. The name index property may be set using the `JNukeMFInfo_setNameIndex` function.

Parameters

this
Reference to the object of type `JNukeMFInfo` whose name index property should be returned

Return value

The value of the name index property is returned.

E.4.5 JNukeMFInfo_setNameIndex

```
void
JNukeMFInfo_setNameIndex (
  JNukeObj * this,
  const int nameidx
)
```

Description

Set the value of the method or field name index property of a JNukeMFInfo object. The current value can be queried by using the JNukeMFInfo_getNameIndex function.

Parameters

this
Reference to the JNukeMFInfo object whose name index property should be set.

nameidx
The new value of the name index property.

Return value

None.

E.4.6 JNukeMFInfo_getDescriptorIndex

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

Description

Query the method or field descriptor index of a JNukeMFInfo object. The descriptor index property is an index into a constant pool entry of type CONSTANT_UTF8. The property may be set using the function JNukeMFInfo_setDescriptorIndex.

Parameters

this
Reference to the JNukeMFInfo object whose descriptor index property should be queried.

Return value

The value of the descriptor index property of the object.

E.4.7 JNukeMFInfo_setDescriptorIndex

```
void
JNukeMFInfo_setDescriptorIndex (
  JNukeObj * this,
  const int desc_idx
)
```

Description

Set method or field descriptor index property of a JNukeMFInfo object. The current value of the property may be queried using the JNukeMFInfo_getDescriptorIndex function.

Parameters

this

desc_idx
The new descriptor index. Note that the value must be greater than zero.

Return value

None.

E.4.8 JNukeMFInfo_getAttributes

```
JNukeObj *
JNukeMFInfo_getAttributes (
  const JNukeObj * this
)
```

Description

Get a vector with all attributes from the attribute table of this object. Each attribute entry in the vector is a reference to an object of type `JNukeAttribute`. If there are no attributes in the table, an empty vector with zero elements is returned. The function `JNukeMFInfo_addAttribute` may be used to add a new attribute to the object.

Parameters

this
Reference to an object of type `JNukeMFInfo` whose attribute table should be returned.

Return value

A reference to an object of type `JNukeVector` containing the attribute objects.

E.4.9 JNukeMFInfo_addAttribute

```
void
JNukeMFInfo_addAttribute (
  JNukeObj * this,
  const JNukeObj * attr
)
```

Description

Add an attribute to the attribute table of a `JNukeMFInfo` object. The new attribute is expected to be a reference to an object of type `JNukeAttribute`. The `JNukeAttributeFactory` offers methods to create standardized attributes. The order in which individual attributes are added to a table is important and will be retained. The function `JNukeMFInfo_getAttributes` returns a vector of all object attributes.

Parameters

`this`
Reference to the object of `JNukeMFInfo` to which the attribute should be added.

`attr`
A reference to the attribute object to be added

Return value

None.

E.5 JNukeMFInfoFactory

JNukeMFInfoFactory is a factory object to create JNukeMFInfo objects from method/field object templates. The factory uses a constant pool object and an attribute factor object to accomplish this task. It is primarily used by the class writer.

E.5.1 Constructor

```
JNukeObj *
JNukeMFInfoFactory_new (
  JNukeMem * mem
)
```

Description

Create a new instance of a JNukeMFInfoFactory object. The memory required for this operation is drawn from a memory manager object given as a parameter.

Parameters

mem
A reference to a memory manager object.

Return value

A new reference to an instance of a JNukeMFInfoFactory object.

E.5.2 JNukeMFInfoFactory_setConstantPool

```
void
JNukeMFInfoFactory_setConstantPool (
  JNukeObj * this,
  const JNukeObj * cp
)
```

Description

Set constant pool associated with this factory. The constant pool is normally used to insert new entries, such as the names of method_info or field_info attributes. The current constant pool object registered with this factory object can be queried using the JNukeMFInfoFactory_getConstantPool function.

Parameters

this
Reference to the factory object whose constant pool should be set.

cp
Reference to the constant pool object to be used in the future

Return value

None.

E.5.3 JNukeMFInfoFactory_getConstantPool

```
JNukeObj *
JNukeMFInfoFactory_getConstantPool (
  const JNukeObj * this
)
```

Description

Get constant pool associated with this factory object. To set the constant pool, the function `JNukeMFInfoFactory_setConstantPool` may be used.

Parameters

this
: Reference to the factory object whose constant pool should be returned.

Return value

A reference to the constant pool associated with this factory or `NULL` if there is no constant pool set.

E.5.4 JNukeMFInfoFactory_setAttributeFactory

```
void
JNukeMFInfoFactory_setAttributeFactory (
  JNukeObj * this,
  const JNukeObj * af
)
```

Description

Set the attribute factory property of the `JNukeMFInfoFactory`. The attribute factory is used to create new attributes, such as the `Code` attribute of a method as described in [28, 4.7.3]. The current attribute factory setting may be obtained using the `JNukeMFInfoFactory_getAttributeFactory` function.

Parameters

this
: Reference to the `JNukeMFInfoFactory` object whose attribute factory property should be returned.

af
: The reference to the new attribute factory to be used.

Return value

None.

E.5.5 JNukeMFInfoFactory_getAttributeFactory

```
JNukeObj *
JNukeMFInfoFactory_getAttributeFactory (
  JNukeObj * this
)
```

Description

Get attribute factory registered with this object. The attribute factory may be set using the `JNukeAttributeFactory_setAttributeFactory` function.

Parameters

this
Reference to the object whose attribute factory should be returned.

Return value

The attribute factory associated with this object or `NULL` if the attribute factory property is not set.

E.5.6 JNukeMFInfoFactory_createMethodInfo

```
JNukeObj *
JNukeMFInfoFactory_createMethodInfo(
  JNukeObj * this,
  const JNukeObj * method
)
```

Description

Create a `method_info` structure of a class file using the internal representation of a method as a template. This data structure is described in [28, Section 4.6].

Parameters

this
Reference to the `JNukeMFInfoFactory` object that should create the data structure.

method
Reference to an object of type `JNukeMethodDesc` that should act as a template

Return value

A reference to a new object of type `JNukeMFInfo` containing the desired `method_info` data structure.

E.5.7 JNukeMFInfoFactory_createFieldInfo

```
JNukeObj *
JNukeMFInfoFactory_createFieldInfo (
  JNukeObj * this,
  const JNukeObj * field
)
```

Description

Create a `field_info` structure of a class file using the internal representation of a variable as a template. This data structure is described in [28, Section 4.5].

Parameters

this

Reference to the `JNukeMFInfoFactory` factory object that should create the data structure.

field

Reference to an object of type `JNukeVarDesc` that should act as a template for the content of the data structure.

Return value

A reference to a new object of type `JNukeMFInfo` containing the desired `field_info` data structure.

E.6 JNukeClassWriter

`JNukeClassWriter` is an object capable of writing an internal memory representation of a class file to disk. Figure E.2 demonstrates how the object can be used in an application.

```
1  JNukeObj *writer, *classdesc;
2  int idx;
3
4  /* 1 - create a new class writer */
5  writer = JNukeClassWriter_new (mem);
6
7  /* 2 - Set descriptor of class to be written */
8  JNukeClassWriter_setDesc (writer, classdesc);
9
10 /* 3 - Modify the class */
11 idx = JNukeConstantPool_addInteger (writer, 8);
12 ...
13
14 /* 4 - Set output directory */
15 JNukeClassWriter_setOutputDirectory (writer, "/home/foo");
16
17 /* 5 - Set class file version */
18 JNukeClassWriter_setMajorVersion (writer, 45);
19 JNukeClassWriter_setMinorVersion (writer, 3);
20
21 /* 6 - Write the class */
22 if (!JNukeClassWriter_writeClass(writer)) {
23    printf("Error writing class file\n");
24 }
25
26 /* 7 - Dispose class writer */
27 JNukeObj_delete (writer);;
28
```

Figure E.2: Example usage of the `JNukeClassWriter` object

E.6.1 Constructor

```
JNukeObj *
JNukeClassWriter_new(
  JNukeMem * mem
)
```

Description

The constructor is called to obtain a new object instance.

Parameters

mem
: A memory manager object of type `JNukeMem`. It is being used to allocate memory for the new object instance.

Return value

A pointer to a new instance of a `JNukeClassWriter` object.

E.6.2 JNukeClassWriter_setClassDesc

```
void
JNukeClassWriter_setClassDesc (
  JNukeObj * this,
  const JNukeObj * desc
)
```

Description

Set class descriptor associated with the class writer. The class descriptor is the internal representation of the class file to be written. It is mandatory to set the class before calling `JNukeClassWriter_writeClass`. The class descriptor can be queried using the `JNukeClassWriter_getClassDesc` function.

Parameters

`this`
Reference to the class writer object

`desc`
New class descriptor.

Return value

None.

E.6.3 JNukeClassWriter_getClassDesc

```
JNukeObj *
JNukeClassWriter_getClassDesc (
  const JNukeObj * this
)
```

Description

Get class descriptor associated with the class writer. The class descriptor can be set using the `JNukeClassWriter_setClassDesc` function.

Parameters

`this`
Reference to the class writer object

Return value

A reference to the class descriptor object associated with the class writer or `NULL` if there is no class descriptor registered with this class writer.

E.6.4 JNukeClassWriter_setMinorVersion

```
void
JNukeClassWriter_setMinorVersion (
  JNukeObj * this,
  const int min
)
```

Description

Set minor version of the class file to be written. Handling of versions of class files is described in [28, 4.1]. This function must be called before `JNukeClassWriter_writeClass`. If a call to this function is omitted, reasonable values or the minor version will be substituted. The current setting for the minor version can be queried using the function `JNukeClassWriter_getMinorVersion`.

Parameters

`this`
Reference to the class writer object

`min`
New minor version number for the class to be written.

Return value

None.

E.6.5 JNukeClassWriter_getMinorVersion

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

Description

Query the setting for the minor version number. The current minor version can be set using function `JNukeClassWriter_setMinorVersion`.

Parameters

`this`
Reference to the class file writer object

Return value

The minor version of the class to be written. If no version was specified using the function `JNukeClassWriter_setMinorVersion`, the default minor version will be returned.

E.6.6 JNukeClassWriter_setMajorVersion

```
void
JNukeClassWriter_setMajorVersion (
  JNukeObj * this,
  const int maj
)
```

Description

Set major version of the class file to be written. Handling of versions of class files is described in [28, 4.1]. This function must be called before `JNukeClassWriter_writeClass`. If a call to this function is omitted, reasonable values for the major version will be substituted. The current setting for the major version can be queried using the `JNukeClassWriter_getMajorVersion` function.

Parameters

this
Reference to the class file writer object

maj
New major version for the class to be written.

Return value

None.

E.6.7 JNukeClassWriter_getMajorVersion

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

Description

Get major version number of the class to be written. The major version may be set using the function `JNukeClassWriter_setMajorVersion`.

Parameters

this
Reference to the class writer object

Return value

The major version of the class file to be written.

E.6.8 JNukeClassWriter_setOutputDir

```
void JNukeClassWriter_setOutputDir (
  JNukeObj * this,
  const char *dirname
)
```

Description

Set an alternate output directory for the class file to be written. The current setting for the output directory can be queried by using the `JNukeClassWriter_getOutputDir` function.

Parameters

this
Reference to the class writer whose output directory should be set.

dirname
The name of the output directory.

Return value

None.

E.6.9 JNukeClassWriter_getOutputDir

```
const char * JNukeClassWriter_getOutputDir
  const JNukeObj * this
)
```

Description

Query the current output directory of a class writer object. The output directory may be set by using the `JNukeClassWriter_setOutputDir` function.

Parameters

`this`
: Reference to the class writer object whose current output directory should be returned.

Return value

The current setting for the output directory property.

E.6.10 JNukeClassWriter_writeClass

```
int
JNukeClassWriter_writeClass (
  JNukeObj * this
)
```

Description

Write an internal memory representation of a class to a new Java .class file. The template for the class to be written must be set before calling this function using the method `JNukeClassWriter_setClassDesc`. The name of the new class file is automatically determined by the `ClassFile` property of the class descriptor.

Optionally, the minor and major version numbers of the class file may be set by calling `JNukeClassWriter_setMinorVersion` and `JNukeClassWriter_setMajorVersion` prior to calling this function.

The class file is usually written into the default output directory `instr`. If the class file should be written to another directory, it must be specified beforehand by calling the function `JNukeClassWriter_setOutputDir`.

Parameters

`this`
: Reference to the class writer object.

Return value

`1` if the writing was successful, `0` otherwise.

E.7 JNukeBCPatchMap

The `JNukeBCPatchMap` object provides a bidirectional byte code location transition map that is used by the instrumentation facility.

E.7.1 Constructor

```
JNukeObj *
JNukeBCPatchMap_new (
  JNukeObj * mem
)
```

Description

Create a new instance of a `JNukeBCPatchMap` object. The memory required for this operation is drawn from a memory manager object.

Parameters

mem
: Reference to a memory manager object.

Return value

A new instance of a `JNukeBCPatchMap` object.

E.7.2 JNukeBCPatchMap_moveInstruction

```
void
JNukeBCPatchMap_moveInstruction (
  JNukeObj * this,
  const int origfrom,
  const int origto
)
```

Description

Register the intention of moving a particular bytecode instruction.

Parameters

this
: Reference to the patch map object

origfrom
: Original bytecode address in the uninstrumented program that should be moved.

origto
: New target bytecode location in the address space of the uninstrumented program if the instruction would have been moved.

Return value

None.

E.7.3 JNukeBCPatchMap_commitMoves

```
void
JNukeBCPatchMap_commitMoves (
  JNukeObj * this
)
```

Description

Commit all registered moves. After a commit is done, the first phase of instrumentation is completed and no further intentions of moving an instruction may be registered with the transition map.

Parameters

`this`
: Reference to the patch map object

Return value

None.

E.7.4 JNukeBCPatchMap_translate

```
int
JNukeBCPatchMap_translate (
  const JNukeObj * this,
  const int origfrom
)
```

Description

Given a bytecode address in the uninstrumented program, return its corresponding new address when all previously registered changes would have been applied to the bytecode. This method may not be called on uncommitted transition maps.

Note that the function `JNukeBCPatchMap_translateReverse` performs the exact opposite address translation.

Parameters

`this`
: Reference to the patch map object

`origfrom`
: Original bytecode offset in the uninstrumented program whose final location in the instrumented program should be determined.

Return value

The new location of the instruction in the instrumented program.

E.7.5 JNukeBCPatchMap_translateReverse

```
int
JNukeBCPatchMap_translateReverse (
  const JNukeObj * this,
  const int newfrom,
  const JNukeObj * bc_vec
)
```

Description

Given the final location in an instrumented program, return its original address in the uninstrumented program. This function may not be executed on uncommitted transition maps.

Note that the function `JNukeBCPatchMap_translate` performs the exact opposite address translation.

Parameters

this
: Reference to the patch map object

newfrom
: Target location of the instruction in the instrumented program whose address should be translated back to the original uninstrumented program.

bc_vec
: Reference to a vector of bytecodes of the method. The vector is used for verification of the result.

Return value

E.8 JNukeBCPatch

JNukeBCPatch is an object capable of patching single Java bytecode instructions. It can also perform some advanced bytecode transformations, such as expanding a specific narrow bytecode instruction into its wide instruction format while semantically retaining its arguments. The instructions how bytecodes should be modified are taken from a JNukeBCPatchMap object that is registered with the object prior modifications are made.

E.8.1 Constructor

```
JNukeObj *
JNukeBCPatch_new (
  JNukeMem * mem
)
```

Description

Create a new instance of a JNukeBCPatch object. The memory required for the object instance is drawn from the memory manager mem.

Parameters

mem
Memory manager object

Return value

A reference to a newly created object of type JNukeBCPatch

E.8.2 JNukeBCPatch_getPatchMap

```
JNukeObj *
JNukeBCPatch_getPatchMap(
  const JNukeObj * this
)
```

Description

Get patch map associated with this object. The current patch map object may be set using the function JNukeBCPatch_setPatchMap.

Parameters

this
Reference to a patch object

E.8.3 JNukeBCPatch_setPatchMap

```
void
JNukeBCPatch_setPatchMap (
  JNukeObj * this,
  const JNukeObj * map
)
```

Description

Set patch map associated with this patch object. The patch map contains the rules how instructions should be modified during the patch process. The current patch map may be queried using the `JNukeBCPatch_getPatchMap` function.

Parameters

this
: Reference to a patch object

map
: A reference to the object that should be the new setting for the patch map property.

E.8.4 JNukeBCPatch_patchByteCode

```
int
JNukeBCPatch_patchBytCode (
  JNukeObj * this,
  JNukeObj * bc,
  JNukeObj * instrument
)
```

Description

Patch a bytecode object according to the patch map associated with this object. This method is capable of gracefully handling overflows in relative offsets of branch instructions, either by expanding narrow branch instructions to their wide format counterpart or by implementing a substitute by completely rewriting the bytecode. As a side effect, instructions may be therefore added or removed from the current method given by the instrumentation context.

Parameters

this
: Reference to the patch object

bc
: Bytecode instruction object to be patched

instrument
: In case of difficulties, the instrumentation context provided by this instrument is used to implement substitutes for overflows in relative offsets.

Return value

This method returns true if a substitute for offset overflow had to be inserted.

E.8.5 JNukeBCPatch_turnIf

```
int
JNukeBCPatch_turnIf (
  const int op
)
```

Description

Given a branch-related Java bytecode opcode that uses a relative offset, return its counterpart that does exactly the opposite. For example return the opcode `ifnonnull` for `ifnull`.

`opcode`
: Bytecode instruction opcode to be transformed

Return value

Opcode doing the opposite. If the instruction has no opposite command, the original opcode is returned.

E.9 JNukeInstrRule

A JNukeInstrRule object encapsulated a single instrumentation rule. A rule consists of a name and two functions eval and exec. Instrumentation rules are usually added to JNukeInstrument container objects.

E.9.1 Constructor

```
JNukeObj *
JNukeInstrRule_new (
  JNukeMem * mem
)
```

Description

The constructor is called to obtain a reference to a new object instance.

Parameters

mem
: A memory manager object of type JNukeMem. It is being used to allocate memory for the new object instance.

Return value

A pointer to a new instance of a JNukeInstrRule object.

E.9.2 JNukeInstrRule_setName

```
void
JNukeInstrRule_setName (
  JNukeObj * this,
  const char *name
)
```

Description

Set the name of the rule.

Parameters

this
: Reference to a rule object whose name should be set

name
: The name to be set. The reference must be valid, i.e not NULL.

Return value

None.

E.9.3 JNukeInstrRule_getName

```
JNukeObj *
JNukeInstrRule_getName (
  JNukeObj * this
)
```

Description

Get rule name.

Parameters

this
Reference to a rule object whose name should be returned.

Return value

Pointer to a string object with the name or `NULL` if the rule has no name.

E.9.4 JNukeInstrRule_setEvalMethod

```
void
JNukeInstrRule_setEvalMethod (
  JNukeObj * this,
  ir_evalfunc evalmeth
)
```

Description

Set location evaluation function connected with this rule.

Parameters

this
Pointer to the rule object whose eval method should be set.

evalmeth
A reference to the eval method. The method must have return type `int`. During instrumentation, exactly two arguments of type `JNukeObj *` will be passed to the method.

Return value

None.

E.9.5 JNukeInstrRule_getEvalMethod

```
ir_evalfunc
JNukeInstrRule_getEvalMethod (
  JNukeObj * this
)
```

Description

Get a reference to the location evaluation function.

Parameters

this
Reference to the rule whose location evaluation function should be queried.

Return value

A pointer to the rule whose location evaluation function or `NULL` if there is no eval method associated with this rule.

E.9.6 JNukeInstrRule_setExecMethod

```
void
JNukeInstrRule_setExecMethod (
  JNukeObj * this,
  ir_execfunc execmeth
)
```

Description

Set instrumenting function.

Parameters

this
Reference to an instrumentation rule object.

execmeth
Pointer to the instrumenting function. The function may not return values, i.e., must have return type `void`. During instrumentation, exactly one parameter is passed, so it must accept a reference of type `JNukeObj *` with the instrumentation context as argument.

Return value

None.

E.9.7 JNukeInstrRule_getExecMethod

```
ir_execfunc
JNukeInstrRule_getExecMethod (
  JNukeObj * this
)
```

Description

Obtain a pointer to the instrumenting function.

Parameters

this
Reference to an instrumentation rule object.

Return value

A pointer to the instrumenting function.

E.9.8 JNukeInstrRule_eval

```
int
JNukeInstrRule_eval (
  JNukeObj * this,
  JNukeObj * instr
  JNukeObj * data
)
```

Description

Perform an explicit call to the location evaluation function associated with the rule.

Parameters

this
: The rule whose eval function should be executed.

instr
: An instrumentation context to be passed to the function as first argument.

data
: Additional data who should be passed to the function as second argument. For most rules, a `NULL` value is expected, but the meaning of this parameter depends on the function.

Return value

This value returns a boolean value with the result of the function call.

E.9.9 JNukeInstrRule_execute

```
void JNukeInstrRule_execute (
  JNukeObj * this,
  JNukeObj * instr
)
```

Description

Perform an explicit call to the instrumenting function connected with the rule.

Parameters

this
: The rule whose exec function should be executed.

instr
: An instrumentation context to be passed to the function as first and only argument.

Return value

None.

E.10 JNukeInstrument

The `JNukeInstrument` is the primary object of the instrumentation facility. It acts as a container for instrumentation rules, i.e., objects of type `JNukeInstrRule`. Instrumentation is done by iterating over all methods of a class. A valid class descriptor must therefore be set before starting the instrumentation process. As the instrumentation rules may modify the constant pool of a class, a `JNukeConstantPool` object must be registered with the instrumentation facility as well.

Figure E.3 may help to understand how to use the instrumentation facility.

```
1   JNukeObj * instr, *rule;
2   /* set up instrumentation facility */
3   instr = JNukeInstrument_new (mem);
4   JNukeInstrument_setClassDesc(instr, classdesc);
5   JNukeInstrument_setConstantPool(instr, cp);
6
7   /* create one or more rules and add them to the facility */
8   rule = JNukeInstrRule_new (mem);
9   ...
10  JNukeInstrument_addRule(instr, rule);
11
12  /* Instrument the class */
13  JNukeInstrument_instrument(instr);
14
15  /* Finally dispose facility */
16  JNukeObj_delete(instrument);
```

Figure E.3: Example program demonstrating the use of the instrumentation facility

E.10.1 Constructor

```
JNukeObj *
JNukeInstrument_new(
  JNukeObj * mem
)
```

Description

The constructor is called to obtain a new object instance.

Parameters

mem
: A memory manager object of type `JNukeMem`. It is being used to allocate memory for the new object instance.

Return value

A pointer to a new instance of a `JNukeInstrument` object.

E.10.2 JNukeInstrument_setClassDesc

```
void
JNukeInstrument_setClassDesc (
  JNukeObj * this,
  JNukeObj * cld
)
```

Description

Set class descriptor associated with the instrumentation facility. The class descriptor specifies the class to be instrumented. The current setting for this property can be queried by calling `JNukeInstrument_getClassDesc`.

Parameters

`this`
Reference to the instrument object whose class descriptor should be set.

`cld`
New setting for the class descriptor.

Return value

None.

E.10.3 JNukeInstrument_getClassDesc

```
JNukeObj *
JNukeInstrument_getClassDesc (
  JNukeObj * this
)
```

Description

Get class descriptor associated with the instrumentation facility. If no class descriptor was previously set, the method returns `NULL`. This method is intended to be used by instrumentation rules to determine the current instrumentation context. The setting of this property may be altered by using the `JNukeInstrument_setClassDesc` function.

Parameters

`this`
Reference to the instrument object

Return value

A reference to an object of type `JNukeClassDesc`.

E.10.4 JNukeInstrument_setWorkingMethod

```
void
JNukeInstrument_setWorkingMethod (
  JNukeObj * this,
  JNukeObj * methdesc
)
```

Description

Set current method the instrumentation facility should work on. This is usually automatically set during the instrumentation process. This method is probably only used in test cases, but hypothetically, an instrumentation rule could alter the working method by calling this method. The current working method can be queried using the function `JNukeInstrument_getWorkingMethod`.

Parameters

this
Reference to the instrument object whose method descriptor should be set.

methdesc
Reference to the new Method descriptor.

Return value

None.

E.10.5 JNukeInstrument_getWorkingMethod

```
JNukeObj *
JNukeInstrument_getWorkingMethod (
  JNukeObj * this
)
```

Description

Get current working method, i.e., the method the instrumentation facility is currently working on. The current working method can be set by calling `JNukeInstrument_setWorkingMethod`.

Parameters

this
Reference to the instrument object

Return value

Reference to a class descriptor for the current working method or `NULL` if there is no current working method.

E.10.6 JNukeInstrument_getWorkingByteCode

```
JNukeObj *
JNukeInstrument_getWorkingByteCode (
  JNukeObj * this
)
```

Description

Get current working bytecode, i.e., the bytecode that is currently being instrumented. This method is intended for the use by instrumentation rules to determine whether and how the instruction should be instrumented. The current working bytecode can be set using `JNukeInstrument_setWorkingByteCode`.

Parameters

this
Reference to the instrument object

Return value

Reference to the current working bytecode object or `NULL` if there is no working bytecode associated with the instrumentation facility.

E.10.7 JNukeInstrument_setWorkingByteCode

```
void
JNukeInstrument_setWorkingByteCode (
  JNukeObj * this,
  const JNukeObj * bc
)
```

Description

Force current working bytecode. During instrumentation, the current working bytecode is usually automatically set. Overriding the working byte can have undesired side effects. The use of this method is therefore not recommended. Use the method `JNukeInstrument_getWorkingByteCode` to query the current working bytecode object.

Parameters

this
Reference to the instrument object, whose working bytecode should be set.

bc
Reference to the new working bytecode.

Return value

None.

E.10.8 JNukeInstrument_setConstantPool

```
void
JNukeInstrument_setConstantPool (
  JNukeObj * this,
  const JNukeObj * constantPool
)
```

Description

Set constant pool associated with the instrumentation facility. The current setting for this property may be queried using the `JNukeInstrument_getConstantPool` function. This function should be called at most once for every instrumentation run, as upon invocation of this method, previous changes to the old constant pool will be lost (unless they are saved otherwise).

Parameters

this
Reference to the instrument object whose constant pool property should be changed.

constantPool
Reference to the new constant pool.

Return value

None.

E.10.9 JNukeInstrument_getConstantPool

```
JNukeObj *
JNukeInstrument_getConstantPool (
  JNukeObj * this
)
```

Description

Obtain a reference to the constant pool object associated with the instrumentation facility. This method is intended for instrumentation rules to get entries from the constant pool and/or to alter the constant pool by adding new entries to it. The `JNukeInstrument_setConstantPool` function can be used to manually set a new constant pool.

Parameters

`this`
Reference to the instrument object

Return value

Reference to the constant pool object or `NULL` if there is no constant pool object associated with the instrumentation facility.

E.10.10 JNukeInstrument_addRule

```
void
JNukeInstrument_addRule (
  JNukeObj * this,
  JNukeObj * rule
)
```

Description

Add an instrumentation rule. Note that the rule is not executed unless the instrumentation is started with the `JNukeInstrument_instrument` method. During instrumentation, rules are invoked in the same order they were previously added to the instrument object.

Parameters

`this`
Reference to the instrument object

`rule`
Reference to the rule object to be appended.

Return value

None.

E.10.11 JNukeInstrument_getRules

```
JNukeObj *
JNukeInstrument_getRules (
  JNukeObj * this
)
```

Description

Get a vector of instrumentation rules associated with the instrumentation facility. If there are no instrumentation rules, then an empty vector is returned.

Parameters

this
Reference to the instrumentation facility whose vector of rules should be returned.

Return value

A reference to a vector containing the instrumentation rules.

E.10.12 JNukeInstrument_clearRules

```
void
JNukeInstrument_clearRules (
  JNukeObj * this
)
```

Description

Clear all rules associated with the instrumentation facility. If there were no rules, the statement has no effect.

Parameters

this
Reference to the instrumentation facility whose rules should be cleared.

Return value

None.

E.10.13 JNukeInstrument_insertByteCode

```
void
JNukeInstrument_insertByteCode (
  JNukeObj * this,
  JNukeObj * bc,
  const int offset
)
```

Description

Register the intention of inserting a bytecode instruction into the bytecode stream at a given offset. The instruction will not be inserted unless the method
`JNukeInstrument_instrument` is called. This method is intended to be used by instrumentation rules to modify the byte code stream. Use the method `JNukeInstrument_removeByteCode` to remove a bytecode instruction from the stream.

Parameters

this
Reference to an instrumentation facility object.

bc
Reference to the bytecode instruction object to be inserted.

offset
Offset in the uninstrumented byte stream where the instruction should be inserted. This value may not be negative.

Return value

None.

E.10.14 JNukeInstrument_removeByteCode

```
void
JNukeInstrument_removeByteCode (
  JNukeObj * this,
  const int offset
)
```

Description

Register the intention of removing a particular bytecode instruction at a given offset from the bytecode stream. Instructions will not be removed unless the method `JNukeInstrument_instrument` is called. This method is intended to be used by instrumentation rules to modify the bytecode stream. Use the method `JNukeInstrument_insertByteCode` to insert a byte code instruction into the stream.

Parameters

this
Reference to the instrumentation facility

offset
Offset of the instruction to be removed.

Return value

None.

E.10.15 JNukeInstrument_instrument

```
void
JNukeInstrument_instrument (
  JNukeObj * this
)
```

Description

Start the actual instrumentation process.

The following conditions must be met prior to executing this function:

1. A class must be associated with the facility by calling `JNukeInstrument_setClassDesc`

2. A constant pool must be registered with the facility by executing `JNukeInstrument_setConstantPool`
3. It is highly suggested, that one or more instrumentation rules are added using the function `JNukeInstrument_addRule`, otherwise this statement will have no effect.

Parameters

`this`
Reference to the instrument object.

Return value

None.

A call to this method usually alters the bytecode in the class associated with the instrumentation facility. It may also modify the registered rules.

E.11 JNukeEvalFactory

All implemented location evaluation functions used in bytecode suitable to be used for instrumentation rules have been combined into a collection object called `JNukeEvalFactory`. Figure E.4 shows a list of all implemented functions. To obtain a particular implementation of an location evaluation function, the factory method `JNukeEvalFactory_get` may be used. Individual functions are selected by using an integer constant with a descriptive name.

`JNukeEvalFactory` is a static object and therefore does not require a constructor.

Name	Parameter
eval_everywhere	—
eval_nowhere	—
eval_atInstruction	opcode
eval_atReturnInstruction	—
eval_atInvokeInstruction	—
eval_atOffset	offset
eval_atBeginOfMethod	—
eval_atEndOfMethod	—
eval_atNewRunnable	—
eval_atThreadInterrupt	—
eval_atObjectWait	—
eval_atObjectNotify	—
eval_matchesThreadChange	ThreadChange
eval_inThreadChangeVector	vector
eval_inMethodNamed	MethodName
eval_inRunMethod	—
eval_atBeginOfMainMethod	—
eval_atBeginOfRunMethod	—
eval_atReturnInRunMethod	—
eval_atClassForName	—

Figure E.4: Predefined factory location evaluation methods

E.11.1 JNukeEvalFactory_get

```
ir_evalfunc
JNukeEvalFactory_get (
  evalFactoryMethod idx
)
```

Description

Return a pointer to a particular implementation from the collection of implemented location evaluation function.

Parameters

`idx`
Symbolic constant to access the location evaluation function.

Return value

A pointer to the factory method or `NULL` if the index was invalid. The returned pointer will reference a function that has the following signature:

```
int
ir_evalfunc (
  JNukeObj * instrument,
  JNukeObj * data
)
```

E.12 JNukeExecFactory

All implemented functions performing actual bytecode instrumentation have been combined into a collection object called `JNukeExecFactory`. Figure E.5 shows a list of all available functions.

To obtain a particular implementation of an instrumentation function, the function factory can be queried using the `JNukeExecFactory_get` method. Individual functions are selected by using an integer constant with a descriptive name.

`JNukeExecFactory` is a static object and therefore does not require a constructor.

Name
exec_doNothing
exec_replay_init
exec_replay_sync
exec_replay_registerThreadForNew
exec_replay_blockThis
exec_replay_objectWaitWorkaroundPre
exec_replay_objectWaitWorkaroundPost
exec_replay_objectNotifyWorkaroundPre
exec_replay_objectNotifyWorkaroundPost
exec_replay_threadInterruptWorkaround
exec_replay_terminate
exec_removeInstruction

Figure E.5: Predefined factory instrumentation methods

E.12.1 JNukeExecFactory_get

```
ir_execfunc
JNukeExecFactory_get (
  execFactoryMethod idx
)
```

Description

Return a pointer to a particular implementation from the collection of implemented instrumentation function.

Parameters

`idx`
Symbolic constant to access the instrumentation function.

Return value

A pointer to the factory method or `NULL` if the index into the function collection was invalid. The returned pointer will reference a function that has the following signature:

```
void
ir_execfunc (
  JNukeObj * instrument
)
```

E.13 JNukeTaggedSet

The **JNuke** framework was extended with a new `JNukeTaggedSet` object. This object acts as a container for objects, much like the normal `JNukeSet`, except that a tag bit is associated with each individual set item. Unlike `JNukeSet`, an item may only be added once to the set.

E.13.1 Constructor

```
JNukeObj *
JNukeTaggedSet_new (
  JNukeMem * mem
)
```

Description

The constructor is called to obtain a new reference to an object instance of type `JNukeTaggedSet`. The memory required for this operation is drawn from a memory manager object.

Parameters

`mem`
: Reference to a memory manager object.

Return value

Reference to a new instance of a `JNukeTaggedSet` object.

E.13.2 JNukeTaggedSet_setType

```
void
JNukeTaggedSet_setType (
  JNukeObj * this,
  const JNukeContent type
)
```

Description

Set content type of the items stored in the set.

Parameters

`this`
: Reference a tagged set object.

`type`
: New content type of the set items.

Return value

None.

E.13.3 JNukeTaggedSet_tagObject

```
int
JNukeTaggedSet_tagObject (
  JNukeObj * this,
  const JNukeObj * obj
)
```

Description

Depending on whether the object is already in the set and whether its tag bit is set, the function behaves as follows:

- If the object is not in the set, it is added to the set and its tag bit is set.
- If the object is already in the set but untagged, its tag bit is set.
- If the object is already in the set and tagged, calling this method has no effect.

Parameters

`this`
Reference to a tagged set object.

`obj`
Reference to an object to be added or modified.

Return value

This method always returns 1.

E.13.4 JNukeTaggedSet_untagObject

```
int
JNukeTaggedSet_untagObject (
  JNukeObj * this,
  const JNukeObj * obj
)
```

Description

Depending whether the object is already in the set and whether its tag bit is set, the function behaves as follows:

- If the object is not in the set, it is added to the set and its tag bit is cleared.
- If the object is already in the set but tagged, its tag bit is cleared.
- If the object is already in the set and its tag bit is not set, calling this method has no effect.

Parameters

`obj`
Reference to a tagged set object.

`obj`
Reference to an object to be added or modified.

Return value

This method always returns 1.

E.13.5 JNukeTaggedSet_isTagged

```
int
JNukeTaggedSet_isTagged (
  JNukeObj * this,
  const JNukeObj * obj
)
```

Description

Check whether an object is both in the set and tagged. If the object is not in the set, the value 0 is returned. If the object is in the set but its tag bit is cleared, the value 0 is returned, too.

Parameters

this
: Reference to a tagged set object.

obj
: Reference to the object whose tag bit should be examined.

Return value

The method returns 1 if the object is both in the set and its tag bit set. In all other cases, the value 0 is returned.

E.13.6 JNukeTaggedSet_isUntagged

```
int
JNukeTaggedSet_isUntagged (
  JNukeObj * this,
  const JNukeObj * obj
)
```

Description

Check whether an object is both in the set and not tagged. If the object is not in the set, the value 0 is returned. If the object is in the set but its tag bit is set, the value 0 is returned.

Parameters

this
: Reference to a tagged set object.

obj
: Reference to the object whose tag bit should be examined.

Return value

The method returns 1 if the object is both in the set and its tag bit is cleared. In all other cases, the value 0 is returned.

E.13.7 JNukeTaggedSet_getTagged

```
JNukeObj *
JNukeTaggedSet_getTagged (
  JNukeObj * this
)
```

Description

Get the subset of objects in the tagged set whose tag bit is set.

Parameters

this
Reference to a tagged set object.

Return value

This method returns a `JNukeSet` container with all elements of the tagged set whose tag bit is set. If the tagged is empty or only contains elements whose tag bit is cleared, an empty set will be returned.

E.13.8 JNukeTaggedSet_getUntagged

```
JNukeObj *
JNukeTaggedSet_getUntagged (
  JNukeObj * this
)
```

Description

Get the subset of objects in the tagged set whose tag bit is cleared.

Parameters

this
Reference to a tagged set object.

Return value

This method returns a `JNukeSet` container with all elements of the tagged set whose tag bit is cleared. If the tagged is empty or only contains elements whose tag bit is set, an empty set will be returned.

E.13.9 JNukeTaggedSet_clear

```
void
JNukeTaggedSet_clear (
  JNukeObj * this
)
```

Description

Remove all elements from the tagged set, independent of whether their tag bit is set or cleared.

Parameters

this
: Reference to a tagged set object to be cleared.

Return value

None.

E.13.10 JNukeTaggedSet_contains

```
int
JNukeTaggedSet_contains (
  JNukeObj * this,
  const JNukeObj * obj
)
```

Description

This method may be used to determine whether a particular object is in the tagged set. The tag bit of the object is disregarded.

Parameters

this
: Reference to a tagged set object.

obj
: Reference to the object.

Return value

This method returns 1 if the object is part of the set. If the object is not in the set, the value 0 is returned.

E.14 JNukeThreadChange

The `JNukeThreadChange` is the internal representation of the necessary conditions for a thread change within a replay schedule. A vector of `JNukeThreadChange` objects can be obtained by using the `JNukeScheduleFileParser` object.

E.14.1 Constructor

```
JNukeObj *
JNukeThreadChange_new (
  JNukeObj * mem
)
```

Description

The constructor is called to obtain a new instance of a `JNukeThreadChange` object. The memory required is drawn from a memory manager.

Parameters

`mem`
: Reference to a memory manager object.

Return value

This function returns a new instance of a `JNukeThreadChange` object.

E.14.2 JNukeThreadChange_set

```
void
JNukeThreadChange_set (
  JNukeObj * this,
  const unsigned int thread,
  const unsigned int method,
  const unsigned int ofs,
  const unsigned int iter
)
```

Description

Set integer properties of a thread change object.

Parameters

`this`
: Reference to the thread change object.

`thread`
: New value for the thread identifier.

`method`
: New value for the method index.

`ofs`
: New value for the bytecode offset.

`iter`
: New value for the iteration index.

Return value

None.

E.14.3 JNukeThreadChange_setClassName

```
void
JNukeThreadChange_setClassName (
  JNukeObj * this,
  const char *className
)
```

Description

Set class name property of a thread change object. To query this property, the functions `JNukeThreadChange_getClassName` and `JNukeThreadChange_getClassNameAsObject` may be used.

Parameters

`this`
Reference to a thread change object.

`className`
New value for the class name property.

Return value

None.

E.14.4 JNukeThreadChange_getClassName

```
const char *
JNukeThreadChange_getClassName (
  const JNukeObj * this
)
```

Description

Get class name property value of a thread change object. The value is returned as a `char*`. The method `JNukeThreadChange_getClassNameAsObject` may be used to obtain the current value as a `JNukeObj*`.

Parameters

`this`
Reference to the thread change object.

Return value

This function returns the current setting for the class name property.

E.14.5 JNukeThreadChange_getClassNameAsObject

```
JNukeObj *
JNukeThreadChange_getClassNameAsObject (
  const JNukeObj * this
)
```

Description

Get class name property value of a thread change object. The value is returned as a `JNukeObj*`. The function `JNukeThreadChange_getClassName` may be used to obtain the current value as a `char*`.

Parameters

`this`
Reference to a thread change object.

Return value

E.14.6 JNukeThreadChange_setCommand

```
void
JNukeThreadChange_setCommand (
  JNukeObj * this,
  const int command
)
```

Description

Set the command property of a thread change. The function `JNukeThreadChange_getCommand` may be used to query this property.

Parameters

`this`
Reference to a thread change object.

`command`
New value for the command property. Currently only the symbolic value `cmd_switch` is allowed.

Return value

None.

E.14.7 JNukeThreadChange_getCommand

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

Description

Get current command property value of a thread change object. The function `JNukeThreadChange_setCommand` may be used to set this property.

Parameters

this
Reference to a thread change object.

Return value

The current value for the command property.

E.14.8 JNukeThreadChange_setThreadIndex

```
void
JNukeThreadChange_setThreadIndex (
  JNukeObj * this,
  const int idx
)
```

Description

Set the thread index property of a thread change object. The function `JNukeThreadChange_getThreadIndex` may be used to query this property.

Parameters

this
Reference to a thread change object.

idx
New value for the thread index property. Negative values are not allowed.

Return value

None.

E.14.9 JNukeThreadChange_getThreadIndex

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

Description

Get current value of the thread index property of a thread change object. The function `JNukeThreadChange_setThreadIndex` may be used to query this property.

Parameters

this
Reference to a thread change object.

Return value

This function returns the current setting for the thread index property of a thread change object.

E.14.10 JNukeThreadChange_setMethodIndex

```
void
JNukeThreadChange_setMethodIndex (
  JNukeObj * this,
  const int idx
)
```

Description

Set method index property of a thread change object. The function `JNukeThreadChange_getMethodIndex` may be used to query this property.

Parameters

this
: Reference to a thread change object.

idx
: New value for the method index object. The value may not be negative.

Return value

None.

E.14.11 JNukeThreadChange_getMethodIndex

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

Description

Get current method index value of a thread change object. The function `JNukeThreadChange_setMethodIndex` may be used to set this property.

Parameters

this
: Reference to a thread change object.

Return value

This function returns the current value for the method index property.

E.14.12 JNukeThreadChange_setByteCodeOffset

```
void
JNukeThreadChange_setByteCodeOffset (
  JNukeObj * this,
  const int newofs
)
```

Description

Set bytecode offset of a thread change object. The function `JNukeThreadChange_getByteCodeOffset` may be used to query this property.

Parameters

`this`
Reference to a thread change object.

`newofs`
New value for the bytecode offset property. The number may not be negative.

Return value

None.

E.14.13 JNukeThreadChange_getByteCodeOffset

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

Description

Get current value of the bytecode offset property of a thread change object. The function `JNukeThreadChange_setByteCodeOffset` may be used to set this property.

Parameters

`this`
Reference to a thread change object.

Return value

This function returns the current setting for the bytecode offset property.

E.14.14 JNukeThreadChange_setIterationIndex

```
void
JNukeThreadChange_setIterationIndex (
  JNukeObj * this,
  const int index
)
```

Description

Set iteration index of a thread change object. The function `JNukeThreadChange_getIterationIndex` may be used to query this property.

Parameters

`this`
Reference to a thread change object.

`index`
New value for the iteration index property. The value must be greater than zero.

Return value

None.

E.14.15 JNukeThreadChange_getIterationIndex

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

Description

Get iteration index property of a thread change object. The function `JNukeThreadChange_setIterationIndex` may be used to set this property.

Parameters

`this`
Reference to a thread change object.

Return value

This function returns the current setting for the iteration index property.

E.14.16 JNukeThreadChange_parse

```
int
JNukeThreadChange_parse (
  JNukeObj * this,
  const char *line
)
```

Description

Parse a single line of a thread replay schedule file into a thread change object. The contents of the line must follow the specification given in Appendix B.

Parameters

`this`
Reference to a thread change object.

`line`
Line to be parsed.

Return value

E.15 JNukeScheduleFileParser

The `JNukeScheduleFileParser` is an object that can read a thread replay schedule given in a text file. The file must be valid according to the specification given in Appendix B. See Figure E.6 for an example how to use the parser.

```
1  JNukeObj *parser, *vec;
2
3  parser = JNukeScheduleFileParser_new (mem);
4  vec = JNukeScheduleFileParser_parse (parser, "filename.cx");
5  JNukeObj_delete (parser);
6  if (vec == NULL) {
7      printf("Error: Unable to thread thread schedule from file\n");
8  }
```

Figure E.6: Example demonstrating the usage of the `JNukeScheduleFileParser` object

E.15.1 Constructor

```
JNukeObj *
JNukeScheduleFileParser_new(
  JNukeMem * mem
)
```

Description

The constructor is called to obtain a new object instance.

Parameters

mem
: A memory manager object of type `JNukeMem`. It is being used to allocate memory for the new object instance.

Return value

A pointer to a new instance of a `JNukeScheduleFileParser` object.

E.15.2 JNukeScheduleFileParser_parse

```
JNukeObj *
JNukeScheduleFileParser_parse(
  JNukeObj * this,
  const char * filename
)
```

Description

Read a text file containing an execution trace into memory.

Parameters

this
: Reference to a schedule file parser object.

filename
: Name of the file to be read. The file must exist and it may not be empty.

Return value

This function returns a vector of `JNukeThreadChange` objects. In case of an error, a `NULL` pointer is returned.

E.15.3 JNukeScheduleFileParser_getClassNames

```
JNukeObj *
JNukeScheduleFileParser_getClassNames (
  const JNukeObj * this
)
```

Description

Return a tagged set with the names of the classes being referenced in the thread schedule. Prior to calling this function, the method `JNukeScheduleFileParser_parse` has to be executed, otherwise the set will be empty.

Parameters

`this`
Reference to a schedule file parser object.

Return value

A `JNukeTaggedSet` with the names of the classes referenced in the thread schedule file.

E.16 JNukeReplayFacility

To allow the automated testing of the replay facility, the code of the **jreplay** application has been out-sourced into a single object. `JNukeReplayFacility` is a static object, i.e., it does not require a constructor.

E.16.1 JNukeReplayFacility_prepareInstrument

```
void
JNukeReplayFacility_prepareInstrument (
  JNukeObj * instrument,
  JNukeObj * tcvec,
  JNukeObj * classNames
)
```

Description

Add the required bytecode instrumentation rules described in chapter 6 to a single instrument object that can later be used in the **jreplay** application.

Parameters

`instrument`
: Reference to the instrumentation facility

`tcvec`
: Reference to a `JNukeVector` object storing the `JNukeThreadChange` objects that contains the thread replay schedule. Such a vector is returned by the `JNukeScheduleFileParser_parse` method.

`classNames`
: Reference to a `JNukeTaggedSet` with the names of the classes to be instrumented. Such a set is returned by the `JNukeScheduleFileParser_getClassName` function.

Return value

None.

E.16.2 JNukeReplayFacility_executeCX

```
int
JNukeReplayFacility_executeCX (
  JNukeMem * mem,
  const char *cxfilename,
  const char *initclass,
  const char *outdir
)
```

Description

Internalize a thread replay schedule from a file and write instrumented versions of the class files mentioned in the file to a specified output directory. This method builds the very core of the **jreplay** application.

Parameters

`mem`
: Reference to a memory manager object

`cxfilename`
: Pointer to the name of the file containing the thread schedule to be replayed.

`initclass`
: Optional pointer to the name of the initial class to be instrumented. The initial class is the class that contains the `main` method of the application. If the class is mentioned in the thread replay schedule file, this parameter may be omitted by passing a `NULL` value.

`outdir`
: Optional pointer specifying an alternate name of the output directory where the instrumented class files will be stored.

Return value

This function returns `1` if all classes could be successfully instrumented. It returns `0` in all other cases.

List of Figures

2.1 Structure of a Java class file . . . 7

2.2 Storing hierarchical information about a method in the constant pool of a class file . . . 8

2.3 Standardized attributes of a Java class file . . . 8

4.1 Thread switch . . . 17

5.1 Control flow in a successful transfer from thread t_1 to t_2 . . . 21

5.2 Control transfer from Thread t_1 to Thread t_2 intercepted by the JVM scheduler . . . 22

5.3 Implementation of a method to block a thread . . . 22

5.4 Implementation of a method to unblock a thread . . . 23

5.5 Three ways to call Object.wait . . . 25

5.6 Equivalent transformation of `wait` at the source level . . . 25

5.7 Extended version of the `block` method . . . 26

6.1 Bytecode transformations of `main` method to initialize the replayer . . . 28

6.2 Bytecode transformation to force an explicit thread switch . . . 29

6.3 Pseudo-code of the `replay.sync` transfer method . . . 29

6.4 Bytecode transformations to register a new thread object . . . 30

6.5 Bytecode transformations of `run` methods to retain the replay invariant for newly created threads . . . 31

6.6 Bytecode transformation to release a terminating thread . . . 32

6.7 Bytecode transformation to instrument `Object.wait` . . . 32

6.8 Bytecode transformation to instrument `Object.notify` . . . 33

6.9 Bytecode transformation to instrument `Thread.interrupt` . . . 33

6.10 Bytecode transformation to instrument `Thread.join` . . . 34

6.11 Summary of instrumented bytecode transformations . . . 34

7.1 Layout of the **JNuke** instrumentation facility . . . 35

7.2 Pseudo-code for instrumentation of a class in memory 36
7.3 Instrumenting the `closedloop` example program 37
7.4 Insertion wait list for the `closedloop` example program 38
7.5 Transition map for the `closedloop` example program 38
7.6 Example bytecode with a large absolute branch offset and instrumented substitute 39
7.7 Example bytecode with a large relative branch offset 40

8.1 Loading Java class files in **JNuke** 42
8.2 Objects involved in writing new Java class files 43
8.3 Attribute objects 44
8.4 Attributes in a `JNukeMFInfo` object for a `method_info` item 45

9.1 Virtual machines used to test the experiments 49
9.2 Measured execution times for the `closedloop` example program 50
9.3 Results of the simple experiments 52

10.1 Wrapper method to prevent deadlocks during replay due to uncaught exceptions in an application 54

A.1 Overview of the **jreplay** application layout 57

B.1 Syntax of a single line defining a thread change in the replay schedule file . 60
B.2 Example thread schedule file 61

C.1 Programming interface to the replayer 63

D.1 Example program to measure the overhead of instrumented explicit thread changes 65
D.2 Thread replay schedule for example `closedloop` 65
D.3 Source code of the r1 example program 66
D.4 Thread replay schedule A for example `r1` 66
D.5 Thread replay schedule B for example `r1` 67
D.6 Thread replay schedule A for the `deadlock` example 67
D.7 Thread replay schedule B for the `deadlock` example 67
D.8 Source code of the deadlock example program 68
D.9 Source code of the `waitnotify` example program 69
D.10 Thread replay schedule A for the `waitnotify` example 70
D.11 Thread replay schedule B for the `waitnotify` example 70
D.12 Thread replay schedule C for the `waitnotify` example 70

D.13 Example program for suspend resume 71
D.14 Thread replay schedule for `suspresume` 72

E.1 UML diagram of **JNuke** objects relevant for this report 74
E.2 Example usage of the `JNukeClassWriter` object 105
E.3 Example program demonstrating the use of the instrumentation facility . . . 120
E.4 Predefined factory location evaluation methods 128
E.5 Predefined factory instrumentation methods 129
E.6 Example demonstrating the usage of the `JNukeScheduleFileParser` object 142

Bibliography

[1] C. Artho. *Finding faults in multi-threaded programs*. Diploma Thesis. Institute of Computer Systems, ETH Zentrum, RZ H15, CH-8092 Zürich, March 2001.
`http://www.inf.ethz.ch/personal/artho/mthesis.pdf` .

[2] M. J. Bach. *The Design of the Unix Operating System*. Prentice-Hall, 1986. ISBN 0-13-201757-1.

[3] B. Bokoswki. *Barat - A Front-End for Java*. Freie Universität Berlin, December 1998. Technical Report
`ftp://ftp.inf.fu-berlin.de/pub/reports/tr-b-98-09.pdf` .

[4] B. Bokowski and M. Dahm. *Poor Man's Genericity for Java*. Springer Verlag, Freie Universität Berlin, Institut für Informatik, Takustr. 9, 14195 Berlin, 1998.
`http://page.inf.fu-berlin.de/~bokowski/pmgjava` .

[5] B. Bokowski, M. Dahm, and A. Spiegel. *Barat*. October 2000.
`http://sourceforge.net/projects/barat` .

[6] D. L. Bruening. *Systematic Testing of Multithreaded Java Programs*. May 1999. Master Thesis
`http://web.mit.edu/iye/www/docs/thesis.pdf.gz` .

[7] J.-D. Choi and H. Srinivasan. *Deterministic Replay of Java Multithreaded Applications*. August 1998. In Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools, pp. 48-59, Welches, Oregon
`http://citeseer.nj.nec.com/choi98deterministic.html` .

[8] W. J. Cody and J. T. Coonen. *Algorithm 722 - Functions to Support the IEEE Standard for Binary Floating-Point Arithmetic*. ACM Press New York, December 1993. Argonne National Laboratory and C-Labs, Volume 19, Issue 4 of ACM Transactions on Mathematical Software (TOMS).

[9] E. G. Coffman, M. J. Elphick, and A. Shoshani. *System Deadlocks*. June 1977. In Computing Surveys, 3(2):67-78
`http://doi.acm.org/10.1145/356586.356588` .

[10] G. A. Cohen, D. L. Kaminsky, and J. S. Chase. *Automatic Program Transformation with JOIE*. December 1999.
`http://www.cs.duke.edu/ari/joie` .

[11] Compaq. *Compaq JTrek Product Information*.
`http://h18012.www1.hp.com/java/download/jtrek` .

[12] M. Dahm, J. van Zyl, et al. *BCEL Byte Code Engineering API*. 2002.
`http://jakarta.apache.org/bcel` .

[13] P. Eugster. *Register Byte Code Optimization*. Computer Systems Institute, ETH Zentrum, CH-8092 Zürich, Switzerland, July 2002.

[14] P. Eugster. *Java Virtual Machine with Rollback Procedure allowing Systematic and Exhaustive Testing of Multithreaded Java Programs*. Formal Methods Group, Computer Systems Institute, ETH Zentrum, CH-8092 Zürich, Switzerland, March 2003. Diploma Thesis.

[15] Formal Methods Group. *JNuke Framework for checking Java Byte Code*. Contact information: Computer Systems Institute, ETH Zentrum, CH-8092 Zürich, Switzerland.

[16] H. S. Gagliano. *Testing of Java Bytecode Transformations*. Diploma Thesis. Institute of Computer Systems, ETH Zentrum, CH-8092 Zürich, August 2002.

[17] E. Gamma, R. Heim, R. Johnson, and J. Vlisside. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, October 1994. ISBN 0-201-63361-2.

[18] J. Gosling, B. Joy, G. Steel, and G. Bracha. *The Java Language Specification*. Addison-Wesley, USA, 2nd edition, 2000.
`http://java.sun.com/docs/books/jls .`

[19] IBM alphaWorks. *CFParse*. September 2000.
`http://www.alphaworks.ibm.com/tech/cfparse .`

[20] IBM alphaWorks. *Jikes ByteCode Toolkit*. March 2000.
`http://www.alphaworks.ibm.com/tech/jikesbt .`

[21] *754-1985 IEEE Standard for Binary Floating-Point Arithmetic*. 1985. ISBN 1-5593-7653-8, 20 pages, IEEE Product No. SH10116-TBR.

[22] J. Kanerva. *The Java FAQ*. Addison-Wesley, 1997. ISBN 0-201-63456-2,
`http://java.sun.com/docs/books/faq .`

[23] R. Konuru, H. Srinivasan, and J.-D. Choi. *Deterministic Replay of Distributed Java Applications*. IBM Thomas J. Watson Research Center, 30 Saw Mill River Road, Hawthorne, NY 10532, 2000. 14th International Parallel and Distributed Processing Symposium (IPDPS)
`http://ipdps.eece.unm.edu/2000/papers/konuru.pdf .`

[24] D. Kranzlmüller. *Event Graph Analysis for Debugging Massively Parallel Programs*. GUP Linz, Joh. Kepler University Linz, Altenbergerstr. 69, A-4040 Linz, September 2000.
`http://www.gup.uni-linz.ac.at/~dk/thesis .`

[25] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, November 1999. ISBN 0-201-31009-0.

[26] T. J. LeBlanc and J. M. Mellor-Crummey. *Debugging parallel programs with Instant Replay*. IEEE Computer Society, April 1987. IEEE Transaction on computers C-36(4):471-482.

[27] H. B. Lee and B. G. Zorn. *BIT: A tool for instrumenting Java bytecodes*. December 1997. In Proceedings of the 1997 USENIX Symposium on Internet Technologies and Systems (USITS97), pages 73-82, Monterey, CA, USENIX Association `http://www.cs.colorado.edu/~hanlee/BIT .`

[28] T. Londholm and F. Jellin. *The Java Virtual Machine Specification*. Addison-Wesley, USA, second edition, 1999.
`http://java.sun.com/docs/books/vmspec .`

[29] C. E. McDowell and D. P. Helmbold. *Debugging concurrent programs*. ACM Press, New York, University of California at Santa Cruz, California, 1998. ACM Computing

Surveys (CSUR), Volume 21, Number 4, pp 593-622
`http://doi.acm.org/10.1145/76894.76897` .

[30] J. Meyer. *Jasmin Java Assembler Interface*. New York University Media Research Lab, 719 Broadway, 12th Floor, New York NY 10003, March 1997.
`http://mrl.nyu.edu/~meyer/jasmin` .

[31] D. Neri, L. Pautet, and S. Tardieu. *Debugging distributed applications with replay capabilities*. 1997.
`http://www.rfc1149.net/download/documents/tardieu-triada97.ps` .

[32] R. Netzer and B. Miller. *On the Complexity of Event Ordering for Shared-Memory Parallel Program Executions*. 1990. In Proceedings of the International Conference on Parallel Processing.

[33] M. Russinovich and B. Cogswell. *Replay For Concurrent Non-Deterministic Shared-Memory Applications*. ACM Press, New York, Department of Computer Science, University of Oregon, Eugene OR 97403, 1996. Proceedings of the ACM SIGPLAN '96 conference on Programming language design and implementation
`http://doi.acm.org/10.1145/231379.231432` .

[34] A. Silberschatz and P. B. Galvin. *Operating System Concepts*. John Wiley & Sons, Inc., Fifth edition, 1997. ISBN 0-471-41714-2
`http://os-book.com` .

[35] W. Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall, Upper Saddle River, NJ 07458, fourth edition, 2001. ISBN 0-13-031999-6
`http://www.prenhall.com/stallings` .

[36] S. D. Stoller. *Testing Concurrent Java Programs using Randomized Scheduling*. Computer Science Dept., State University of New York at Stony Brook, USA, 2002.
`http://www.cs.sunsyb.edu/~stoller` .

[37] Sun Microsystems, Inc. *The Java Tutorial: Understanding Thread Priority*.
`http://java.sun.com/docs/books/tutorial/essential/threads/priority.html` .

[38] Sun Microsystems, Inc. *javadoc*.
`http://java.sun.com/j2se/javadoc` .

[39] Sun Microsystems, Inc. *Java 2 Platform, SE v1.4.1 API documentation*.
`http://java.sun.com/j2se/1.4.1/docs/api/index.html` .

[40] Sun Microsystems, Inc. *Why Are Thread.stop, Thread.suspend, Thread.resume and Runtime.runFinalizersOnExit Deprecated?*
`http://java.sun.com/products/jdk/1.2/docs/guide/misc/threadPrimitiveDeprecation.html` .

[41] K.-C. Tai, R. H. Carver, and E. E. Obaid. *Debugging Concurrent Ada Programs by Deterministic Execution*. IEEE Press, Piscataway, New Jersey, January 1991. IEEE Transactions on Software Engineering, Volume 17, Issue 1, pp. 45-63.

[42] The Real-Time for Java Expert Group. *The Real-Time Specification for Java*. Addison-Wesley, June 2000. ISBN 0-201-70323-8,
`http://www.rtj.org/rtsj-V1.0.pdf` .

[43] S. M. Wynne. *Blackdown.org FAQ*. January 2000.
`http://www.blackdown.org/java-linux/docs/support/faq-release/FAQ-java-linux.html` .