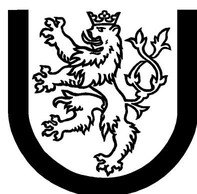

University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitní 8
30614 Pilsen
Czech Republic

An Approach to Concurrent Java Programs Debugging and Run-Time Analysis

Jaroslav Kačer

Technical Report No. DCSE/TR-2003-14
April 2003

Distribution: public

Technical Report No. DCSE/TR-2003-14
April 2003

An Approach to Concurrent Java Programs Debugging and Run-Time Analysis

Jaroslav Kačer

Abstract

With increasing power of today's computers, the Java programming language is more and more chosen by developers as the implementation language of various types of software. The wide range of applications also include concurrent multithreaded programs. However, programming, analyzing and testing concurrent programs is a problem of a higher order than programming and testing classic sequential programs, even if they are implemented at a high level of abstraction that Java provides.

The purpose of this work is to describe an approach to concurrent Java programs debugging and run-time analysis done by means of a simulator of a subset of the JVM functionality and a converter of Java source code. The tested program can be deterministicly traced (just one thread at once) at run-time and the relationships of various entities can be observed and analyzed. As a consequence, the programmer can find potential bugs concerning concurrency issues in a shorter time than when using standard debugging and tracing techniques, usually inefficient for concurrent programs.

This paper first summarizes some requirements that a tool implementing this approach should meet, then it presents work related to concurrent programs testing and analysis, done worldwide. Subsequently, it discusses an invented state-space model of concurrent Java programs and a controller based on it, executing the tested program. Finally, it shows how Java programs are transformed in order to use the controller. Chapter 6 concludes the document.

Copies of this report are available on
http://www.kiv.zcu.cz/publications/
or by surface mail on request sent to the following address:

University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitní 8
30614 Pilsen
Czech Republic

Copyright ©2003 University of West Bohemia in Pilsen, Czech Republic

Contents

1 Purpose of the Work **5**

1.1 Detection of Safety Violations 6

1.2 Liveness Observation 7

1.3 Observation of Relationships between a Concurrent Program's Entities 7

1.4 Activity Logging 7

1.5 Invariant Checking 8

2 Related Work **9**

2.1 Petri Nets 9

2.1.1 Usability 10

2.2 LTSA – Labeled Transition System Analyzer 10

2.2.1 FSP – Finite State Processes 11

2.2.2 The Technique of Analysis 12

2.2.3 Usability 12

2.3 The Model Checker SPIN – Checking Formal Models 12

2.3.1 Description of a Process and the State Space of the System 13

2.3.2 Description of Correctness Claims 13

2.3.3 Finding Possible Correctness Violations 14

2.3.4 Usability 15

2.4 Java PathFinder 1 – Constructing Formal Models from Java Multithreaded Programs 15

2.4.1 Usability 16

2.5 The Model Checker VeriSoft – Checking UNIX Concurrent Systems 16

2.5.1 Description of a Concurrent System 17

2.5.2 State Space Exploration Techniques 17

2.5.3 Usability 18

2.6 An Executable Model of Java Programs – The ExitBlock Algorithm 18
2.6.1 Testing Criteria 19
2.6.2 The Basic Algorithm 19
2.6.3 ExitBlock-RW 21
2.6.4 ExitBlock-DD 21
2.6.5 Usability 22
2.7 Java PathFinder 2 – Executable Models of the Java Virtual Machine 22
2.7.1 Abstraction 23
2.7.2 Static Analysis 23
2.7.3 Runtime Analysis 23
2.7.4 Usability 24

3 State-Space Model of Concurrent Java Programs 25
3.1 Basic Entities of a Concurrent Java Program 25
3.2 The State Space 27
3.3 Walking through the State Space 30
3.3.1 The Relation of Neighbourhood 30
3.3.2 Simple Linear Neighbourhood 31
3.3.3 Conditional Neighbourhood 31
3.3.4 One-of-N Choice Neighbourhood 32
3.3.5 Cyclic Neighbourhood 33
3.3.6 Special 1-State Cyclic Neighbourhood 33
3.3.7 One-of-N Choice Cyclic Neighbourhood 34
3.3.8 All-to-All Neighbourhood 34
3.4 An Example: The Producer-Consumer Problem 35
3.5 JVM-like Entities and Their Relationships 39
3.6 Interesting Entry Points to the JVM 41
3.6.1 Beginnings of Synchronized Blocks 41
3.6.2 Ends of Synchronized Blocks 41
3.6.3 Calls to `wait()` 42
3.6.4 Calls to `notify()` and `notifyAll()` 42
3.6.5 Calls to `destroy()` 42
3.6.6 Priority Changes 42
3.6.7 Calls to `interrupt()` 43
3.6.8 Calls to `interrupted()` and `isInterrupted()` 43

3.6.9 Calls to `join()` . . . 43
3.6.10 Thread Suspensions and Resumes . . . 43
3.6.11 Calls to `stop()` . . . 43
3.6.12 Calls to `setDaemon()` . . . 44
3.6.13 Sleeps . . . 44
3.6.14 Calls to `yield()` . . . 44
3.6.15 Thread Creations, Startups and Deaths . . . 45

4 The Controller 46
4.1 Forcing the Execution Order . . . 47
4.1.1 Possible Optimization . . . 49
4.2 The Need for Various Types of Scheduling . . . 50
4.3 Replacing the Entry Points' Behavior . . . 51
4.4 The Singleton Design Pattern . . . 52

5 The Converter 53
5.1 The Conversion Process . . . 53
5.1.1 Conversion of a Single Class . . . 55
5.2 Parametrization of the Conversion . . . 57
5.3 Conversion of Threads . . . 58
5.4 Conversion of Normal Classes . . . 59
5.5 Conversion of the Main Class . . . 59
5.6 Conversion of Private and Nested Classes . . . 59
5.7 Using Java Source Code Parsers . . . 59

6 Conclusion 61
6.1 Current State of Work . . . 61
6.1.1 Related Publications . . . 61
6.2 Application Area . . . 62
6.3 Goals of the Ph.D. Thesis . . . 62

List of Figures

3.1 Simple Linear Neighbourhood of Thread Consistent States 31
3.2 Conditional Neighbourhood of Thread Consistent States 32
3.3 One-of-N Choice Neighbourhood of Thread Consistent States . . 32
3.4 Cyclic Neighbourhood of Thread Consistent States 33
3.5 Special 1-State Cyclic Neighbourhood of Thread Consistent States 33
3.6 One-of-N Choice Cyclic Neighbourhood of Thread Consistent States 34
3.7 All-to-All Neighbourhood of Thread Consistent States 34
3.8 State-Space Model of the Producer 36
3.9 State-Space Model of the Consumer 37
3.10 State-Space Model of the Producer-Consumer Program 38

4.1 The Controller – A New Layer Inserted between the Program and the Java Virtual Machine 47

5.1 Conversion of a Single Java Class 56

Chapter 1

Purpose of the Work

Testing is a necessary part of the life-cycle of any software product, regardless of its size, application area, or its user. There are many techniques developed in software engineering, dealing with the issue of testing. Usually, they have one thing in common: they are taylored exclusively for purely sequential programs. If a sequential program is executed twice with the same input data and the program does not involve any randomization inside, the same output is obtained in both runs. Therefore, if invalid or unpredicted output data is obtained, it is very easy to reproduce the same program behavior again and it is very likely that a bug causing the problem will be discovered, sooner or later.

However, this is not true for concurrent programs anymore. Concurrent programs contain more or less independent units of code (processes for multitasking and threads for multithreading) executed in parallel. Usually, there is an indeterministic scheduler (at the level of operating system or at the language level) which causes that the concurrency units – or rather their parts, consisting of an arbitrary number of instructions – are always executed in a different order. The total number of all possible schedules of the same concurrent program is therefore almost infinite. Since the concurrency units usually share some data, which is the reason why they are part of the same computation system, any bug concerning the access to these shared data causes that invalid output of the whole computation is generated or even causes the total computation process to behave in an unpredictable way or to crash completely. Finding these bugs and finding the reasons why a concurrent program behaves in a strange and unintended way are not trivial tasks.

Although the Java programming language provides all necessary means for multithreading (concurrency at the level of a single process) already in the basic package and the language was designed to support multithreading from the very beginning, it cannot protect the programmer from all possible traps, pitfalls, and dangerous things that multithreading threatens with.

The usual way of "debugging" used when something goes wrong in the program is to print out comments on the standard or error output. This system can inform the programmer *what* is happening inside the program, but it cannot tell *why* something (e.g. a deadlock) has happened. It does not analyze all the

relationships between all entities of the program that may be the reason of a wrong behavior. The aim of this work is to provide a system that does right that: it analyzes the running program from the point of view of the programmer and it intercepts any change of its properties (regarding the concurrency issues) so it is later able to analyze why a certain action was applied by the Java virtual machine or by the program itself. The analysis of relationships of program entities – threads, locks, monitors, ... – can substantially help the programmer to understand why the program behaves incorrectly.

There exist theoretical methods (see chapter 2) how a concurrent system can be analyzed. These methods work well but they require several conditions to be fulfilled first:

- The system must be described in such a manner that the method can work with it. This can be a problem if the tested program has already been implemented but its model had not been created before so a kind of reverse engineering must be employed here.
- Even if the required model exists, the actual software may not be implemented exactly according to it. So the theoretical method may declare the model as error-free while the actual software is full of bugs. Therefore, the real software must be an exact copy of the model, at least in all aspects concerning concurrency.

Practical methods are usually a mess of different techniques that the programmer is used to apply in sequential programs. This includes printouts, use of debuggers and breakpoints, periodic checking of values of various variables or states of threads, etc. These methods are usually not coordinated with each other and may even influence the tested program in the sense that its behavior is changed and the bug is not found.

This work attempts to systematically employ all possible techniques, including static and dynamic analysis, monitoring states of various object of the program, monitoring relationships between objects of the program, activity logging, and checking for user conditions that must always hold during the program execution. Some demanded features are discussed in the following sections.

Also, it attempts to give the user a chance to step through a concurrent program in such a way that he/she can observe the behavior of any single thread that is in runnable state and its impact on the rest of the program. This "stepping through the program" is exactly defined in section 3.3 and can be either automatic or manual, which is described in section 4.2.

1.1 Detection of Safety Violations

Java introduces so-called `synchronized` blocks to protect data from being accessed or modified from different threads simultaneously. However, the programmer is not forced to use synchronization and therefore, as a result of that, incorrect data can be easily produced. These incorrectly synchronized reads or writes are called safety violations.

Finding all violations against this mutual-locking discipline[1] can be done at the level of source code analysis and does not require the program to be run (but false alerts can be reported then). There are already tools able to do this analysis, for example Eraser [Era].

1.2 Liveness Observation

If mutual locking (or synchronization) is over-used or used in a wrong way, so-called liveness properties of the program (or just a part of the program) can be violated. This applies to deadlocks caused by a wrong order of locking of multiple different locks, badly used `wait()`s (before notifying another thread), badly formulated guarding conditions in monitor methods, badly formulated notification conditions in monitors, and some other situations.

Therefore, observing the state of every thread and knowing about every state change of a thread may help the programmer to discover bugs related to liveness. The programmer is then able to compare the actual behavior of the program to the intended one and find the buggy piece of code.

1.3 Observation of Relationships between a Concurrent Program's Entities

As described later in section 3.5, a Java concurrent program is in fact composed of entities of various types, invisible[2] to the programmer at the level of Java source code. These entities are, for example, threads, locks used for synchronization, lock wait sets, lock delayed sets, thread join sets, and some others.

Monitoring these entities (usually hidden in the JVM representation of the program) and monitoring their relationships can considerably help the programmer to understand the actual behavior of his/her program. If the behavior differs from the intended one, the programmer can easily find the places in his/her source code where these relationships were changed and he/she can conclude a result from that.

1.4 Activity Logging

Activity logging does not only involve thread-specific issues but applies to all programs in general. It comprises logging calls to methods, logging exception throwing, logging creation of objects of interesting classes, logging communication with the environment (file opening and closing, reading and writing), and some other situations that could be monitored.

[1] Synchronization uses locks that guarantee mutual exclusion.

[2] Only some of them are invisible, some of them are visible and are mapped to real Java objects.

1.5 Invariant Checking

An invariant is a user statement describing a certain condition about the tested program that must hold at any time. Evaluation of this statement returns a logical value. In the case of a correct program, evaluation of every invariant should always return `true`.

An invariant could be described as a method returning a `boolean` value. The method can be called at runtime and whenever an invariant is violated (the condition is evaluated as `false`), the user can be informed and a particular action can be taken.

Chapter 2

Related Work

So far, much effort has been put to many different solutions of the problem being solved in this work. In this chapter, some of them – the most known and widespread – will be presented. Some of them only analyze systems described using a formal model, some of them directly analyze an already implemented program in its source or runnable form to get true and unbiased results. Only some of the tools and methods presented here are specialized to Java, those that are not would usually be usable with Java after some rework.

We will start with the most theoretical approaches that are at the highest level of abstraction and then we will proceed with methods focused on issues of implementation in a programming language. Finally, we will present algorithms and tools that work directly with Java software, either in the form of source code or in the form of byte code (compiled source code).

2.1 Petri Nets

Petri Nets is a formal and graphical appealing language which is appropriate for modelling systems with concurrency. It was Carl Adam Petri who defined the language. It was the first time a general theory for discrete parallel systems was formulated. The language is a generalisation of automata theory such that the concept of concurrently occurring events can be expressed.

A Petri net is a graphical and mathematical modeling tool. It consists of *places*, *transitions*, and *arcs* that connect them. *Input arcs* connect places with transitions, while *output arcs* start at a transition and end at a place. There are other types of arcs, e.g. inhibitor arcs. Places can contain *tokens*; the current state of the modeled system (the marking) is given by the number (and type if the tokens are distinguishable) of tokens in each place. Transitions are active components. They model activities which can occur (the transition fires), thus changing the state of the system (the marking of the Petri net). Transitions are only *allowed to fire if they are enabled*, which means that all the preconditions for the activity must be fulfilled (there are enough tokens available in the input places). When the transition fires, it removes tokens from its input places and

adds some at all of its output places. The number of tokens removed / added depends on the *cardinality* of each arc. The interactive firing of transitions in subsequent markings is called token game.

Petri nets are a promising tool for describing and studying systems that are characterized as being *concurrent, asynchronous, distributed, parallel, nondeterministic, and/or stochastic.* Tokens are used in these nets to simulate the dynamic and concurrent activities of systems. As a mathematical tool, it is possible to set up state equations, algebraic equations, and other mathematical models governing the behavior of systems.

To study performance and dependability issues of systems, it is necessary to include a *timing concept* into the model. There are several possibilities to do this for a Petri net; however, the most common way is to associate a *firing delay* with each transition. This delay specifies the time that the transition has to be enabled, before it can actually fire. If the delay is a random distribution function, the resulting net class is called *stochastic Petri net.* Different types of transitions can be distinguished depending on their associated delay, for instance *immediate transitions* (no delay), *exponential transitions* (delay is an exponential distribution), and *deterministic transitions* (delay is fixed).

2.1.1 Usability

Using Petri nets requires that a model of the simulated / analyzed system has to be constructed first. This requires a detailed analysis of the source code of the tested software if not done during the design phase. Constructing the model is further complicated by the fact that a Java concurrent program can hardly be considered a discrete system so a process of abstraction must be applied. Petri nets are certainly a helpful verification mean in the design phase but almost unusable if analysis of an already existing software is required.

2.2 LTSA – Labeled Transition System Analyzer

LTSA is a verification tool for concurrent systems developed by Jeff Magee and Jeff Kramer. It mechanically checks that the specification of a concurrent system satisfies the properties required of its behavior. In addition, LTSA supports specification animation to facilitate interactive exploration of system behavior. LTSA can be downloaded from [LTSA].

A system in LTSA is modelled as a *set of interacting finite state machines.* The properties required of the system are also modelled as state machines. LTSA performs *compositional reachability analysis* to exhaustively search for violations of the desired properties. More formally, each component of a specification is described as a Labelled Transition System (LTS), which contains all the states a component may reach and all the transitions it may perform. However, explicit description of an LTS in terms of its states, set of action labels and transition relation is cumbersome for other than small systems. Consequently, LTSA supports a process algebra notation FSP (Finite State Processes) for concise

description of component behavior. The tool allows the LTS corresponding to a FSP specification to be viewed graphically.

LTSA considers a process as having a state modified by invisible (or atomic) actions. Each action causes a transition from the current state to the next state. The order in which actions are allowed to occur is determined by a transition graph that is an abstract representation of the program. In other words, processes can be modelled as finite state machines. Although the representation of a process is finite, the behavior described need not be finite.

2.2.1 FSP – Finite State Processes

For larger number of states and transitions, an algebraic notation called Finite State Processes (FSP) is used to describe process models. Every FSP description has a corresponding state machine (LTS) description. A process is described by its states and the actions between them. Processes are in uppercase, actions in lowercase letters. The terminal state of a process (if the process terminates) is called `STOP`. An action `a` transforming a process from state `A` to state `B` is prefixed before `B`, together with the prefix operator `->`:

```
A = (a -> B)
```

To describe repetitive behavior, recursion can be used. For example, a non-terminating process describing a ligth switch with two states `ON` and `OFF` and two actions `on` and `off` would look like the following:

```
SWITCH = OFF,
OFF = (on  -> ON),
ON  = (off -> OFF).
```

If we substitute the definition of `ON` in the definition of `OFF`, we will get the following result:

```
SWITCH = OFF,
OFF = (on -> (off -> OFF)).
```

The definition of `OFF` can be further used in the definition of the initial state of `SWITCH`, which will produce the following:

```
SWITCH = (on -> off -> SWITCH).
```

The three different definitions of the `SWITCH` process generate identical state machine.

The sequence of actions produced by the execution of a process (or a set of processes) is referred to as a *trace*. In general, processes have many possible execution traces.

More than one transition can lead from a state to several next states. If an action occurs, the process passes to the next state corresponding to the performed action. The choice is expressed by the operator `|`.

```
X = (a -> A | b -> B),
A = (x -> X),
B = (x -> X).
```

The choice can be even non-deterministic, i.e. the same action `x` can lead both to state `A` and to state `B`:

```
X = (x -> A | x -> B),
A = (y -> X),
B = (y -> X).
```

There are many other language elements in FSP, like guarded actions (the keyword `when`), indexed processes and actions, process labeling and hiding, etc. They can be found in [Conc] and will not be discussed here for space issues.

2.2.2 The Technique of Analysis

After the processes are described, LTSA can perform a breadth-first search on the LTS. If a property violation or a deadlock is found, the shortest trace of actions that would lead to the property violation or deadlock is displayed. Alternatively, this algorithm can be replaced by Holtzmann's bitstate hashing technique – Supertrace – which is not a complete search of the state space. Supertrace is only used if the complete search is not possible due to size.

2.2.3 Usability

LTS, FSP, and the LTS Analyzer (now in version 2.2) is a set of extremely helpful tools that can diagnose the concurrent system before it is implemented. Therefore, the actual software has to be written according to the model. However, as in case of any other theoretical method, errors can be introduced to the code during further development, so another tool has to be used later.

2.3 The Model Checker SPIN – Checking Formal Models

SPIN is a verification system for models of distributed software systems. It was developed in Bell Laboratories and its brief description can be found in [SPIN]. It supports the design and verification of asynchronous process systems. The verification is mainly focused on proving the correctness of process interaction, specified either with randez-vous primitives or with asynchronous message passing through buffered channels, or access to shared variables or any combination of these.

The verified system's design specification is written in the verification language Promela and the correctness claims are specified in the syntax of Linear Temporal Logic (LTL). The Promela language imposes two important constraints

on the modelled sytem: it must be bounded and it must have only countably many distinct behaviors.

The typical usage of SPIN has the three following steps:

1. The specification of a high-level model of a concurrent system or a distributed algorithm, using SPIN's graphical front-end XSPIN.
2. Interactive simulation, to gain basic confidence about the behavior of the modelled system's design.
3. Generation of an optimized on-the-fly verification program from the high-level specification. This verifier is compiled and can be thus directly executed. The verifier is able to find counterexamples to correctness claims (if there are any) that can be used again during a repeated interactive simulation to inspect details of the cause of the correctness violations.

2.3.1 Description of a Process and the State Space of the System

The verified concurrent system, described in Promela, consists of one or more user-defined *process templates* (`proctype` definitions) and at least one *process instantiation*. The templates define the behavior of different types of processes. Any running process can instantiate further asynchronous processes, using the process templates.

Each process template is translated to a *finite automaton*. The global behavior of the concurrent system is obtained by computing an asynchronous *interleaving product of automata*, one automaton per asynchronous process behavior. The resulting global system behavior is itself again represented by an automaton. This interleaving product is referred to as the *state space of the system* or as the *global reachability graph* (because it can be represented as a graph).

2.3.2 Description of Correctness Claims

Every correctness claim is expressed as an LTL formula. An LTL formula can contain a *propositional symbol* p, combined with *unary or binary, boolean and/or temporal operators*, using the following grammar:

```
f     := p  | true | false | (f) | f binop f | unop f
unop  := [] | <>   | !
binop := U  | &&   | ||     | ->  | <->
```

The operators have these meanings: `[]` means always, `<>` means eventually, `!` is logical negation, `&&` is logical and, `||` is logical or, `->` is implication, `<->` is equivalence and `U` means strong until. For example, `[] (p U q)` states that it is always guarateed that p remains true at least until q becomes true.

SPIN generates a *Büchi automaton*[1] for each formula with several states, either accepting or non-accepting. The algorithm is following: First, the formula is converted to *normal form.* An initial state is created, marked with the formula to be matched and an incoming edge. The remainder of the automaton is computed *recursively.* At each stage, a subformula that remains to be satisfied is taken and, according to its leading operator, the current state may be split into two states, each of them inheriting a different part of the subformula. In the last phase, some of the states are identified as accepting according to the *presence or absence of* subformulae with *until operators.*

2.3.3 Finding Possible Correctness Violations

When a correctness claim is translated to a Büchi automaton, it is merged with the automaton of the global state space – the so called synchronous product is computed. The result is again a Büchi automaton. If the language accepted by this automaton is *empty*, the original correctness claim is *not satisfied.* If the language is *nonempty*, it contains those *behaviors that satisfy the original LTL formula.*

In SPIN, the correctness claims describe errorneous system behaviors, i.e. *behaviors that are undesirable.* The reasons for this approach are stated in the following paragraphs. So the verification process then either proves that such erroneous behavior is impossible or it provides examples of behavior that match.

A proof of a positive claim requires that the language of the system is included in the language of the claim. On contrary, a proof of a negative claim requires that the intersection of both languages is empty. The worst-case state space size of both the positive and negative claim proof is the size of the Cartesian product of the system and the claim. The best-case state space size of the positive claim proof is the size of sum of the two state spaces and the best-case state space size of the negative claim proof is zero. This is the reason why SPIN works with negative claims and uses language intersection.

A Büchi automaton accepts a system execution if and only if that execution forces it to *pass through one or more of its accepting states infinitely often.* (Such behaviors are called *acceptance cycles.*) To prove that no execution sequence of the system matches the negated claim, it suffice to prove the *absence of acceptance cycles* which is of much less complexity than proving the former behavior. So the computation terminates when an acceptance cycle (a counterexample to the correctness claim) is found or when the complete intersection product has been computed.

SPIN uses *Tarjan's depth-first search algorithm* to find an acceptance cycle in the global reachability graph. The algorithm constructs strongly connected components. If at least one accepting state exists in a strongly connected component, then there exists an acceptance cycle. If the acceptance state is reachable from the initial system state then the acceptance cycle is also reachable, which is a counterexample to the original correctness claim.

[1] A Büchi automaton is defined over infinite input sequences, rather than finite ones as in standard finite state machine theory.

2.3.4 Usability

Although SPIN is a mature and proven tool, it cannot be found convenient for the purposes of this work, enumerated in chapter 1. The reasons are as follows:

1. SPIN works with formal models only, i.e. with software described in its design phase, it does not work with real programs (either in runnable or source form), i.e. software brought to its implementation phase.
2. SPIN is language-independent and therefore it cannot deal with all issues specific to Java. Process interaction is limited to shared variables and message passing, some Java language constructs (like synchronized blocks) would be quite hardly expressed.
3. Moreover, the possibility to formulate correctness claims and check for their validity or violations is only a part of our requirements. Also, the way how the claims are formulated is not convenient. A Java-like style would be necessary.

2.4 Java PathFinder 1 – Constructing Formal Models from Java Multithreaded Programs

Since a tool for verification of concurrent systems exists (SPIN) but it is able to work with formally defined model only, a tool creating these models from existing source code was developed, allowing SPIN to be used for verification of Java concurrent programs.

The tested Java program may contain assertions which are traslated to the corresponding assertions in the Promela language. The SPIN model checker then checks for any violation of the specified assertions, and for deadlocks, too. The assertions can also be expresses directly in the linear temporal logic (LTL), as in the case of SPIN. In the former case, the tester can use the `Verify` class and its static method `assert()`. The body of the method is not translated to Promela (only its parameter – the tested condition) but can be used during normal testing.

The process of translation from Java source code to Promela can be found in [PF1] and will not be presented here due to its complexity. Java PathFinder supports just a subset of the Java language, however, this subset seems to be a reasonable part of the whole language and the the other, unsupported, part will unlikely be missed. The two parts are as follows:

Supported Features: Dynamic creation of objects with data and methods, class inheritance, threads, synchronization primitives for modeling monitors (the `synchronized` keyword and methods `wait()` and `notify()`), exceptions, thread interrupts, assignment statements, conditional statements, loops, ...

Unsupported Features: Packages, overloading, method overriding, recursion, strings, floating-point numbers, some threads operations like `suspend()` and `resume()`, control statements like `continue`.

The arrays are not modeled as Java objects, they are traslated to Promela special arrays. Also, the predefined class library is not converted, only the user's source code. One assumption is made about the analyzed Java program: It must have a finite and tracable state space.

The translator is implemented in Common Lisp. A part of the translator is a parser of the Java language, written in MoscowML. The Promela language had to be changed slightly in order to make the traslation easier.

2.4.1 Usability

Java PathFinder has already proved its qualities. It has been used in NASA to test real programs in the project of development of a new operating system for the Deep-Space 1 space craft. Several errors were found, concerning time-dependend behavior leading to a deadlock. However, Java PathFinder is no longer being developed in the form it was presented here. The authors decided to avoid translation to Promela which seems to be a difficult task (and which can probably be never 100% completed) and the new version of PathFinder works directly with bytecode. (See section 2.7.)

2.5 The Model Checker VeriSoft – Checking UNIX Concurrent Systems

VeriSoft is a verification tool for systematic exploration of the state space of systems composed of several concurrent processes executing arbitrary C code. Therefore, there is a substantial difference between the model checker SPIN presented in 2.3 and VeriSoft: SPIN needs a formal model of the examined system written in a special language (Promela) while VeriSoft operates with real source code of the verified system. A brief description of the model checker Verisoft and the algorithms it uses can be found in [VS].

Every process of the concurrent system to be analyzed is mapped to a Unix process. The execution of the system processes is controlled by an external process, called the *scheduler*. By suspending and resuming a given process, the scheduler is able to step through the state space (see section 2.5.1) of the examined system. When a deadlock or an assetion violation is detected, the program is stopped and the computation history is shown to the user.

VeriSoft also checks for *divergencies* and *livelocks*. A divergence occurs when a process does not attempt to execute any visible operation[2] for more than a given amount of time. A livelock occurs when a process has no enabled transition[3] during a sequence of more than a given number of successive global states.

[2]See next section for explanation of this term.
[3]This term is also explained later.

2.5.1 Description of a Concurrent System

The verified concurrent system is composed of a finite set $\mathcal{P}$ of *processes* and a finite set $\mathcal{O}$ of *communication objects*. Each process $P \in \mathcal{P}$ executes a sequence of operations – a sequential deterministic program. Processes communicate with each other by performing operations on communication objects. A communication object $O \in \mathcal{O}$ is defined by a pair (V, OP), wher V is the domain of the object and OP is its set of operations.

Operations on communication objects are called *visible* while other operation are by default *invisible*. The execution of a visible operation is *blocking* if it cannot be completed. Invisible operations cannot block.

At any time, the system is said to be in a *state*. It is in a *global state* when the next operation to be executed by every process in the system is a visible operation. The *initial state* of the system is also considered to be global. A *process transition* is a visible operation followed by a finite sequence of invisible operations performed by a single process. A transition may be disabled when the execution of its visible operation is blocking, otherwise it is enabled. By completing a transition of a process from a global state s, the system reaches a successor of s.

So the global state space of a concurrent system is seen by VeriSoft as a set of global states plus a set of transitions between pairs of these global states plus an initial global state.

2.5.2 State Space Exploration Techniques

The basic algorithm of state-space exploration is based on *recursion*. It starts with the initial state and executes all enabled transitions leading from it. The same algorithm repeats in each state found from the initial state. The algorithm assumes that each state has a *unique identifier* and a *unique and managable representation* that can be stored to its data structures. However, this assumption is not valid anymore when dealing with real programs because the values of all memory locations a process can read from or write to are different if the same state is reached for the second time.

The basic algorithm uses the following data structures:

- A set of states whose *successors are still unexplored*.
- A hashtable of already *visited states*.
- Sets of *enabled transitions*, one set per state.

Because of the broken assumption stated above, so called *state-less search* algorithm can replace the original one. The main difference is that the latter one does not store any intermediate results to the memory. An unpleasant result is that the algorithm may not terminate if the program contains cycles, even if the set of global states is finite.

VeriSoft uses an enhanced version of the state-less search algorithm, employing so called *sleep sets* and *persistent sets*. They serve the following purposes:

Sleep Set – A sleep set is associated with every global state. It contains those transitions that are enabled but will not be explored from a given state. Initially, this set (the sleep set of the initial state) is empty. If a transition is executed, it is added to the sleep set of the state from which it was executed so that it cannot be executed anymore if the computation reaches the same state again. The target global state of a transition t inherits the sleep set of the last explored state[4] but only those transition that are independent[5] with t are kept in the set.

Persistent Set – A persistent set is also associated with every global state. It is such a subset T of the set of all enabled transitions in a given state s that all transitions not in T (and enabled in s or in a state reachable from s through transitions not in T) are independent with all transitions in T.

The modified state-less search algorithm explores only those transitions in every global state that are in its persistent set but not in its sleep set. Thanks to this technique, the run-time explosion of the original state-less search is avoided and the number of transitions that have to be explored is considerably reduced.

2.5.3 Usability

VeriSoft seems to be an efficient and powerful tool for verification of concurrent systems. Also, its sophisticated method of reduction of transitions to be explored gives it additional value. Although the principles it is based on are applicable anywhere, the tool itself is restricted to operate in Unix environment only (it works with processes of the operating system) and therefore not usable for purposes of Java programs analysis and debugging.

2.6 An Executable Model of Java Programs – The ExitBlock Algorithm

ExitBlock is an algorithm for finding errors in concurrent Java programs resulting from unintended timing dependencies. The algorithm was invented and implemented at MIT and its description can be found in [ExBlk1] or, in more detail, in [ExBlk2]. ExitBlock enumerates all possible behaviors of a Java multithreaded program for a given input. It does not need a separate model or specification of the tested program or its source code, it does the testing dynamically using information gathered while running the program.

ExitBlock is able to detect common timing-dependent errors, for example errors that occur only in certain orderings of modules that are individually correctly synchronized, or using a simple `if` statement instead of a `while` cycle to test a condition variable. It also has an extended version that is able to detect deadlocks in the program.

[4] The state from which the transition leads.

[5] Two transitions t_1 and t_2 are independent if they can neither disable nor enable each other and when from a given global state s another global state s' can be reached both when t_1 is executed first and then t_2 or vice versa. – Enabled independent transitions are commutative.

The implemented algorithm needs a special virtual machine in order to be executed, conventional JVMs cannot be used. Such a JVM – the Rivet JVM – was developed along with the tool at MIT.

2.6.1 Testing Criteria

ExitBlock requires that the tested program comply to the three following testing criteria:

Mutual-Exclusion Locking Discipline – This discipline dictates that each shared variable is associated with *at least one mutual-exclusion lock* and that the lock(s) are *always held* whenever any thread accesses that variable. Since ExitBlock is not able to detect violations of this discipline itself, it uses the Eraser algorithm [Era], running in parallel, to detect them.

Finalization – In Java, the `finalize()` methods can be run in any order and at any time, depending on when the JVM schedules them to run. The algorithm assumes that the finalizers do not influence the rest of the program in the sense that neither assertions nor deadlocks can result from different schedules of finalizers.

Terminating Threads – ExitBlock requires the tested program (every thread of the program) to have a defined *endpoint*. ExitBlock uses the depth-first search algorithm to enumerate all possible schedules of the program. With a non-terminating thread, the algorithm would never reach a leaf in the tree of possible schedules and it would not therefore inspect the rest of schedules but would stay locked in a cycle forever.

2.6.2 The Basic Algorithm

The ExitBlock algorithm is based on the following idea: Let's divide the program into atomic blocks (based on synchronized blocks). Shared variable cannot be accesses outside these atomic blocks because the locking discipline would be violated. Shared variables modified by a thread inside of an atomic block cannot be accessed by other threads until the original thread exits that atomic block. This means that enumerating possible orders of atomic blocks of a program covers all possible behaviors of the program. So the algorithm only needs to consider schedules at the level of atomic blocks, not at the level of Java bytecode instructions.

All possible atomic blocks cannot be enumerated statically since their number is data-dependent. So it is necessary to run the program and enumerate all possible schedules using the depth-first algorithm.

The algorithm first executes one complete schedule of the program. Then, it goes back up from the end of the program to the end of the last atomic block. Here, by choosing a different thread to run, the algorithm creates a new branch of the schedule tree. The program is again executed until completion. This process is systematically repeated. If there is no such thread in a certain node

of the tree that has not run yet from here, the algorithm goes back up one atomic block (one level in the tree) and starts to make new branches from here. The algorithm terminates if there is no such thread in the initial state which means that all schedules have already been inspected.

The algorithm needs to have guaranteed that it can back up the program to a previous state (before the current branch was executed), which is done using checkpointing, and it must also be provided with the ability of deterministic replay to ensure the program receives the same inputs as it is re-executed. Both the checkpointing and the deterministic replay are provided by the Rivet virtual machine.

An atomic block ends with a lock exit or the end of a thread and it begins at the point where the last atomic block ends or at the beginning of a thread. So non-synchronized code is grouped together with its following synchronized code to form one atomic block. This produces as few atomic blocks as possible and therefore the number of possible schedules is also reduced. Nothing changes if there are nested synchronized blocks. The borders of atomic blocks are still lock exits and thread births and deaths, nothing more.

Every thread of the program is put in one of the three following sets:

Block Set – A thread is in a block set of a lock if it needs to acquire that lock but the lock is held by another thread.

Delayed Thread Set – A set of threads not allowed to execute at a checkpoint in order for ExitBlock to schedule another thread. This set contains threads already executed from that checkpoint. They are all re-enabled (removed from this set) after the first atomic block of the branch.

Enabled Set – All other threads are put in the enabled set.

These sets are attributed to every checkpoint in the program. When the execution returns to a checkpoint, these sets must be restored. This avoids the situation when a thread created after a checkpoint would be scheduled when the execution gets back to that checkpoint, i.e. it would be scheduled at a place and "time" where it "does not exist yet".

If the currently running thread cannot obtain a lock, a new lock must be scheduled instead. Since threads can switch on atomic block boundaries only, the running thread and the current branch of the tree must be aborted, the program must undo back to the end of the previous atomic block and another thread must be scheduled instead. The original thread is placed in the block set of the lock.

Call to the `wait()` method of a lock are handled in the way that other locks exits are – an atomic block is terminated. The thread calling `wait()` is also removed from the set of enabled threads. On contrary, all threads notified with a call to `notifyAll()` are put to the block set of the respective lock because all notified threads must first obtain the lock in order to get schedulable. With `notify()`, the algorithm is the same but just one thread can be woken up. Since it is a nondeterministic behavior, a new branch must be created for every potentionally woken-up thread. Because a different thread must be notified in

every branch, another thread set exists: the **No Notify Set**. It contains threads already notified in a branch that cannot be notified again.

In this basic version of the algorithm, the number of schedules that must be explored grows *exponentially* in both the number of threads and the number of locks used by each thread.

2.6.3 ExitBlock-RW

The ExitBlock-RW algorithm is a modification of the basic ExitBlock algorithm enabling some schedules to be omitted during the program verification. If two atomic blocks have no data dependencies between them, then the order of their execution with respect to each other has no effect on whether an assertion is violated or not. The ExitBlock-RW uses data-dependency analysis to prune the tree of explored schedules.

While executing an atomic block, all reads and writes are recorded. Instead of delaying the current thread, a set of delayed threads is kept along with the reads and writes that were executed in the atomic block. Only those threads from the delayed set are re-enabled after the next atomic block, whose reads and writes intersect with the currently running thread's reads and writes.

Two sets of reads and writes (r_1, w_1) and (r_2, w_2) interact if and only if $(w_1 \cap w_2 \neq \emptyset) \vee (r_1 \cap w_2 \neq \emptyset) \vee (w_1 \cap r_2 \neq \emptyset)$.

The number of schedules that must be explored in this version of ExitBlock is *polynomial* in the number of locks and still *exponential* in the number of threads.

2.6.4 ExitBlock-DD

In general, two kinds of deadlock can be distinguished in concurrent Java programs according to the authors of ExitBlock:

Lock-Cycle Deadlocks involve two or more threads blocked only on locks, i.e. blocked when they attempt to enter a synchronized block.

Condition Deadlocks involve two or more threads that are blocked inside a synchronized block on `wait()`, i.e. after they have found that a guarding condition does not hold.

As for the former kind of deadlock, the algorithm must keep track of *thread-lock cycles* since at least two locks and two threads must be involved in a deadlock. If a lock cannot be acquired by a thread, the algorithm uses so-called reverse *lock chain analysis*. If the current thread cannot obtain a lock and it is possible to follow a cycle of owner and lock relationships back to the current thread, then a lock-cycle deadlock is detected.

To detect the latter kind of deadlock is much easier and straitforward. A condition deadlock occures when the algorithm *runs out of enabled threads*, i.e. the Enabled Set is empty at a certain checkpoint. A deadlock cannot be reported if there are delayed threads that cannot be scheduled due to the used algorithm but would be runnable in reality.

2.6.5 Usability

The ExitBlock algorithm and its implementation is targeted directly to the Java platform. It is able to detect timing dependencies errors and deadlocks. Moreover, it uses Eraser to detect race conditions resulting from improper (unused) synchronization. The main advantage of ExitBlock usage is its *ability to test all possible schedules* for given input data in one run and the fact that it *needs no formal model* but uses a real program compiled to Java bytecode. On the other hand, a special virtual machine must be used.

2.7 Java PathFinder 2 – Executable Models of the Java Virtual Machine

Java PathFinder 2 is an explicit-state model checker for programs written in Java. It is itself written in Java and contains a special-purpose Java virtual machine MC-JVM. It brief description can be found in [PF2]. Java PathFinder 2 works directly with Java bytecode, not with source code, as its predecessor.

Currently, it can only check deadlocks and invariants. An invariant is specified as a Java method that returns a `boolean` value. Both the invariant and the program to be checked are converted into bytecode and given as input to the model checker. The checker consists of two parts: the MC-JVM that executes the bytecode and the depth-first search algorithm that does the traversal of the state-space graph of the program. The algorithm can tell the MC-JVM to:

- move *forward* one step,
- move *backward* one step,
- *evaluate* the current invariant.

The MC-JVM keeps track of visited states, also the current path of states is stored on the stack which allows easy backtracking. The output of the model checker is whether or not the invariant holds or the program is deadlock-free. If a deadlock is found, a counterexample is provided which can be fed back to the MC-JVM in a simulation mode in order to recreate the error.

The most important features of Java PathFinder 2 are as follows:

Canonical Heap Representation – Regardless of the order of instructions, dynamic and static memory is always allocated in the same memory locations as they were allocated in a previous run. This reduces the size of the state space.

Garbage Collection – The MC-JVM has its own garbage collection scheme to prevent the state space from growing indefinitely.

Nondeterminism – The MC-JVM supports nondeterminism by trapping calls to `Verify.Random(int)` and `Verify.RandomBool()`.

Atomicity – The level of atomicity can be set to one bytecode instruction, to one Java instruction, to one line of Java code, or to independent Java blocks[6].

Structured State – Each state of the program can be highly structured, since it consists of a number of different Java objects.

2.7.1 Abstraction

A so-called abstraction is provided by PathFinder which converts a Java program to an abstract program with respect to user-specified abstraction criteria. The user can specify abstractions by removing variables or adding new variables to the abstract program. The new variables typically depend on the already existing variables that can (but need not) be removed. The inter-class abstraction is also possible. In such case, a new abstract variable depends on "old" variables from several classes.

Abstraction can be applied to components of the whole program only, if it is too complicated to apply abstraction globally.

2.7.2 Static Analysis

Java PathFinder uses static analysis to compute information to perform partial order reduction during model checking. The static analyzer uses program dependencies to identify safe program blocks. A safe block is a block of consecutive Java statements that can be executed by the virtual machine without worrying about interleaving, e.g. a sequence of statements using only local variables.

Java PathFinder uses static analysis to alleviate the state explosion problem during model checking.

2.7.3 Runtime Analysis

Runtime analysis is based on the idea of *executing the program just once* to extract various kinds of information. This information can be used to *predict* whether other different execution traces may violate some properties of our interest. The algorithm is able to detect even those violations that did not happen during the one run of the program but would potentially happen in another run with different schedule. However, it is not guaranteed that all violations are found and it is not guaranteed that all found violations are not false.

An example of runtime analysis is the data race detection algorithm Eraser, implemented in PathFinder. Or the locking order analysis, looking for potential deadlocks.

Runtime analysis can be used in two modes. First, it can be used *stand-alone* in simulation mode. Second, runtime analysis can be used to *guide the model checker*. PathFinder can generate a so-called *race window* consisting of the

[6]Two blocks of code are independent if they can be executed concurrently.

threads involved in a race condition. The model checker is then launched, focusing only on the race window by forcing the scheduler always to pick threads in the window before other threads. In addition, the window can be extended with threads that write to objects read by the threads in the original window.

2.7.4 Usability

Java PathFinder 2 is intensively used in NASA. It is an efficient tool for finding deadlocks and race conditions. Its plus is that it works directly with Java bytecode. Its qualities can be proved by the fact that it was used to detect a deadlock in software called the Remote Agent in 1999, when in deadlocked in space in the middle of operation.

Chapter 3

State-Space Model of Concurrent Java Programs

The Java language supports multithreading from its nature, therefore all Java programs use the same language elements and patterns. There are no additional libraries, no API functions of an operating system, just an exactly defined set of keywords, standard classes and their methods.

This allows us to define a model of concurrent Java programs, analyze source code of any program and observe how the program behaves when mapped to the model. The model itself and its practical usage for purposes of program debugging and run-time analysis are descibed in the following sections.

3.1 Basic Entities of a Concurrent Java Program

If a typical Java concurrent program is analyzed, the following basic types of entities can be found:

Threads – A dynamic set of runnable objects. Every thread has its own local data set, e.g. thread attributes, thread methods' local variables, own stack, etc.

Monitors – A dynamic set of objects holding global data that threads use for mutual communication. Methods manipulating with a monitor's data must necessarily be synchronized. Threads keep references to monitors whose data have to be shared among them.

Note that we completely exclude from our model those variables that are shared but access to them (both reading and writing) is not synchronized at all or only partially (at some places), e.g. writing is synchronized while reading is not. According to the Java virtual machine specification [JVMS], the Java platform is not obliged to update the master copy in the main memory of those variables

whose writing is not synchronized and it is not obliged to fetch the current value from the main memory of those variables whose reading is not synchronized. Therefore, in our further reasoning, we will assume that all accesses to shared variables obey the proper mutual-exclusion locking discipline.

Two types of operations can be distinguished in a thread's code:

Local Computing – Sequence of operations not manipulating with global data, only with the thread's local data.

A Call to a Monitor Method – Sequence of operations manipulating both with a monitor's (i.e. global) data and the thread's local data. However, the modifiation of the calling thread's local data has no effect on our model. A possible result returned from the method can be stored to a local variable.

This category also includes synchronized blocks that are part of an unsynchronized method or even a synchronized block inside another synchronized block.

A thread's code can be thought of as a sequence of "local computing" blocks (there is no interaction with the thread's surrounding environment), delimited by calls to monitor methods (an interaction with the thread's environment). Monitor method calls are possibly blocking, i.e. threads may get suspended inside a monitor method (and woken up later) or while entering a synchronized block if its lock cannot be acquired at the moment. They can also return a value that can be stored into a local variable. A thread's life is thus as follows:

```
Thread born
Local computing
A call to a monitor method
Local computing
A call to a monitor method
  . . .
Thread dead
```

To explain the state-space model used as a conceptual framework for the presented method of Java program serialization, we will use the following set of assumptions:

- The set of monitors used for computation is static, the cardinality of the set is m, no monitor service uses another monitor services (a monitor cannot reference another monitor).
- The set of threads used for computation is static as well, the dimension of the set is n.
- Monitors are "passive", i.e. they are not implemented using "internal" threads.
- All monitor methods are synchronized.

In fact, these assumptions are not substantial for the presented method implementation, they only simplify the state-space model explanation and must be ignored later when real concurrent programs are debugged and analyzed. Now we are ready to introduce the model description.

3.2 The State Space

A monitor is in *consistent state* when no thread is currently executing its code, that is:

- when no thread is currently running inside its method, and/or
- when one or more threads are passivated inside one of its methods, using `wait()`.

Every consistent state of any monitor can be described as follows:

$$\begin{aligned}
monitor_id &= \{monitor_class_name, monitor_instance_num\} \\
monitor_state_attr &= set_of_monitor_data_values \\
monitor_entry &= \{monitor_id, method_name\} \\
monitor_state &= \{monitor_id, monitor_state_attr\} \\
monitor_state_id &: monitor_state \rightarrow integer
\end{aligned}$$

Here we assume that every monitor has its own integer number (*monitor_instance_num*) that is unique within its class, so a monitor identification is done by a number instead of (or moreover to) its program name (which can be different in different threads). Alternatively, the *id* could be replaced with a reference to the monitor itself, which is also unique.

We define that a thread is in *consistent state* when:

- it has been born, but its `start()` method has not been invoked yet, or
- its control flow reaches a point when it calls a monitor method, or
- it has died already but its data are still valid.

As for the second case, two possibilities can occur:

- the state is "tangible", i.e. the thread "blocks" in the monitor (Java thread *passive* state), or
- the state is "volatile", i.e. thread continues its execution without blocking (i.e. the thread keeps itself in Java *runnable* state).

Every consistent state of a Java thread can be then described as follows:

$$\begin{aligned}
thread_id &= \{thread_class_name, thread_instance_num\} \\
thread_state_id &= born \mid monitor_entry \mid dead \\
thread_state_attr &= set_of_thread_local_data_values \\
thread_state &= \{thread_id, thread_state_id, thread_state_attr\}
\end{aligned}$$

A transition between thread consistent states can be described as a tuple of transformations:

$$\begin{aligned}
next_state_id &: thread_state \rightarrow string \\
next_state_attr &: thread_state \rightarrow attr_type
\end{aligned}$$

It is necessary to explain that the abstract functions *next_state_id* and *next_state_attr* work with the set of a thread's local data and with the set of a monitor's data (the reference to a monitor is a part of *thread_state*). A body of this abstract function consists of the body of the monitor method and from the thread code part following the monitor method call.

Now we can proceed to a description of a discrete state space of a concurrent Java program computation, also called the global state space:

- Every thread has its own *coordinate axis within the space*, so the state-space dimension equals to n.
- Every (discrete) state-space coordinate axis has a limited mumber of points (nodes), that correspond to the thread state identifiers (*thread_state_id*).
- State-space then can be viewed as a bounded regular grid of nodes within an n-dimensional space.
- *State of the computation* can be defined as a point (node) within the grid, i.e. as the set of single threads state identifiers.
- *Control-flow of the computation* can be modeled as an oriented graph whose nodes are all the points of the lattice. From every node at most

  $$\sum_{i=1}^{n} PT_{i,thread_state(i)}$$

  transitions can lead to other nodes. Here n is the total nuber of threads (and therefore the dimension of the state space and the number of axis of the grid) and $PT_{i,thread_state(i)}$ is a set of possible transitions of a given thread from its current state to its neighbouring states – see section 3.3.1. This set usually contains more than one transition due to program branches, cycles, etc. This will be discussed in more detail in section 3.3. Every transition means a finished thread local activity, i.e. a point when

the thread reaches a (new) consistent state. Due to the fact, that a transition means a single thread activity, only one state-space coordinate value changes with this transition.

- *Path of computation* is a sequence of transitions that an instance of computation passes through the control-flow graph of computation.

For a concurrent Java program executed by the JVM, the following behavior of computation can be observed:

- State of computation that is modelled by means of a state-space node is not consistent in the sense, that its parts (coordinates values) do not represent neither threads' nor monitors' consistent states. A value say s_1 at t_1 axis means, that thread t_1 either resides in its consistent state identified as s_1 or passed the state and has not reached a next consistent state yet.
- So in every state of concurrent computation several transitions "have fired" (runnable threads computation) and it is generally impossible (due to the data dependency and due to an unpredictable speed of a thread computation caused by the nondeterministic JVM scheduler and the environment outside the JVM) to forecast which thread will be "the winner of the race", i.e. which state of computation will follow. Moreover it gives no great sense to stop the computation in a state that is not consistent.
- As a (known) consequence follows that the path of computation is quite nondeterministic. It causes majority of known troubles with concurrent programs testing and debugging.

The main idea of a concurrent Java program serialization is to choose deterministically for every node of the above defined state-space of concurrent computation only one thread (from the set of runable threads) to perform its transition.

The resulting path of computation will be one of possible computation paths, therefore it will reflect a possible real execution of the program. The basic idea of the proof is stated in [ExBlk1] and is as follows: *Shared variables cannot be accessed outside of synchronized blocks (if the program follows the locking discipline, as stated in 3.1) and therefore a thread cannot influence another thread's behavior during its local computing block, only inside a monitor method. Therefore, enumerating all possible orders of all monitor procedures results in enumerating all possible behaviors of the program.* It is not then necessary to order the local computing blocks, because:

1. A local computing block of a thread has no influence on another thread's behavior, and
2. The order of local computing blocks (i.e. the order of transitions in the state-space graph) is fully determined by the order of monitor procedures (i.e. the order of global consistent states in the state-space graph) that they lead to.

An implementation of the idea should bring the following important consequences:

- The computation can be stopped in every node of its state-space (and the transition to be performed can be chosen).
- When the computation has stopped in a node, the corresponding state of computation is consistent, i.e. all monitors and all threads reside within their consistent states (which means that these states can be observed and/or analyzed).
- The path of computation can be made deterministic, i.e. every possible path through the control-flow graph of computation can be chosen either automatically (i.e. according a rule) or manually during a “manual” debugging of the program. This is of great importance in a process of Java concurrent program debugging.

3.3 Walking through the State Space

In this section, we will show why a global consistent state does not have just n[1] neighbouring states, as one would think. Actually, if we omit the final consistent states symbolising that a thread is dead, the number of neigbouring states can vary from n in the simplest case to

$$\sum_{i=1}^{n}(CS_i - 1)$$

in the most complicated case. CS_i denotes the number of all consistent states of the ith thread. This number is decremented by one because no thread can return to its initial state which corresponds to the thread’s creation. Therefore, we can state that the following expression

$$n \leq \sum_{i=1}^{n} PT_{i,thread_state(i)} \leq \sum_{i=1}^{n}(CS_i - 1)$$

holds in any of the global consistent states except for all global states composed from at least one final state (when at least one thread is dead already).

3.3.1 The Relation of Neighbourhood

It can be found from the above statement that the definition of the term “neighbour” is not a trivial task. We will define it as follows: *Let t be a thread in a consistent state s. A constistent state s_i of the same thread t is a neighbour of the consistent state s if such an execution of the thread t’s code exists which leads from s to s_i and this execution does not contain any t’s consistent state s_j, $s_j \neq s_i$.*

It can be easily seen that this relation of neighbourhood:

[1] n is the number of threads in the program, see 3.2.

- is *not reflexive*, because a consistent state need not necessarily be reachable from itself,
- is *not symmetric*, because the fact that a consistent state B is reachable from a consistent state A does not imply A to be reachable from B,
- is *not antisymmetric* since B can be a neighbour of A and A can be a neighbour of B and the two consistent states A and B need not be identical,
- is *not transitive* since the definition disallows it from the very beginning.

As for the global consistent states, the definition of the neighbourhood relation may look like this: *Let S be a global consistent state. Then S_i is a neighbour of S if there exists a thread t such that the sub-state of S_i with regard to t is a neighbour of the sub-state of S with regard to t in t's code.*

In the following paragraphs, a few examples of the neigbourhood relation together with examples of the corresponding Java source code will be given in order to illustrate some possibilities how the computation path can be constructed. This list is not complete, any other possible configuration can exist. Moreover, these possibilities can be combined, i.e. a diagram from the following sections can be put at an arbitrary place (between two consequent nodes) of another diagram and the result will be a valid state-space model diagram of a part of code.

3.3.2 Simple Linear Neighbourhood

This is the simplest case of consistent states neighbourhood. A consistent state just follows another one in the oriented graph, as shown in figure 3.1.

Figure 3.1: Simple Linear Neighbourhood of Thread Consistent States

The corresponding Java source code would look like this:

```
Local computation
A call to a monitor method
Local computation
A call to a monitor method
Local computation
```

3.3.3 Conditional Neighbourhood

Since the computation path of (almost) every program is data-dependent, a consistent state can be entered when a condition holds and avoided otherwise. This is shown in figure 3.2.

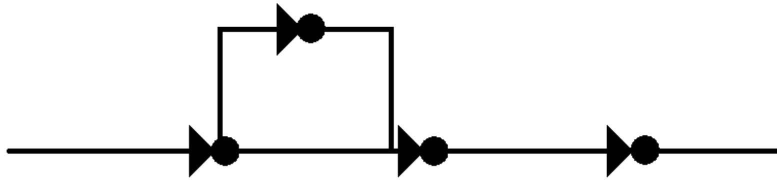

Figure 3.2: Conditional Neighbourhood of Thread Consistent States

An example of Java source code:

```
Local computation
if (condition statement) {
    Local computation
    A call to a monitor method
    Local computation }
Local computation
A call to a monitor method
```

3.3.4 One-of-N Choice Neighbourhood

A more general case of the previous one is when one (or none) of N consistent states can be chosen to be entered. This can be accomplished by a `switch` statement with different consistent states in some branches.

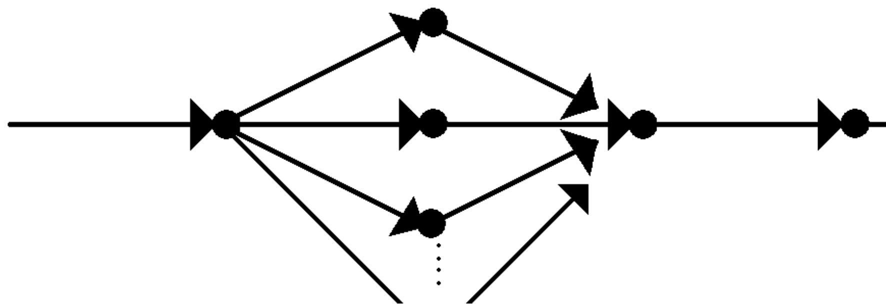

Figure 3.3: One-of-N Choice Neighbourhood of Thread Consistent States

An example of Java source code:

```
Local computation
switch (expression) {
    case C1: Lcl cmp; A call to a mon method; Lcl cmp; break;
    case C2: Lcl cmp; A call to a mon method; Lcl cmp; break;
    case C3: Lcl cmp; break; }
Local computation
A call to a monitor method
```

3.3.5 Cyclic Neighbourhood

If several consistent states are placed inside a cycle, they can be repeatedly entered in the same order. This is show in figure 3.4.

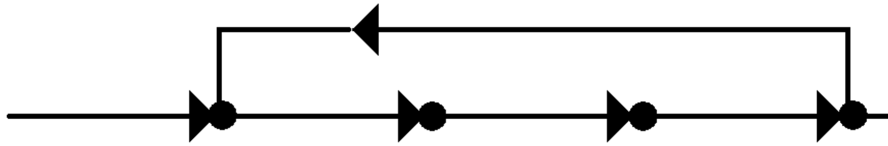

Figure 3.4: Cyclic Neighbourhood of Thread Consistent States

In Java, it would be expressed like this:

```
Local computation
while (condition) {
    Lcl cmp; A call to a mon method;
    Lcl cmp; A call to a mon method;
    Lcl cmp; A call to a mon method; Lcl cmp; }
Local computation
A call to a monitor method
```

3.3.6 Special 1-State Cyclic Neighbourhood

A consistent state can also be neighbour of itself, if it is placed inside a cycle. This is a special case of the previous one, when just one consistent state can be reached inside the cycle.

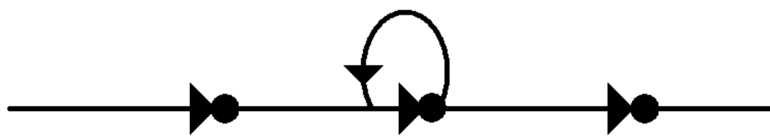

Figure 3.5: Special 1-State Cyclic Neighbourhood of Thread Consistent States

```
Local computation
while (condition) {
    Lcl cmp; A call to a mon method; Lcl cmp; }
Local computation
A call to a monitor method
```

3.3.7 One-of-N Choice Cyclic Neighbourhood

If 3.3.4 and 3.3.5 are merged together, i.e. a switch statement is placed inside a cycle, we get the situation depicted in figure 3.6.

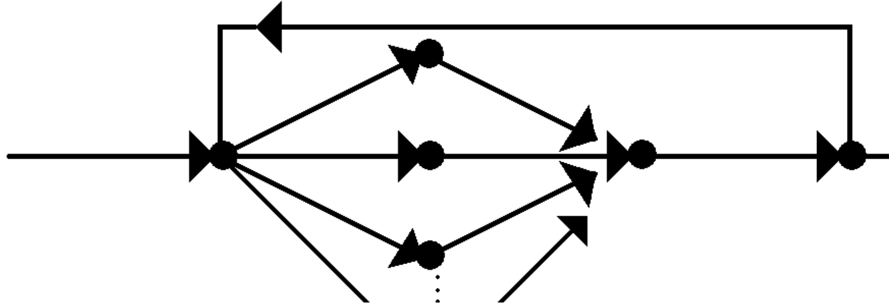

Figure 3.6: One-of-N Choice Cyclic Neighbourhood of Thread Consistent States

It is obvious how the corresponding Java source code will look like – inside a cycle there will be a `switch` statement with consistent states in some branches and some other consistent states will be placed outside the `switch` block but inside the cycle.

3.3.8 All-to-All Neighbourhood

If we keep all the consistent states located in branches of the `switch` block but remove the consistent states outside of it (and inside the cycle), we get the situation from figure 3.7.

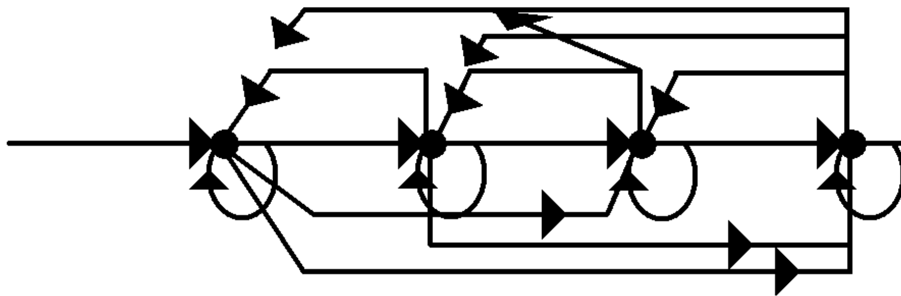

Figure 3.7: All-to-All Neighbourhood of Thread Consistent States

Because the cycle can be repeated at least twice, any of the states in question can follow after any other, the selection of transition depends only on the expression used it the header of `switch` and on the condition in the cycle's header.

3.4 An Example: The Producer-Consumer Problem

Let's take a real example of a concurrent program now and show how its state-space model, created according to the previous section, will look like. The example is a classic producer-consumer problem, realized in Java. Two threads (one producer and one consumer) share one monitor which has a limited amount of memory (let's say an array of integers of length N) for storing data exchanged between the producer and the consumer. The threads do not access this memory directly but use synchronized methods of the monitor.

The producer repeatedly invokes the monitor's `write()` method which inserts an integer to the shared array. However, if the whole capacity of the array is already used, the producer must passivate itself inside the `write()` method using `wait()` until the consumer frees at least one slot of the array. On the other hand, the producer must assume that the consumer is also potentionally passivated (if the array is empty) and, therefore, it must activate the consumer in the case when the number of occupied slots of the array changes from 0 to 1. Since there is always just one waiting thread in this case, the producer calls the `notify()` method of the monitor. So the producer's program may look like this:

```
public void run() {
 do {
     monitor.write((int) (Math.round(Math.random() * 100.0)));
} while (Math.random() != 0.0); }
```

Normally, the producer-consumer algorithm is an infinite one. To demonstrate the usage of the model, we turn the algorithm to finite by allowing both the producer and the consumer to finish after at least one number is written/read to/from the shared array. We will use random numbers to accomplish that.

The consumer repeatedly invokes the monitor's `read()` method which takes out an integer from the shared array and returns it to the consumer. Again, the consumer can passivate itself if it cannot read from the shared memory, i.e. when all the slots of the array are free. The consumer stays passivated until the producer writes at least one integer into the array. The consumer must also assume that the producer may have been passivated and it must try to reactivate the producer using `notify()` when the number of occupied slots goes from N to $N-1$. The consumer's code is as follows:

```
public void run() {
 int x;
 do {
     x = monitor.read();
} while (Math.random() != 0.0); }
```

The state spaces of both threads are almost identical. They have five consistent states each: one initial state, one terminal state, one state for entering the monitor, one state for leaving the monitor and one state for passivation inside

the monitor methods by `wait()`. The two state spaces are depicted in figures 3.8 and 3.9.

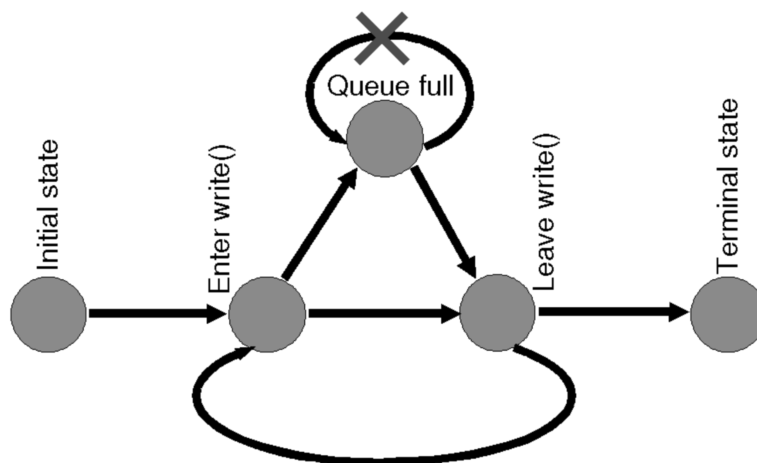

Figure 3.8: State-Space Model of the Producer

We also assume that the activation signals are sent only when it is necessary, therefore the threads will not enter the state "Queue empty" or "Queue full" as soon as they leave it. (That would be because every guarding condition should be evaluated in a `while` cycle, not just in an `if` statement.)

We should also show how the monitor methods will be implemented:

```
public void synchronized write(int i) {
 while (occupied >= N) wait();
 // Writing down i to the array
 occupied++;
 if (occupied == 1) notify(); }

public int synchronized read() {
 int temp;
 while (occupied <= 0) wait();
 // Reading temp from the array
 occupied--;
 if (occupied == N-1) notify();
 return temp; }
```

Now we can finaly proceed with construction of the global state-space model of the whole program. It can be constructed by using superposition of the two

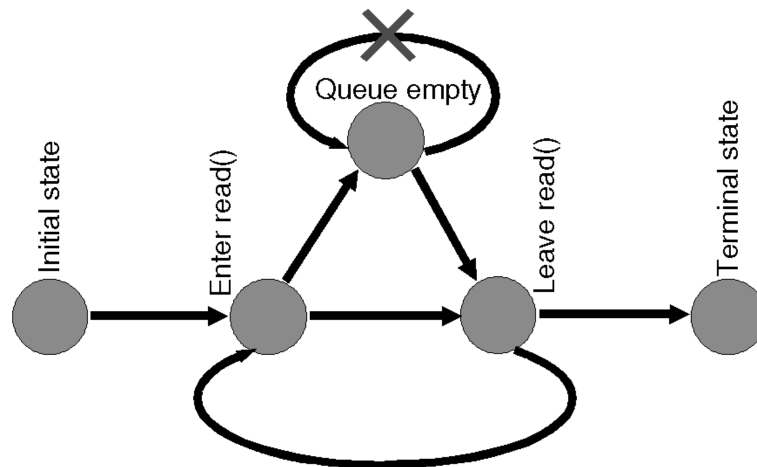

Figure 3.9: State-Space Model of the Consumer

independent models, i.e. every consistent state of the producer is paired with every consistent state of the consumer which results in a 2-dimensional grid of global consistent states of the whole program shown in figure 3.10. Actually, the grid should be 3-dimensional but we ignore the main thread that just creates and starts the producer and consumer threads and dies afterwards. The real model would not have 25 but 50 states, which could be imagined as two grids from image 3.10 put one above the other, whose corresponding nodes are connected with an arrow (a transition between consistent states).

The model has 25 (5×5) consistent states, some of which are not reachable. The x-axis belongs to the producer thread, the y-axis to the consumer thread. An *unreachable* state or an *unrealizable transition* is red-crossed. Note that it would be extremely hard or even impossible to detect unreachable states and unrealizable transitions. In this case, they are marked thanks to the fact that we exactly know the threads' behaviors and their relationship from the code shown above and from the informal description of the algorithm.

The serialized producer-consumer program starts computation in the upper left consistent state with meaning "no thread has run yet". Any of the two threads can be then chosen to run, therefore two other states are possibly following the first one. In a different consistent state, more than two other states can be reached, which reflects data dependencies in the program (random numbers and the state of the shared array).

A state is unreachable if there exists no realizable transition leading to it. For ex-

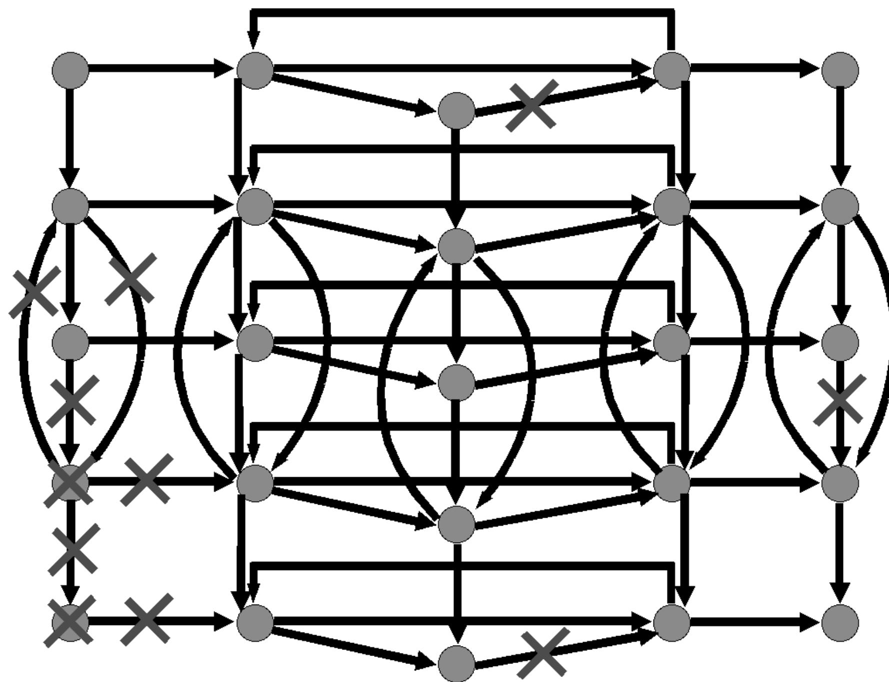

Figure 3.10: State-Space Model of the Producer-Consumer Program

ample, the third from top and first from left state is not reachable because both transitions leading to it are unrealizable. The first one would correspond to the situation when the consumer successfully completes reading but the producer has not run yet, therefore the shared array must be empty. That is an obvious nonsense. The second transition would correspond to the situation when the blocked consumer is reactivated and successfully completes reading from the empty array. This is also nonsence, because it would have to be reactivated by the producer which has not run yet.

If a consistent state is unreachable, all transitions leading from it are unrealizable.

Just note that *deadlock can happen* in the program. This is due to the possibility of both threads to finish after writing/reading at least one number, introduced artificially to reach the threads' final states. If the producer decides to finish in the moment when the consumer has just read the last available integer number from the array and attempted to read again, the consumer will stay blocked forever. Conversely, if the consumer decides to finish in the moment when the producer has just written an integer to the last available slot and attempts to write again, the producer will stay blocked in the monitor's `write()` method forever. The possible deadlock is expressed by the two unrealizable transitions on the right and at the bottom of the graph.

The last unrealizable transition corresponds to the situation where the producer first passivates itself because of the full array, then is reactivated and completes

writing, without waiting for the consumer to free at least one flot in the array. Again, this is an impossible behavior.

3.5 JVM-like Entities and Their Relationships

When looking at a Java multithreaded program from a point of view that is closer rather to the level of the Java virtual machine than to the high-abstraction level of Java source code, it can be found that we must take into account entities that are not used or mentioned in the source code at all. These entities are as follows:

Threads – Here, the word "thread" has its proper meaning known from the upper level, i.e. a thread is an instance of the class `Thread` (or its subclass). A thread has a state which is one of the following: *new*, *runnable*, *not runnable*, *dead*. A thread can be associated with one or more locks that it is currently holding.

Locks – Any object (instance of any class, this excludes primitive types) that is used for synchronization in the source code. Actual Java objects are used here, not just the invisible and inaccessible 1-bit value hidden inside every object that is actually used for synchronization. At most one thread can acquire one lock at a given time, but it can do it recursively. Two operations are provided by locks that threads can use: `lock()` and `unlock()`. The `lock()` operation assigns the lock to the calling thread if the lock is not currently acquired by another thread, otherwise the calling thread is blocked and put into the lock's delayed set of threads. The `unlock()` operation marks the lock free for use by other threads if the delayed set is empty or (if it is not) takes an arbitrary thread from that set and assigns the lock to it. Every lock should also provide a method replacing the behavior of original Java methods `wait()`, `notify()`, and `notifyAll()`.

Wait Set – A set of threads associated with every lock. It stores threads that have invoked the `wait()` method of the lock (every invocation of `wait()` has its lock object, even if it not explicitely stated in the source code) and can be potentially resumed later when the `notify()` or `notifyAll()` method of the same lock is called.

Delayed Set – A set of threads associated with every lock. It stores threads that have tried to enter a region of code synchronized with its respective lock. This set is managed by the `lock()` and `unlock()` operations.

Join Set – A set of threads associated with every thread. Threads in the set are temporarily suspended, until the owner of the given join set (a thread) dies. Then, all threads from this set are resumed and excluded from the set.

The locks are key elements in a Java concurrent program, since everything (thread switching, suspension, reactivation, . . .) depends on their state. Let's discuss the behavior of locks in more detail now. Every method in the following

list manipulates with the calling thread's state and the lock's wait set and delayed set:

- `lock()` first tests if the owner of the lock is `null`. If it is, the lock is marked as "acquired" and the calling thread can enter the protected region of code that follows. The owner must also be stored. If the owner is not `null`, the calling thread is inserted to the delayed set and suspended. It can be later reactivated by the `unlock()` function. The wait set is not touched at all in the `lock()` function.

  If `lock()` is invoked recursively (i.e. one thread invokes the `lock()` method of the same lock after invoking the same lock's `lock()` method without invoking `unlock()` in between them), the thread "may pass" which means that it is neither suspended nor inserted to the delayed set, although the lock owner is not `null`. A counter associated with every lock must therefore exist, keeping information about the current lock owner's recursion level.

- `unlock()` relinquishes the lock and enables thus other threads to succeed during a call to `lock()`. The owner is set to `null` first. Then, if the delayed set is not empty, an arbitrary thread is taken from there and assigned to the lock as if it won the competition during the lock acquisition. Others are left untouched. Different strategies of the "winner" selection can be applied here, as mentioned in section 4.2. If the delayed set is empty, nothing happens, the lock is ready to be assigned to any other thread.

  All this is performed only if the recursion-level counter reaches zero. Otherwise, the counter is decremented and the thread keeps the ownership of the lock.

- `wait()` is the lower-layer counterpart to the classic `wait()` method used in Java source code. First, it calls `unlock()` in order for the calling thread to relinquish the lock. Then, it suspends the calling thread and inserts it to the wait set of the lock. The delayed set is not touched at all. The thread remains suspended until the lock's `notify()` or `notifyAll()` method is called, or the thread's `interrupt()` method. When one of these possibilities happens, the thread is taken out of the wait set and the `lock()` function must be called. That's because of the fact that the thread must compete with other threads in the usual way the get the lock upon exit from the `wait()` method. If it is not successful, it stays blocked in `unlock()`, being inserted in the delayed set. `wait()` changes the calling thread's state to *not runnable*.

- `notify()` selects an arbitrary thread from the wait set and reactivates it. The thread then removes itself from the wait set and continues running inside `wait()` where it got suspended before. As stated above, it can get blocked again inside `lock()` whose invocation is mandatory. The delayed set is kept unchanged. If the wait set is empty, nothing happens. `notify()` changes the calling thread's state to *runnable*.

- `notifyAll()` is identical with `notify()` except for one difference: All threads from the wait set are reactivated, not just one. This means that

all of them immediately start competing for the lock but at most one of them can get it (or none if it is currently held by another thread) so most or all of them are going to get blocked again in `lock()`. However, they are going to be inserted to a different set: to the delayed set instead of the wait set. `notifyAll()` changes the calling thread's state to *runnable*.

3.6 Interesting Entry Points to the JVM

Now when we have a brief understanding of what the model of a concurrent Java program should deal with, we should enumerate the points of Java source code where the program interacts with the JVM scheduling mechanism. In order to replace this mechanism with our own, we must:

1. Properly detect all such places in the program's source code;
2. Replace all found pieces of source code with code interacting with our model;
3. Re-implement all functionality of the original JVM scheduler to allow us to monitor and controll the program's behavior, but without negatively affecting the program with inappropriate changes of behavior that are not normally possible.

The interesting places in source code are discussed in the following sections.

3.6.1 Beginnings of Synchronized Blocks

The beginning of a `synchronized` block must be replaced with a call to the `lock()` method of the lock used in the program. The lock can be either explicitely denoted by the programmer (if it is a `synchronized block` inside a method) or it the object whose `synchronized` method is called. In the latter case, the `synchronized` keyword is a part of the method's header. In the case of a static method, the lock of the class must be used.

If `synchronized` blocks are nested, `lock()` must be called for every block and the lock object must be evaluated independently for every call.

3.6.2 Ends of Synchronized Blocks

The end of a `synchronized` block must be replaced with a call to `unlock()` of its respective block. The rules for finding the lock are the same as in the case of `synchronized` blocks beginnings. It should be mentioned here that in the case of `synchronized` methods, the opening and closing braces are used both as the method's delimiters and as the `synchronized` block's delimiters.

If `synchronized` blocks are nested, `unlock()` must also be called for every block, with its corresponding lock object.

3.6.3 Calls to `wait()`

Any `wait()` invocation must be replaced with a call to the respective lock's `wait()` method, described in section 3.5. Again, it is necessary to find the right lock object. It can be explicitely stated in the code (`aLock.wait()`) or the rules mentioned above must be used.

3.6.4 Calls to `notify()` and `notifyAll()`

The `notify()` and `notifyAll()` methods must be mapped to the methods `notify()` and `notifyAll()` of their respective lock, in the same manner like in the case of `wait()`. Again, two possibilities exist: either it can be explicitely stated in the source code or it must be found.

3.6.5 Calls to `destroy()`

There are two possibilities how this method could be reimplemented in the model of the program:

1. The thread could be marked as "non-existing" and excluded from further scheduling. All the locks that it has currently acquired must remain acquired, which will very likely produce a deadlock in the modelled program (not in the modeling system). This scenario corresponds to the ideal situation in which the `destroy()` method would be implemented according to its specification.

2. According to the Java API documentation, this method is not implemented and an exception is thrown out when this method is invoked. Therefore, the program could cut off the running thread (not the thread whose method is called), warn the user, free all resources held by the calling thread and continue with scheduling the rest of threads. This scenario corresponds to the actual current behavior of the Java virtual machine.

3.6.6 Priority Changes

As it is in the case of a real JVM, every thread must be assigned an integer number that corresponds to its priority. The scheduling mechanism must take into account these priorities. If there are more runnable threads at the moment, the chosen thread will be that with higher priority, if the priorities are not equal. If they are equal, other conditions (mainly conditions of the scheduler itself, see section 4.2) will influence the choice.

`setPriority()` should be replaced with a function changing the calling thread's priority inside the controller, `getPriority()` with a function returning it.

3.6.7 Calls to `interrupt()`

A flag must exist for every thread, saying whether the thread has been interrupted or not. In a normal case, the replacement of the `interrupt()` call only sets this flag to `true`. However, if the thread in question is in a wait set of a lock, it must be resumed (as if the `notify()` function of its current lock were invoked) and a new instance of `InterruptedException` must be thrown out from the lock's `wait()` method.

3.6.8 Calls to `interrupted()` and `isInterrupted()`

The replacements of both functions only return the flag of interruption for a given thread. The only difference is that `isInterrupted()` must reset this flag back to `false` while `interrupted()` must keep this value unchanged.

3.6.9 Calls to `join()`

In Java, a thread may temporarily suspend itself until another specified thread dies (reaches the end of its `run()` method and changes its state to *dead*). All such threads are stored in the join set (see 3.5) of the given thread and are blocked. When the given thread dies, all the threads from its join set are resumed (their state is changed back to *runnable*) and are ready to be scheduled from now on.

The replacement of `join()` should suspend the calling thread (if the other thread has not died alredy) and put it to the join set. The suspended thread should be resumed by the operation that executes at the end of every thread's life – see section 3.6.15.

3.6.10 Thread Suspensions and Resumes

Although these methods have been deprecated for several years, it is still possible that they will be used, therefore, the controller of the model should be able to handle them.

It would be quite a hard task to model the behavior of the real `suspend()` method since the target thread can be suspended at any point of its code. Fornunately, we can profit from the principles introduced in sections 3.2 and 3.3. When this method (or rather its replacement action of the controller) is called, the target thread is always blocked by the controller inside another replaced entry point. Therefore, it is enough to set the suspension flag to `true` and the thread will be excluded from scheduling from now on until the replacement function of `resume()` is called, that changes this flag back to `false`. During `suspend()`, the ownership of locks must remain.

3.6.11 Calls to `stop()`

This method is also marked as deprecated, although it can be used without problems. Again, the target thread can be stopped at any point of its code

which would normally cause hard problems if needed to be properly modeled. We can use the same trick as in the case of the `suspend()` replacement, but the flag must not be set to `true` since even a stopped thread can execute. The ownership of all locks must be cancelled. In addition, a new `ThreadDeath` exception must be thrown out to truly simulate the original behavior.

3.6.12 Calls to `setDaemon()`

Every thread must have a flag saying whether the thread is a daemon or not. There are no differences when scheduling daemons, except for the situation when there are no threads but daemons to be scheduled. In such a case, the controller should terminate all threads and finish.

The replacement of `setDaemon()` should simply set the flag to `true` or `false`.

3.6.13 Sleeps

Since it is impossible to guarantee that a thread using the `sleep()` method will be suspended and woken up later after exactly the specified amount of time elapses, two possible approaches how to model `sleep()` seem to be reasonable:

1. Calls to `sleep()` will not be replaced at all. The calling thread will really sleep for the specified amount of time. This approach does not truly correspond to reality since other threads are not allowed to run during the waiting.
2. The calling thread is temporarily suspended (its activity flag is set to `false`) and in every global state of the system (see 3.2), the controller checks whether the specified amount of time has elapsed. If it has, the activity flag of the thread is restored. This approach does not correspond to reality in the sense that the calling thread will very likely sleep longer than it wanted.

3.6.14 Calls to `yield()`

Since it is not exactly specified how the `yield()` method should work, several different approaches are again possible:

1. The calling thread can be suspended (using the activity flag) for the next step, i.e. until the system gets to the next global state.
2. The calling thread can be suspended until all other runnable threads get a chance to run. This may be very unrealistic since there can be a small set of threads that will not run for a long time.
3. The calling thread can be suspended until all runnable threads with priority higher than or equal to the priority of the calling thread get a chance to run. Again, this approach is not much realistic.

4. `yield()` could be modeled as `sleep()` with a reasonable value of the argument. The problem is to find the reasonable value.

5. `yield()` could be ignored completely.

3.6.15 Thread Creations, Startups and Deaths

The controller must be aware of any thread present in the modelled system. It is therefore crucial for it to get information about every thread creation, startup and death. It is not enough to monitor just the startups and deaths, since some methods of a thread can be invoked even before it is started, for example the `join()`, `stop()`, `suspend()`, and `interrupt()` methods.

The act of a thread creation can be noticed inside its constructor. It would also be possible to check for the `new` operator, however, there are many more occurencies of `new` in the source code than occurencies of a thread constructors (just one per a thread class, or more if the constructor is overloaded).

When a thread is started, its `run()` method begins to be executed. So it is quite sufficient to focus on the beginning of the `run()` method, calls to the `start()` method can stay untouched. At the beginning of `run()`, the controller must get informed that a new thread has just been started and must include it to the set of runnable threads ready for scheduling.

At the end of `run()`, just before the thread dies, it must "log-out" itself from the system and tell thus the controller not to schedule it anymore. Also, all threads from the dying thread's join set must be resumed.

Since the `main()` function of the program is functionally equal to the `run()` method of a thread, the same principles should be applied to its beginning and end.

Chapter 4

The Controller

In the previous chapter, a formal model of a serialized Java program was defined. However, the specified behavior, corresponding to the model, must be imposed to the program somehow. The existence of the model itself does not guarantee anything since its rules defined in 3.2 are not followed by a concurrent program running on top of a classic JVM.

The correct (defined) behavior could be achieved by means of a special Java virtual machine, used instead of the classic one. The virtual machine would then take care about all problematic places mentioned in 3.6 and hadle them appropriately. This special JVM would be used only for testing purposes and it would be replaced later when the tested program is trusted to be free of errors. This approach is used in [ExBlk1] where the *Rivet JVM* is used and in [PF2] where *MC-JVM* is used. Other JVM exist, e.g. the *Java-in-Java Virtual Machine*, written itself in Java.

A different approach is used in this work. A classic "commercial" JVM is used, but a layer in a form of several Java classes is inserted between the running program and the JVM. This layer controls the execution of every single thread in the program (as described in section 4.1) and therefore is called the *controller*. It is just here where all nesessary scheduling-related algorithms conforming to the defined model's behavior are implemented. The structure of a serialized program is shown in figure 4.1.

As it can be understood from the picture, the controller does not intercept new thread creations and startups. But it must intercept all other calls to the JVM already enumerated in section 3.6. This does not involve only calls to methods but also normally "invisible" calls to the JVM, such as entering a synchronized block. For deterministic thread switching, the controller internally uses services of the JVM that the serialized program is not allowed and capable to use.

The controller exposes some methods for "public use" – it is so called *interface of the controller*, available for the serialized program. All the scheduling and thread-switching algorithms are hidden inside, inaccessible. The controller classes are grouped in a package to allow easy import.

Of course, the existence of the controller does not force the program to use its

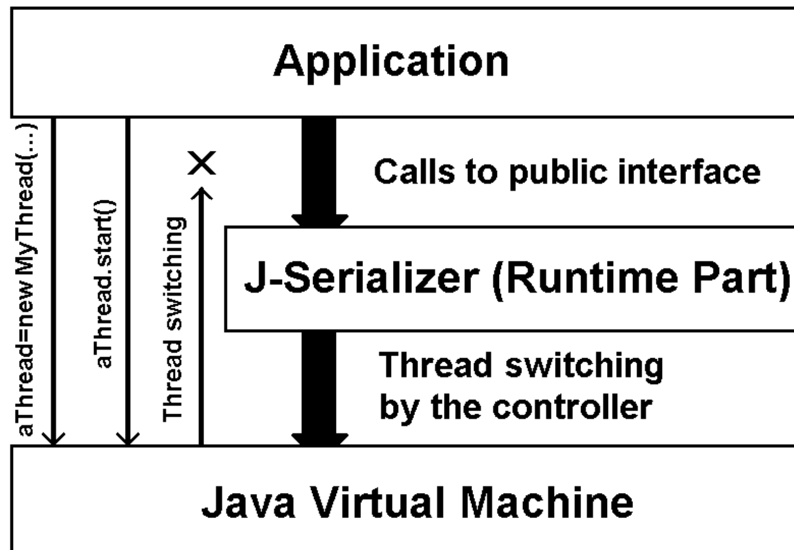

Figure 4.1: The Controller – A New Layer Inserted between the Program and the Java Virtual Machine

services and to abandon the standard ones. Therefore, the program must be converted from its original form to a form aware of the controller and free of any forbidden JVM calls. Chapter 5 discusses the conversion process.

4.1 Forcing the Execution Order

Since the model presented in chapter 3 requires that exactly one transition – a transition of any enabled thread – is allowed to be executed from the global consistent state that the system currently resides in, it must be assured by the controller that exactly one thread of all possibly executing threads is running at any time. Moreover, it must be assured that the controller (or rather the scheduler) is able to deterministicly select the next thread to run and that this thread will be allowed to run only to the next global consistent state. Whenever a consistent state is reached, the currently running thread must suspend itself and allow the scheduler to select a different thread according to its preferences.

After having been started using its `start()` method, every thread runs completely independently from the others as well as from the thread that created and started it. According to the number of real processors, either one or more threads can run at a given time. If there are more threads than processors, the classic JVM switches them in a unpredictable way. In order to switch them deterministicly and replace thus the JVM's natural behavior, *the controller's internal methods of synchronization* must be used.

The `Thread` class provides three methods allowing to manipulate with a thread's

execution state: `suspend()`, `resume()`, and `stop()`. The `suspend()` method temporarily suspends execution of a thread until `resume()` is invoked on the same thread. The last method interrupts execution of a thread and causes it to exit its `run()` method. However, all of them have been marked as deprecated since JDK 1.2. There were very serious reasons to do so, mainly concerning security. The reasons can be found in [Lea]. So the controller does not use these methods at all.

Instead, the controller uses synchronization, `wait()`s and notification, classic means for implementation of monitors and guarded actions. Every thread that reaches a consistent state and calls a public method of the controller first modifies some properties of the entities of the model (such as wait sets, delayed sets, join sets, a thread's state, ...) and then enters a `synchronized` block (or rather a `synchronized` method) where all the thread switching takes place.

The deterministic switching is then possible thanks to the following factors:

- A common lock, available to all threads, exists. It is the lock that the global switching method is synchronized with.
- Every thread in the system has a unique ID number. The controller keeps information about the currently running thread in variable called `runningThread`. This variable is updated whenever the scheduler selects a new thread to un. If no thread is running, the constant `NO_THREAD` is set.

Let's have a look now at how the switching process is implemented in Java. A piece of code of the global switching method is shown here.

First of all, the ID number of the currently running thread is stored to the variable `myThread`. It is important to save this value before it changes inside the method as it will be needed later when a signal is received from another thread that wants to activate a passive thread. Only the controller keeps this information, not the threads themselves. Since the switching can take place in any object (including all non-threads), the threads would have to pass this information as parameters to every method of a non-thread which would be inefficient and hard to add to the source code during the conversion process.

```
myThread = getRunningThread();
```

Right after that, the scheduler is informed that the thread is getting suspended.

```
setRunningThread(NO_THREAD);
```

Then, it is necessary to select the next thread to run, respecting the criteria of the scheduler. The ID number of the selected thread is stored into variable `runningThread`. It is purely a matter of the scheduler (which in turn is a subpart of the controller, not visible from outside) to select the next thread to execute. The controller has already been informed that a consistent state had been reached (and which consistent state it was, too) and all necessary information that the selection of the next thread depends on has been updated.

```
setRunningThread(scheduler.getNextThreadToRun());
```

Then, all threads present in the system, suspended with the same lock, are woken up by a signal sent by `notifyAll()`. The calling thread is not affected in any way and continues normally. It is necessary that `notifyAll()` be placed before the thread suspends itself. Otherwise, the waking-up signal would never be sent and all threads would remain passive forever.

```
notifyAll();
```

Finally, the thread can passivate itself using `wait()`. When it is later woken up by a signal sent by `notifyAll()` from another thread, it compares its number, saved at the very beginning, with the number set by the scheduler. If the numbers are identical, it leaves the `while` cycle and returns from the switching method and then immediately from the controller's method representing a certain type of consistent state[1]. Otherwise, the thread passivates itself again and waits for another signal sent during next switching when the selected thread reaches a next consistent state.

```
while (getRunningThread() != myThread)
{
 try { wait(); }
 catch (InterruptedException e)
 { System.err.println("Error in controller!"); }
} // while
```

4.1.1 Possible Optimization

The switching mechanism presented above is not ideal in the sense that during thread switching, all threads are woken up and suspended right after, not only the selected thread. This mechanism puts additional load on the simulator of the program and uses processor time inefficiently. However, it can be improved in such a way that only one thread will wake up.

The basic idea is as follows: Every thread has its *own lock*. When a thread suspends itself inside the switching method of the controller, it uses that lock. When another thread comes to the switching method and has to wake up the first thread, it first gets the ID of the selected thread (as before) but also gets its lock. Then, the running thread enters a block `synchronized` with this lock and invokes `notify()`. *This wakes up just one thread – the selected thread.* The thread woken up then exits the `synchronized block`, exits the switching method, exits the method representing a global state and proceeds as the only running thread in the system. The originally running thread exits the block `synchronized` with the other thread's lock and *enters another `synchronized` block, now synchronized with its own lock.* Then, it suspends itself using `wait()` until it selected by the scheduler and woken up by another thread.

[1] An entry to a monitor method, a passivation inside a monitor, ...

4.2 The Need for Various Types of Scheduling

Even if all principles of Java thread scheduling (priorities, thread states, ...) are respected, there is usually a number of runnable threads with the same chance that can run from a given global consistent state. However, only one of them can be selected by the scheduler to pass through its next transition. Therefore, the scheduler must have an algorithm to decide which thread will be given priority.

Since the behavior of a standard JVM is not defined anywhere (at least not in [JVMS]), the scheduler can implement several different algorithms that can all be used during testing. The different algorithms can be, for example:

Random Switching – The next running thread is selected randomly from the set of all runnable threads. This is probably the most similar to the implementation of scheduling algorithms in real JVMs. (But it cannot be said for sure.)

The Last Recently Run Thread Algorithm – The next running thread is such a thread from the set of all runnable threads that ran the longest time before the moment when the new scheduling decision takes place. Therefore, the threads are scheduled with a certain period, which is equal to the number of threads, if all threads are in runnable state.

Round Robin Algorithm – All threads are ordered in a list. The scheduler jumps from a thread to its successor. If it reaches the end of the list, it jumps to the beginning. All not-runnable threads are ignored when their turn comes. The main difference between this algorithm and the last one is that a thread that has been suspended for a very long time and resumed recently will have a very big chance to run in the last case but its chance to be scheduled, using the round robin algorithm, depends on the current pointer to the list of threads and therefore is equal to the chances of all other threads.

Time Measurement Algorithm – The real time spent by every thread in the system is measured and the thread with the least time of all is selected. This approach seems to be problematic and not very usable in some cases since the time measurement function `currentTimeMillis()` does not return the real time with the precision of one millisecond but is hardware-dependent. So short sequences of code are reported to be executed in zero time, which is a problem.

Modified Time Measurement Algorithm – The last strategy can be combined with the random switching strategy. As a result, the algorithm would compute a random number, using the measured real time as a parameter, for example random numbers with exponential distribution could be generated, where the real time would be equal to $1/\lambda$. λ is a parameter of the generator.

User-Driven Selection – Sometimes, the user is suspicious about a piece of code that seems to be problematic and is very likely the source of program errors, such as deadlocks. This strategy allows the user to schedule threads

manually so the problematic places in the program can be reached in a much shorter time than using a different strategy. The controller must then read user input in every global consistent state, e.g. using an AWT window.

Defined Path Algorithm – Sometimes, it is necessary to perform thread switching in an exactly defined order. For example, if a deadlock is found and the computation path is stored somewhere, the user might want to reproduce the deadlock and therefore wants to perform an exactly the same order of switchings as in the last program run. The result will be identical if the same data are given to the program, including random numbers generated during the program's run. Otherwise, a different transition of the same thread can be executed from a given state. It is the thread's decision which of them will be actually executed, not the scheduler's decision. Actually, the scheduler just selects a thread and the selected thread can still influence the computation path if more than one transition can lead from its local consistent state. The system can therefore get to several global consistent states even if the same thread is selected.

It would also be convenient to explore all possible schedules of the program at once, as ExitBlock does it. However, this task seems to be a very difficult one because of the following reasons:

1. ExitBlock assumes that all threads are terminative so the depth-first search algorithm can be used. We do not have this assumption.
2. Every computation path of the program has a different length and different sets of threads can be enabled in the states of different computation paths. Even sometimes, the computation path may be infinitely long. It would be then very difficult to describe such paths that the controller would have to use. On the other hand, the program would be run N times in "normal mode" so all computation paths would be explored in iteration, not during one run.
3. If the last issue – the description of computation path – has to be overcome and the program has to execute all different schedules at once, it is necessary to have means to back up the current state and to return later to this state as ExitBlock does it, using the Rivet virtual machine. All this seems to be impossible without the necessary support from the JVM, unless the controller itself is able to clone the states of the program and to roll back to a backuped state.

4.3 Replacing the Entry Points' Behavior

The desired behavior of replacement routines has alredy been described in section 3.6, it is not therefore repeated here. But several things must be said in order to clearly define the controller's functionality:

1. All the replacement routines are methods of the `Controller` class and must have the `public` access right in order for the program to be able to call them.

2. All of them influence somehow the relationships of the modeling entities (that correspond to real Java objects, e.g. locks, threads, thread sets, ...) inside the controller but only some of them cause the controller to perform thread switching. For example, the replacement of `wait()` causes the controller to switch while the replacement of `notify()` does not.

3. The replacement of some functions, e.g. `join()`, cause the controller to switch even if the points where the original functions are called were not defined as consistent states of a thread in section 3.2. In the case of `join()`, the switching will take place only sometimes, if the called thread has not finished yet. It must be therefore seen as a conditional consistent state. See section 3.3.3 – Conditional Neighbourhood.

4.4 The Singleton Design Pattern

As a consequence of the section 4.1, it must always be guaranteed that all threads present in the program use services of *the same controller*. Otherwise, the thread switching would not work as intended because they would be switched in several different switchers so very likely more than one thread thread would run at a single point of time, which is of course inacceptable.

Therefore, it must be assured that the class from the controller package providing the interface to the serialized program has always *just one instance*. There exists a design pattern, called *Singleton*, invented right for this purpose.

The class that needs the property of a unique instance (let it be the class `Controller` in this case) has no public constructor which does not allow other class to create an instance of the class. A public method – `getInstance()`, for example – returns a static reference to an object typed to `Controller`. The instance is created either in the static initialization block of the class or in the `getInstance()` method, when it is called for the first time. In the latter case, the code of the method might look like the following:

```
public static synchronized Controller getInstance() {
    if (instance == null)
        instance = new Controller();
    return instance;
}
```

In the former case, the method always returns `instance`, which is created when the `Controller` class is loaded to memory by the class loader.

Chapter 5

The Converter

As described in chapter 4, the analyzed program uses services of a layer inserted between the program and the JVM – the controller. However, the insertion of the controller is not done automatically. The way how it is inserted between the two original layers is the subject of this chapter.

The approach presented here is based on the following assumptions:

- The source code of the tested program is available and it has been already successfully compiled.
- All additional libraries the program uses are also available in the form of source code. If they are not, the program cannot be properly analyzed if these libraries use thread-specific commands, such as synchronization, guarded actions, new thread creation, etc.

Normally, the tested program uses only services of the JVM, defined in standard libraries, such as the package `java.lang`. The *redirection of the thread-specific commands* must therefore be used in order for the tested program to use the controller instead of the JVM. This is not done at run-time but already before the tested program is compiled – the basic idea is based on *modification of source code* of the tested program. The modification is also called the *conversion* and the tool modifying the source code is therefore called the *converter*.

5.1 The Conversion Process

When the converter is started, it first tries to determine what are the input and output directories. The input directory contains the original program, the output directory will contain the converted program, ready to be tested with the controller package. If these parameters are not supplied by the user, implicit values are used. After that, the real conversion process can start.

Since it would be quite complicated to analyze Java source texts, the converter uses another approach. Instead of using `.java` files only, it uses their compiled equivalents – `.class` files – to gather all necessary information about the tested

program. A class must be compiled and its bytecode must be put to the input directory together with its source file in order for a class to be properly converted. If its bytecode is not present in the input directory, the source file itself is not touched by the converter and the class is not converted.

During the conversion process, the mechanism called *reflection* plays an important role. It is an ability of a Java program to get information about itself and its parts (classes, their fields, methods, constructors, objects, return types, parameters of methods, etc.) at run-time. Tools being part of the reflection subsystem are located in the `java.lang.reflect` package. The binary image of a class is loaded into memory and converted to an instance of the `Class` class[1]. This conversion is done by a *class loader* – an instance of the `FileClassLoader` class. It is necessary to use this non-standard class loader, subclassed from the `ClassLoader` class, since the standard one, present implicitly in every Java program, has certain limitations:

- It does not load classes that are not in the current directory or in the directory pointed to by the `CLASSPATH` environment variable.
- If a class belongs to a package, the standard class loader loads it only if its bytecode is stored in a file located in a subdirectory of the current directory or the `CLASSPATH` directory and the subdirectory's name corresponds to the name of the package.

After all classes are successfully loaded from the input directory, they are sorted into several categories. As described in chapters 5.3, 5.4, 5.5, and 5.6, there are different rules how to handle the conversion of a thread, of a "standard" class, etc. The categories are as follows:

Public Threads – Classes derived (directly or indirectly) from the `Thread` class. The class is public, i.e. it has the `public` modifier in its declaration and it is stored in a source file having the name of the class (+ `.java`).

Non-public Threads – Classes derived (directly or indirectly) from the `Thread` class. The class is not public, i.e. it has no access modifier in its declaration and it is stored in a source file together with another public class (either a thread class or a non-thread class).

Nested Threads – Classes derived (directly or indirectly) from the `Thread` class. The class is not public, i.e. it has no access modifier in its declaration and it is stored inside another class declaration.

Public Non-threads – "Standard", not runnable, classes derived (directly or indirectly) from `Object`, but not derived from `Thread`. The class is public, i.e. it has the `public` modifier in its declaration and it is stored in a source file having the name of the class (+ `.java`).

Non-public Non-threads – "Standard", not runnable, classes derived (directly or indirectly) from `Object`, but not derived from `Thread`. The class is not public, i.e. it has no access modifier in its declaration and it is stored in a source file together with another public class.

[1] A class whose name is `Class`.

Nested Non-threads – “Standard”, not runnable, classes derived (directly or indirectly) from `Object`, but not derived from `Thread`. The class is not public, i.e. it has no access modifier in its declaration and it is stored inside another class declaration.

Main Class – A public class having a `public static` method called `main` with an argument of type array of `String`s. There can be more such classes in the converted program. This category overlaps with other categories, for example a public thread class can also contain the `main()` method.

Note: In the rest of the chapter, non-threads are often referred to as “normal classes” or just “classes” while threads and their subclasses are reffered to as “threads”.

Interfaces are completely excluded from conversion since they have no real code and thus it is not necessary to convert them. Abstract classes are not excluded because they can contain non-abstract methods that are not overwritten in their subclasses.

After having sorted all loaded classes, the converter takes one after another from each category and converts them according to the algorithm specific to the category. The conversion process stops either if all classes are successully converted or if an error is encoutered during a class conversion.

5.1.1 Conversion of a Single Class

As described in section 5.1, the conversion of a single Java class is one of many steps performed during the whole conversion process. After all classes are sorted into categories, they are taken one by one and converted separately. The conversion of one class is what this section puts focus on.

The mechanism used for conversion of a class is depicted in figure 5.1. Data inputs, outputs, and tools used are represented by rectangles, actions by bubbles.

First of all, a `.class` file is taken from the input directory and loaded into memory. At this moment, the source file (`.java`) is not touched at all. Loading the `.class` file first assures that only classes being properly written and compiled without problems will be converted. When binary data are loaded from the file, the class loader (instance of `FileClassLoader`) converts them to an instance of `Class`. The `Class` class is a part of the Java reflection package, as mentioned on page 53. Using various methods of the `Class` class, the following information is retrieved:

- Information about *class modifiers*. This includes access modifiers, but also the `abstract` and `interface` modifiers. The method `getModifiers()` is used for this purpose.
- The name of the class’ *source file*. Since there is not a method to return the source file’s name, the following approach is used: If the class has the `public` modifier, its source file’s name is identical to the class’ name, plus ‘`.java`’. If the `public` modifier is not set, `javap`[2] is executed as a separate

[2]Java decompiler, distributed as a standard part of the JDK.

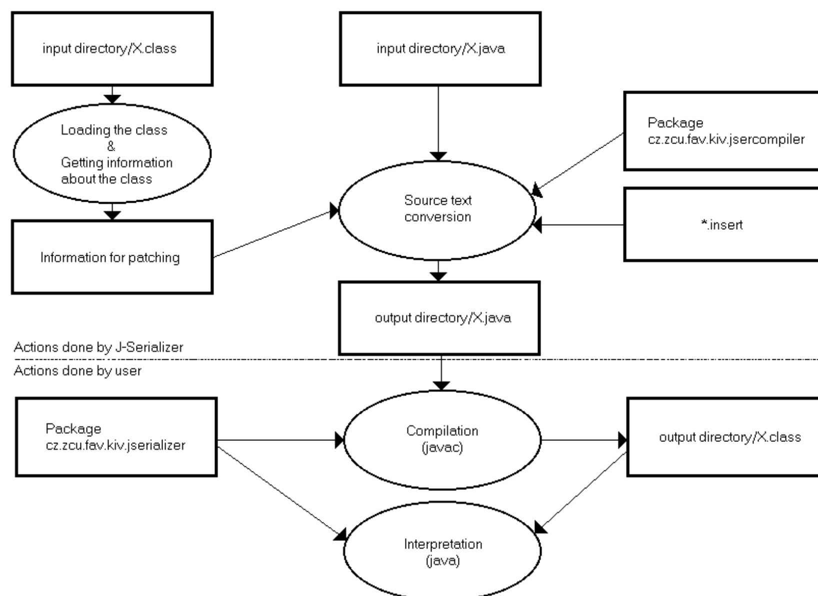

Figure 5.1: Conversion of a Single Java Class

process and its output is redirected to an `InputStreamReader`. The first line of `javap`'s output contains the name of the source file.

- Information about all class' *constructors* and *methods*. Two arrays (one array of type `Constructor` and one array of type `Method`) are returned. `getDeclaredMethods()` is used to get the array of methods and `getDeclaredConstructors()` is used to get the array of constructors.

When all necessary information is available, the real conversion process can start. From the input directory, the class' source file is taken, modified in several steps and written to the output directory. When the output file is written, the class is supposed to be properly converted and the converter can pick up another class for conversion.

Several stages can be distinguished during the conversion of a source file. They are as follows:

1. Comments are removed from the source file. This allows the converter to work more quickly and precisely.
2. Braces (characters '`{`' and '`}`'), which denote beginning and end of blocks of code in Java programs, are put on separate lines. At most one brace is placed on a line and nothing than the brace itself. This is due to the source code parser which must be able to find beginnings and ends of blocks easily.
3. Spaces are inserted between consequent tokens of the code. Spaces are inserted between any two consequent identifiers, operators or separators,

only square brackets '[' and ']' are left stuck together. Again, this is due to the source code parser which must be able to find sequences of strings in the source code.

4. Some important keywords[3], such as `wait`, `notify`, `notifyAll`, `synchronized`, `join`, ...[4] are replaced with their converter-specific identifier if they are not inside double quotes. This is due to the converter which tries to replace these keywords with a piece of code linking the program to the controller – an `.insert` file (see section 5.2). However, these words may be put in a string (inside double quotes) which means that they must not be replaced.

5. Either a method or a constructor is *patched*, i.e. its beginning if found, the method's or constructor's body is parsed and some important changes are made – a keyword is replaced, a piece of code is added from an `.insert` file etc. This point repeats as needed, i.e. until all methods and constructors of the class are patched. Sometimes, some other changes are made, e.g. an `.insert` file is inserted at the beginning of the file or at the beginning of the class' declaration. The way how the converter behaves at this point is the principal difference between the converter of normal classes and the converter of threads. This point will be described in more detail in subsequent sections.

5.2 Parametrization of the Conversion

The source code that is inserted to the original program's source code resides in several files with common suffix `.insert`. They must must be located in the `insert` directory of the directory from which the converter is run. A separate file exists for each entry point type (see section 3.6) to be replaced, for each category of classes and for each type of scheduler. The type of scheduler is only distinguished in `.insert` files patching the `main()` method because the type is specified only once to the controller, when it initializes the scheduler.

This concept brings several advantages when compared to the case when patching information is stored directly in the source code of the converter:

- The patching information for a specific purpose is located at one place only. Therefore, if there is a need to modify it, it is possible to do it without affecting the converter's source code where the same piece of inserted code might be located more than once.

- If a part of the controller package changes, the changes have usually no influence on the converter itself, only on the `.insert` files.

The type of the required scheduler is given as an argument to the converter, the rest of information necessary for selecting the right `.insert` file is determined from the type of the currently converted class and from the currently patched method or from the keyword or method call being replaced.

[3]They are not real keywords of the Java language, they are only called so in this text.
[4]See section 3.6 for complete listing.

Not all types of `.insert` files are available for all class categories. For example, it makes no great sense to provide an `.insert` file to patch the beginnings and ends of the `run()` method of non-thread classes.

An example of the `.insert` file to be inserted to the beginning of the `run()` method of a thread:

```
Controller.getInstance().registerNewThread(this);
```

Sometimes, it is necessary to substitute a part of an `.insert` file with a value computed or otherwise found during the conversion. For example, the replacement of the `wait()` method needs to know what lock is used for this operation. The lock can be either directly denoted in the source code (`myLock.wait()`) or it has to be found by the converter. So, an `.insert` file can be *parametrized* with a certain parameter. The parameter must be clearly distinguishable from the rest of the `.insert` file and a routine of the converter must exist to expand it to a sequence of Java code.

An example of the `.insert` file to replace any call to the `notifyAll()` method:

```
Controller.getInstance().callNotifyAll($GET_LOCK$);
```

The `$GET_LOCK$` formal parameter must be expanded by the converter to return a reference to the actually used lock, i.e. `this` or any other explicit lock.

5.3 Conversion of Threads

There are many places in a thread's source code where important changes are made. At the very beginning of the source file, all classes from the controller package are imported – the file `all_import.insert` is inserted.

The *constructor* is patched in a special way, differently from methods. A code registering the thread's creation is added to the end of the constructor – file `Thread_constructor_end.insert`. If there are more than one versions of constructor (it is overloaded), all of them must patched. But the controller must be aware of the situation when a constructor calls another version of the constructor or it calls the constructor of its superclass.

Also the `run()` method is patched in a special way because this method is called automatically by the JVM when a thread is started. At the beginning, the patched version contains code performing registration and synchronization of all started threads which is inserted from `Thread_run_beginning.insert`. At the end, a deregistration routine is inserted from `Thread_run_end.insert`. The rest of the method is patched in exactly the same way that other methods are.

All other code, including the code inside `run()`, is patched according to section 5.1.1. All important calls to the JVM functions are replaced by the content of an `.insert` file.

At the beginning and at the end of every method, a file is inserted whose code informs the controller that a method has just been invoked or that a thread is about to leave a method. This can be extremely useful when logging is required.

5.4 Conversion of Normal Classes

In short, conversion of normal classes does not differ much from conversion of threads. It can be seen as a generalization of thread conversion. It is not necessary to patch the constructor and the `run()` method. If such method exists, it has nothing to do with `Thread.run()` and can be treated as any other method.

5.5 Conversion of the Main Class

Any class with the method `public static void main(String[] args)` can be started by the JVM when the program is launched. Therefore, the converter must apply similar actions in the `main()` method as it applies in the `run()` method of a thread class. The implicit thread must also be registered at the beginning and deregistered at the end.

Patching the constructor (if it exists) has no effect here since the `main()` method is static so no constructor is actually called. The controller does not need to know about the thread before it is started because no controller exists at that time.

The controller must properly detect the situation when `main()` is called from within an already running program, i.e. it is not the first method run in the program. In such case, the current thread must not be registered again.

5.6 Conversion of Private and Nested Classes

The same principles are used for non-public and nested classes as for "normal" public classes. However, there are some technical obstacles that could prevent the converter from functioning properly. Therefore, the source code of a non-public or nested class is always copied to an independent temporary file, kept apart of the original source file and copied back after conversion.

The converter must be aware of non-public and nested classes when converting a public class, too. The corresponding code must be skipped. Or, better, all classes (including public classes) should be copied to an independent file, converted independently, and all composed together (in a convenient order) afterwards.

5.7 Using Java Source Code Parsers

The presented method of conversion has several weak points:

- It needs the bytecode. This is not a big problem since it can be generated at any time using `javac`.
- It is prone to errors. Ususally, there are more ways how a certain language element can be written and the converter must be able to deal with all possibilities. A typical example is the order of keywords in a method header.
- It is possible that it does not cover the whole Java language. Some language elements may not be understood or may be ignored.
- The Java language may change in future in such a way that the converter will not cover all language elements (some elements will be added) or it will fail during the converion process (some elements will be removed or their syntax will be changed).

It would therefore be convenient to replace the above conversion with a more sophisticated algorithm, based exclusively on a parser of the Java language. Such algorithm would correctly go through the whole source file and detect all important points where an appropriate action would be taken. The bytecode could be still used, but only as a guarantee that the source code is well written. Moreover, one pass through the `.java` file would be sufficient to convert the class, it would not be necessary to repeat the search for every method or constructor.

There exist parser generators like JavaCC (Java Compiler Compiler) that take a grammar description as input and generate the source code of a program able to parse input formatted according to the specified grammar. If the whole grammar of Java, available at [JLS], would be given to JavaCC, the generated program would parse Java source files correctly.

A parser of the Java language has also been written already. It is available at [Parser]. It is not perfect in the sense that it covers just Java 1, so, for example, inner classes are not supported. The ANTLR parser generator (part of PCCTS, the Purdue Compiler-Construction Tool Set) was used to generate this parser. A little inconvenience is that the parser is generated in C++, not in Java, so it must be executed via `Runtime.exec()` and it must be compiled for any platform it is intended to be used on. Version 2.0 of ANTLR should be able to generate parsers in Java which would make things much easier. In order to support Java 2, the file `java.g` (the Java grammar read by ANTLR) would have to be extended.

Using a parser is certainly a promising method of Java source code conversion. It will be seriously considered during further work to replace the currently used method.

Chapter 6

Conclusion

This chapter summarizes the current state of work that has lead to this paper, enumerates possible usage of the tool built up upon the presented principles, and finally states the goals of further work that should end up with a Ph.D. thesis.

6.1 Current State of Work

So far, initial versions of the controller and the converter have been developed. They are both parts of a tool called J-Serializer.

The controller is not able yet to handle all consistent states or other entries to the JVM, only beginnings and ends of synchronized blocks and thread creation, startup and death. The rest is still to be implemented. Also, the controller supports just one scheduling strategy – the time measurement strategy – that suffers from imperfect implementation of time measurement in Java or on certain hardware platforms in general. Not all entities have been incorporated to the controller yet, the controller just handles threads. The support for locks, wait sets, etc. is a subject of further work. However, it has been proved that the principles of deterministic thread switching are realizable and that this method can be a benefit in the area of concurrent Java programs tracing and debugging.

The converter works on principles described in section 5.1.1, which is not a guarantee of successful conversion in all cases. Therefore, the use of a Java language parser is seriously considered and its incorporation to J-Serializer or modification will also be a subject of future work.

6.1.1 Related Publications

Just after the first version of J-Serializer was made, it was described in a technical report [J-Ser]. A short overview can be found in [Ser], a more detailed overview with an explanation of the state-space model can be found in [JCPD]. The work summarized to overhead projector slides can be found at [DSS].

The system of deterministic Java thread switching was first introduced in J-Sim, a Java library for discrete-time simulation. The switching system is described in [J-Sim1], the usage of J-Sim can be found in [J-Sim2] or in its documentation.

6.2 Application Area

The purpose of this work is not to invent an already invented algorithm or to duplicate work that has already been done by others, as stated in chapter 2. In fact, just two tools from that chapter target the area of Java concurrent programs: the implementation of the ExitBlock algorithm and the Java PathFinder 2 tool. They both require a special Java virtual machine and work directly with bytecode. This ensures that the program is tested in its native, unmodified form. However, the feedback to the user is whether the program is correct or not and the schedule leading to a potentional deadlock. Also, there are some assumptions about the tested program.

The main advantage of J-Serializer (when fully implemented) is the feedback to the user. He/she should know what happens *at the level of his/her Java source code*, not at the level of Java bytecode instructions. Only then the reasons of malfunction can be easily understood and corrected. Also, there are no assumptions about the finitness of threads of the program. J-Serializer is able to execute the program in a neverending loop, if the user wishes to do so. One of the execution modes – the user-driven one – should allow him/her to *concentrate on suspicious or problematic parts* of the code, without the need of executing the rest of the program, if it is possible. In general, it could be said that J-Serializer is more "user-friendly" than the other tools are and it has a broader area of application due to nonexisting assumptions about the tested program. An example of application is testing so-called *reactive systems* – sets of objects that respond to incoming messages and change their state accordingly – where the communication between the user and the program plays an important role. On the other hand, Java concurrent programs are very rarely used for massive mathematical computation. For such type of tasks, better tools written in languages offering higher performance already exist (e.g. PVM, MPI), allowing the task to be distributed on a network.

6.3 Goals of the Ph.D. Thesis

When taking into account the most probable application area (see the last section), the future work should focus mainly on the following subjects:

- There should be an ability of the tester to influnce the tested system (in its positive meaning) and observe the reaction. For example, pre-defined messages to the objects of the system could be sent in a defined order and at defined time or during defined transitions.
- For the purpose of testing reactive systems, a kind of model time[1] would

[1] As it can be found in J-Sim, for example.

be useful. This time could be implemented as a simple counter or the real time or any combination of those.

- The behavior of the tested program could be tested in terms of the reaction time, e.g. the real time or model time or the number of consequent consistent states could be measured until the analyzed part of software "responds" to the message. The term "response" means achieving a defined state, fulfilling a pre-defined condition, sending other messages back to the original sender or to any other objects, etc.

Concerning the technical part of the work, i.e. the implementation of the presented principles to the Java programming language, the tool has to be considerable improved. In future, most effort should be put mainly into the following tasks:

- The converter should be rewritten to use a reliable and up-to-date parser of the Java language that would assure nonproblematic conversion of source texts.
- The controller should be fully implemented to support existing operation on threads.
- If there exists an exact and stricly defined specification of Java thread scheduler, it should also be incorporated to the controller as one of its scheduling strategies.
- There should be a way how to describe a complete execution path (for terminating programs only) and how to impose the controller to follow that path. Moreover, there should be a way to enumerate all possible execution paths and to "replay" them. The program would be then executed N times, each time from the beginning to the end. It is very likely that the set of all possible schedules will have to be created step-by-step, during paths executed so far. However, even this method will not probably be able to handle dependencies on random numbers and real time.

After successfully finishing all above tasks, J-Serializer should be able detect all problems during all possible schedules of the program (as ExitBlock does it), while keeping the principles it is based on, mainly the friendly and intuitive feedback on the source-code level. Moreover, it should be superior in some specific application areas, like the above stated reactive systems.

Bibliography

[JLS] James Gosling, Bill Joy, Guy Steele, Gilad Bracha: **The Java Language Specification**, Second Edition. Sun Microsystems, 2002. `http://java.sun.com/docs/books/jls/second_edition/html/-j.title.doc.html`

[JVMS] Tim Lindholm, Frank Yellin: **The Java Virtual Machine Specification**, Second Edition. Sun Microsystems, 1999. `http://java.sun.com/docs/books/vmspec/2nd-edition/html/-VMSpecTOC.doc.html`

[ExBlk1] Derek Bruening, John Chapin: **Systematic Testing of Multithreaded Programs**. MIT/LCS Technical Memo, LCS-TM-607, Massachusettes Institute of Technology, Cambridge, Massachusettes, USA, 2000. `http://web.mit.edu/iye/www/docs/tester-TM607.ps.gz`

[ExBlk2] Derek Bruening: **Systematic Testing of Multithreaded Java Programs**. Master's Thesis, Massachusettes Institute of Technology, Cambridge, Massachusettes, USA, 1999. `http://web.mit.edu/iye/www/docs/thesis.ps.gz`

[PF1] Klaus Havelund, Thomas Pressburger: **Model Checking Java Programs Using Java PathFinder**. International Journal on Software Tools for Technology Transfer, Vol. 2, No. 4, 2000. `http://ase.arc.nasa.gov/people/havelund/Publications/-jpf-sttt.ps`

[PF2] Guillaume Brat, Klaus Havelund, SeungJoon Park, Willem Visser: **Java PathFinder – Second Generation of a Java Model Checker**. Proceedings of the Workshop on Advances in Verification, Chicago, Illinois, USA, 2000. `http://ase.arc.nasa.gov/visser/wave00.ps.gz`

[Sem] Abhik Roychoudhury, Tulika Mitra: **Specifying Multithreaded Java Semantics for Program Verification**. Proceedings of the 24th International Conference on Software Engineering, Orlando, Florida, USA, 2002. `http://portal.acm.org`

[ACL2] J Strother Moore, Robert Krug, Hanbing Liu, Gerge Porter: **Formal Models of Java at the JVM Level – A Survey from the ACL2**

Perspective. Presented at the Workshop on Formal Techniques for Java Programs, University Eötvös Loránd, Budapest, Hungary, 2001.
http://www.cs.utexas.edu/users/moore/publications/-jvm-models/full.ps.gz

[VS] Patrice Godefroid: **Model Checking for Programming Languages using VeriSoft**. Proceedings of the 24th ACM Symposium on Principles of Programming Languages, Paris, France, 1997.
http://www.bell-labs.com/project/verisoft/popl97.ps

[SPIN] Gerard J. Holzmann: **The Model Checker SPIN**. IEEE Transactions on Software Engineering, Vol. 23, No. 5, 1997.
http://spinroot.com/spin/Doc/ieee97.pdf

[Era] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, Thomas Anderson: **Eraser: A Dynamic Data Race Detector for Multithreaded Programs**. ACM Transactions on Computer Systems, Vol. 15, Issue 4, 1997.
http://portal.acm.org

[J-Sim1] Jaroslav Kačer: **J-Sim – A Java-based Tool for Discrete Simulations**. Proceedings of the 23rd International Autumn Colloquium on Advanced Simulation of Systems. Ostrava, Czech Republic, 2001.

[Ser] Jaroslav Kačer: **Java Programs Serialization**. Proceedings of the 5th International Conference on Information Systems Modelling. Ostrava, Czech Republic, 2002.

[J-Sim2] Jaroslav Kačer: **Discrete Event Simulations with J-Sim**. Proceedings of the Inaugural Conference on the Principles and Practice of Programming in Java. Maynooth, Co. Kildare, Ireland, 2002.

[JCPD] Jaroslav Kačer, Stanislav Racek: **A Method of Java Concurrent Programs Debugging**. Proceedings of the 5th International Scientific Conference on Electronic Computers and Informatics. Košice, Slovakia, 2002.

[J-Ser] Jaroslav Kačer: **J-Serializer – Java Programs Serializer**. Technical Report DCSE/TR-2001-07. University of West Bohemia, FAV, KIV, Pilsen, Czech Republic, 2001.
http://www.kiv.zcu.cz/publications/2001/tr-2001-07.pdf

[Petri] Armin Zimmermann: **Petri Nets**. Technische Universität Berlin, Germany.
http://pdv.cs.tu-berlin.de/~azi/petri.html

[Lea] Doug Lea: **Concurrent Programming in Java – Design Principles and Patterns**, Second Edition. Addison-Wesley Longman, USA, 2000. ISBN 0-201-31009-0.

[Conc] Jeff Magee, Jeff Kramer: **Concurrency – State Models and Java Programs**. John Wiley and Sons, UK, 1999. ISBN 0-471-98710-7.

[Parser] Jim Coker, Terrence Parr: **Parsers, Part III: A Parser for the Java Language**. jGuru.com, 1997.
`http://developer.java.sun.com/developer/-technicalArticles/Parser/SeriesPt3/`

[DSS] Jaroslav Kačer: **Observation and Control of Multithreaded Java Program Execution**. Slides from a DSS Seminar. University of West Bohemia, FAV, KIV, Pilsen, Czech Republic, 2003.
`http://www.kiv.zcu.cz/vyzkum/groups/dss/download/-presentation-2003-04-14.zip`

[LTSA] Jeff Magee: **LTSA – Labelled Transition System Analyser**. University of London, Imperial College of Science Technology and Medicine, Department of Computing, London, UK, 1999.
`http://www-dse.doc.ic.ac.uk/concurrency/ltsa/LTSA.html`