# Trace-based Debugging and Visualisation of Concurrent Java Programs with UML

Katharina Mehner

Paderborn, 10 February 2005

Doctoral thesis submitted to the Faculty of Electrical Engineering,

Computer Science and Mathematics in partial fulfilment of the

requirements for the degree of *Dr. rer. nat.*

Supervisors:

Prof. Dr. Gregor Engels, Universität Paderborn

Prof. Dr. Stefan Jähnichen, Technische Universität Berlin

# Abstract

This thesis describes an approach for automated detection of concurrent liveness failures in the execution of Java programs.

Concurrent programs are highly prone to failure because of the inherent nondeterminism. Developers of concurrent Java programs are not well supported in detecting concurrency failures, i.e. failures that are due to interactions between multiple threads. These failures are neither well documented nor do tools like debuggers allow developers to identify them at runtime.

This thesis analyses and classifies liveness failures, a special kind of concurrency failures, and the associated potentials in Java. A UML statechart is developed that models the interaction of Java threads. Liveness failures and potentials are specified formally in terms of the states controlling the interaction of threads and in terms of the events exchanged by interacting threads.

Detection algorithms are developed to identify the specified failures in a program execution. A UML profile extending UML interaction diagrams is developed to visualise the execution of concurrent Java programs and detected liveness failures and potentials.

In order to deploy the algorithms and the UML profile, tool support concepts are provided. This involves the specification of a trace format and a tracing method to collect execution data from a running Java program, and the specification of methods to analyse the trace and to visualise the trace and the analysis results.

The concepts are implemented in the JAVIS prototype, which consists of a Java tracer with an analysis facility for monitoring liveness in concurrent Java programs, and a plug-in extension to the UML CASE tool Together for importing and displaying concurrent Java traces including failures and potentials.

# Acknowledgments

I would like to thank Gregor Engels for giving me the opportunity to write this thesis and for his enduring support. His advice and critical comments were a great help in shaping this thesis.

My thanks also go to Stephan Herrmann and Stefan Jähnichen for supporting me in the final stage of the thesis. Stefan Jähnichen kindly agreed to be my second supervisor.

I wish to thank all my former colleagues at the University of Paderborn for providing a good working atmosphere and engaging in fruitful discussions and a number of successful collaborations.

I am grateful to all my colleagues at the Technical University of Berlin for their support while I was completing the thesis, especially for their critical proof reading and helpful suggestions.

My work on the thesis also benefited from the warm welcome and the cordial atmosphere I experienced during my stay at Lancaster University.

Finally, I would like to thank my family and friends for their love, encouragement and understanding.

*To my parents*

# Contents

**List of Figures** **xv**

**List of Tables** **xvii**

**1 Introduction** **1**

**2 Motivation** **5**
2.1 Background . . . . . 5
2.2 Problem Description . . . . . 12
2.3 Goals . . . . . 21
2.4 Synopsis . . . . . 23

**3 Requirements for Automated Failure Detection** **25**
3.1 Failure Description and Classification . . . . . 26
3.1.1 Purpose . . . . . 26
3.1.2 Scope . . . . . 26
3.2 Automated Detection . . . . . 27
3.2.1 Contextual Requirements . . . . . 28
3.2.2 Data Collection Requirements . . . . . 29
3.2.3 Data Analysis Requirements . . . . . 31
3.2.4 Failure Visualisation Requirements . . . . . 32
3.3 Summary . . . . . 33

**4 Concurrent Programming in Java** **35**
4.1 Thread Principles . . . . . 36
4.2 Thread Lifecycle . . . . . 42
4.3 Unsynchronised Interaction . . . . . 49
4.4 Synchronised Interaction . . . . . 51
4.5 Summary . . . . . 60

**5 Liveness Failures and Potentials in Java 61**
5.1 Terminology . . . 62
5.1.1 Error, Mistake, Fault, and Failure . . . 62
5.1.2 Potential for Failure . . . 64
5.1.3 Failure and Symptom . . . 67
5.2 Liveness Failures . . . 68
5.2.1 Liveness and Safety . . . 68
5.2.2 Concurrent Java Liveness Failures . . . 70
5.2.3 Potentials for Java Liveness Failures . . . 82
5.2.4 Classification of Java Liveness Failures and Potentials . 84
5.2.5 Failures in Concurrent Application Logic . . . 88
5.3 Summary . . . 88

**6 A Model of Thread Synchronisation 89**
6.1 Model Requirements . . . 90
6.1.1 Concurrent Liveness Failure Characteristics . . . 90
6.1.2 Dynamics of Thread Synchronisation . . . 91
6.1.3 Control Flow States . . . 93
6.1.4 Summary of Model Requirements . . . 95
6.2 Related Work . . . 95
6.2.1 The Java Language Specification . . . 96
6.2.2 Formal Java Semantics . . . 97
6.3 A Statechart Model for Thread Synchronisation . . . 98
6.3.1 The UML Statechart Approach . . . 99
6.3.2 Classification of Thread Lifecycle Methods . . . 102
6.3.3 Principles for Designing States and Transitions . . . 104
6.3.4 The Resulting Statechart . . . 111
6.3.5 System State and History . . . 119
6.4 Formalising Liveness Failures and Potentials . . . 122
6.4.1 Formal Description of Failures . . . 122
6.4.2 Formal Description of Potentials . . . 124
6.5 Summary . . . 126

**7 Trace-based Data Collection 129**
7.1 Tracing Requirements . . . 130
7.1.1 Format, Schema, and Encoding . . . 131
7.1.2 Trace Generation . . . 133
7.2 Related Work . . . 135
7.2.1 Basic Techniques . . . 135
7.2.2 Trace Formats . . . 139
7.2.3 Code Instrumentation . . . 139

7.2.4 Runtime APIs . . . 140
7.2.5 Debuggers . . . 143
7.2.6 Trace-based Tools . . . 145
7.2.7 Comparison . . . 146
7.3 JAVIS-Tracer for Concurrent Java Programs . . . 148
7.3.1 Trace Format . . . 148
7.3.2 Trace File Generation . . . 150
7.3.3 Tracer Architecture . . . 153
7.4 Summary . . . 155

**8 Trace-based Failure and Potential Analysis 157**
8.1 Analysis Requirements . . . 157
8.1.1 Functional Requirements . . . 157
8.1.2 Time and Space Complexity . . . 159
8.2 Related Work . . . 159
8.2.1 Deadlock Detection . . . 159
8.2.2 Deadlock Potential Detection . . . 160
8.2.3 Failures and Potentials Involving wait() or join() . . . 162
8.3 JAVIS-Algorithms . . . 162
8.3.1 Cycle Detection . . . 162
8.3.2 Missed Notification . . . 165
8.3.3 Potentials for Cyclic Dependencies . . . 166
8.4 Summary . . . 166

**9 Trace-based Failure and Potential Visualisation 167**
9.1 Visualisation Requirements . . . 167
9.1.1 Trace Visualisation . . . 168
9.1.2 Failure and Potential Visualisation . . . 170
9.1.3 Visualisation Environment . . . 171
9.2 Related Work . . . 171
9.2.1 Visualisation of Concurrent Programs . . . 172
9.2.2 Object-Oriented Visualisation . . . 173
9.2.3 UML-based Visualisation . . . 177
9.2.4 Comparison . . . 180
9.3 JAVIS-Visualisation . . . 181
9.3.1 UML Profile for Java Traces, Failures, and Potentials . 181
9.3.2 Visualisation Architecture . . . 189
9.4 Summary . . . 189

**10 Using the JAVIS Prototypes** **191**
10.1 Example for Automated Failure Detection . . . . . 191
10.1.1 The Banking Example Revisited . . . . . 193
10.1.2 Tracing . . . . . 194
10.1.3 Automated Deadlock Detection . . . . . 195
10.1.4 Trace and Deadlock Visualisation . . . . . 195
10.2 Example for General Purpose Tracing . . . . . 199
10.2.1 A Simulation Software Example . . . . . 199
10.2.2 Tracing and Visualising JEVOX . . . . . 201
10.3 Summary . . . . . 201

**11 Conclusion** **203**
11.1 Contributions . . . . . 203
11.2 Evaluation . . . . . 204

**12 Outlook** **209**
12.1 Remaining Issues . . . . . 209
12.2 Evolution . . . . . 210
12.3 Related Domains . . . . . 211

**Bibliography** **213**

**Index** **223**

# List of Figures


2.1 Banking Example . . . . . 8
2.2 Banking Example with Deadlock . . . . . 9
2.3 Cyclic Resource Dependency and Wait-For Graph . . . . . 11
2.4 Source Code View . . . . . 14
2.5 Call Stack View . . . . . 15
2.6 Monitor View . . . . . 16
2.7 Monitor View with Callstacks . . . . . 17
2.8 Trace Example . . . . . 19
2.9 Thesis Goals . . . . . 22

3.1 Requirements for Automated Failure Detection . . . . . 27
3.2 First Refinement of Goals . . . . . 33

4.1 Banking Simulation Classes . . . . . 37
4.2 Unsynchronised Access of Accounts . . . . . 40
4.3 Lost Update . . . . . 41
4.4 Deriving New Threads . . . . . 43

5.1 Error Terminology . . . . . 63
5.2 Extending the Basic Terminology with Potential . . . . . 64
5.3 Potential for Deadlock . . . . . 65
5.4 Deriving Potential Conditions from Failure Conditions . . . . 66
5.5 Safety and Liveness Terminology . . . . . 69
5.6 Deadlock . . . . . 71
5.7 Missed Notification . . . . . 73
5.8 Balancing wait() and notify() . . . . . 74
5.9 Nested Monitor Lockout . . . . . 76
5.10 Circular Join . . . . . 77
5.11 Self Join . . . . . 77
5.12 Join-induced Deadlock . . . . . 78
5.13 Livelock . . . . . 79

6.1 Failures and Potentials as Legal System Behaviour . . . . . 92
6.2 Thread Lifecycle Statechart . . . . . 113
6.3 Thread Lifecycle Statechart with Guards and Actions . . . . . 115
6.4 Object Synchronisation Behaviour Statechart . . . . . 116

7.1 Requirements for Data Collection . . . . . 130
7.2 Levels of Instrumentation . . . . . 137
7.3 The Java Platform Debugger Architecture (JPDA) . . . . . 142
7.4 Trace Format Example . . . . . 149
7.5 Class Diagram of the Tracer . . . . . 154

8.1 Requirements for Data Analysis . . . . . 158
8.2 Cycle in Graph Structure . . . . . 163

9.1 Requirements for Data Visualisation . . . . . 168
9.2 GThread History View . . . . . 172
9.3 GThread Mutex View . . . . . 173
9.4 Jinsight Execution View . . . . . 174
9.5 Jinsight Execution View - Zoom . . . . . 175
9.6 Jinsight Reference Pattern . . . . . 176
9.7 Tagged Values for Messages in Interaction Diagrams . . . . . 185
9.8 Graphical Notation for Stereotypes . . . . . 186
9.9 Cyclic Failures using Stereotypes . . . . . 188
9.10 Class Diagram of the Together Visualisation . . . . . 190

10.1 Banking Application Class Diagram . . . . . 192
10.2 Sequence Diagram Generated by Together . . . . . 192
10.3 Trace Generation Dialogue . . . . . 193
10.4 Part of the Trace . . . . . 194
10.5 Importing a Trace in Together . . . . . 195
10.6 Sequence Diagram of a Trace . . . . . 196
10.7 Collaboration Diagram with Involved Methods . . . . . 198
10.8 Collaboration Diagram with Deadlock . . . . . 199
10.9 Complete Trace Visualisation in Together . . . . . 200
10.10 Part of the Remote Trace . . . . . 202

# List of Tables


4.1 Signature of sleep() . . . . . 45
4.2 Signature of Interrupt Methods . . . . . 51
4.3 Signature of join() . . . . . 52
4.4 Keyword synchronized . . . . . 53
4.5 Signature of wait() and notify/All() . . . . . 56
4.6 Java Concurrency Concepts . . . . . 59

5.1 Liveness Failures . . . . . 86
5.2 Potential Liveness Failures . . . . . 87

6.1 Mapping Java Behaviour to Statechart Events . . . . . 109

7.1 Trace Collection Approaches . . . . . 147

9.1 UML Profile for Concurrent Java Traces: Tags . . . . . 184
9.2 UML Profile for Concurrent Java Traces: Stereotypes . . . . . 187

# Chapter 1

# Introduction

*Concurrency* is an important concept for developing modern software systems. While it is indispensable for most applications in distributed and embedded programming, in general it is a useful concept that allows systems to be structured efficiently, intuitively and flexibly.

In the past, it was mainly operating systems that provided support for *concurrent programming*. Since then, the level of abstraction has risen. Today, thread libraries [But97] for widespread programming languages such as C/C++ are used. The most recent approach is to tightly integrate concurrency at the language level such as in Java [GJSB00]. Note that concurrent programming is not restricted to a particular programming paradigm, although we focus on imperative object-oriented programming languages.

Despite advances in the field of concurrent programming, concurrency still causes programmers problems. From time to time, *concurrency problems* cause severe incidents. A famous example is the software failure of the Mars pathfinder, which experienced a priority-inversion problem [Ree98]. In such a situation, a high-priority computation is waiting for a resource from a low-priority computation. The low-priority computation cannot finish as long as medium-priority computations are executing [SRL90]. The pathfinder system was trying to handle the problem by resetting again and again, which resulted in loss of data [Jon97]. Fortunately, the problem was quickly analysed and could be corrected in sito by transmitting a software patch. It has also been reported that the problem was present in test runs but was not detected because the test-analysis tools failed to address it. While concurrency problems in embedded systems tend to have disastrous effects, non-safety-critical and non-real-time systems can also experience concurrency problems. In 1999, COMPAQ, a provider of a thread library, found that their users were often blaming the library for containing bugs not realising that they had constructed deadlock-prone programs [Har00].

Concurrency problems are difficult to prevent because concurrent programs are inherently *nondeterministic*. Their behaviour at runtime, including undesirable behaviour, is difficult to predict. In 1977, Lamport provided a general classification of concurrency problems into liveness and safety problems [Lam77, OL82], i.e. problems which result in lack of progress of a program and problems which violate the consistency of program data. Other authors have provided necessary and sufficient conditions for individual liveness problems such as the deadlock [Hol71]. Nevertheless, the above examples demonstrate that programmers did not always deal with these problems adequately. Mainly, two reasons for this are conceivable: missing training and missing tool support because even experts cannot cope manually with the complexity of concurrent programs. Programmers do not only need editors and compilers. The success of a complex paradigm such as concurrency depends also to a large extent on enhanced tool support.

An area where tool support has not been evolving at the same pace as concepts is *debugging* for concurrent object-oriented programming languages. It is not a new problem that especially debuggers do not grow to the needs for software development, especially for software that is becoming increasingly complex. From time to time, state-of-the-art surveys document almost complete lack of progress in that area. In 1997, ACM published a Communications issue titled "The Debugging Scandal and What to Do About It" [ACM97], which reported that still too much buggy software reaches its users and that debugging support has not changed much within 30 years of programming with inserting print statements into code still being the tool of choice. Better tools were assumed to save significant effort also for experienced programmers. At that time new approaches such as debuggers which keep a history of a program or provide improved visualisations were only present in academia. Still, to date, such tools exists mainly in research.In 2003, the ACM Queue [ACM03], actually dedicated to future trends in computing, still saw a need to address the basic topic of debugging of concurrent and distributed software in depth.

In the case of concurrency, today's debuggers lack concepts for dealing with multiple threads and automated detection of many well-known concurrency problems. This is particularly problematic, because testing and debugging are crucial and time consuming activities, and lack of support hampers the successful use of concurrent object-oriented languages such as Java. This thesis examines how the situation can be improved for Java.

The background of safety problems has been analysed in depth in research. Many algorithms and tools have been proposed, also for Java, such as in [CLL+02, vPG01]. To the contrary, Java liveness problems have not yet been dealt with in depth. Java liveness problems are neither completely doc-

umented nor extensively supported by debuggers of integrated development environments (IDE) or by dynamic analysis tools. Also, the visual capabilities of the tools for supporting the development of concurrent Java programs are not adequate when dealing with multiple threads.

The goal of this thesis is therefore to provide a detailed analysis of the background of Java liveness failures and to develop concepts for automatically detecting liveness problems in running programs and for improving the visualisation of tools for debugging Java programs.

# Chapter 2

# Motivation

In this chapter, we motivate the need for improving concepts for debugging concurrency problems. In section 2.1 we give an overview of the main concepts of concurrent programming and typical concurrency problems. In section 2.2 we demonstrate that there is a lack of concepts for debugging concurrency errors in Java. Neither are the errors well documented nor do tools such as debuggers account for their special characteristics. We conclude the motivation by stating the goals of this thesis in section 2.3. Section 2.4 gives an overview of this thesis.

## 2.1 Background

In this section, we will introduce the main concepts of concurrent programming. Then we will shed light on the problems in concurrent programs. Concurrent programs suffer from two kinds of errors, liveness and safety errors. As the focus of this thesis is on liveness errors, in this section, we will illustrate the characteristics of liveness errors with an example.

### Concurrent Programming

Concurrency in an imperative program is the seeming or actual simultaneity of control flows. The multiple flows of control in a concurrent program are also called *threads of control* or simply threads. In general, the number of threads in a program is not limited and not determined at compile time. Threads do not only work in isolation from each other but can interact in a number of ways either by synchronisation or by shared data.

The behaviour of a concurrent program is *nondeterministic* in the sense that its execution order is only partially defined. More specifically, the timing

of the progress of threads and of progress relative to each other is undetermined. No assumptions can be made about a specific timing. As a result, the order in which threads interact is not predictable. The nondeterminism is the reason for the general irreproducibility of concurrent program behaviour Thus a concurrent program can deliver differing results when repeatedly run on the same input data but need not.

Today, most commercially available object-oriented programming environments support *concurrent programming*. For example, C/C++ provides libraries for concurrent programming [But97]. Relatively new is the integration of concurrency into a commercial language itself, such as in Ada95 [Ada95, Nag03], Modula-3 [Mod] or Java [GJSB00]. Non-commercial languages integrating concurrency already have a longer tradition. Actor languages [AWY93] were among the first to integrate concurrency with objects, providing active objects or actors. These languages hide many lowlevel issues of thread control as threads are automatically instantiated and deleted. The above commercial languages were following these ideas by merging threads of control with the object-oriented paradigm while keeping a more explicit and lowlevel style of control.

The most important role among the commercial languages plays *Java*, which is increasingly gaining importance in academia and industry mainly owing to its platform independence and its support for distributed and internet computing. The tight integration of concurrency features into Java emphasises the importance of concurrent programming.

Concurrency is required for *domains* such as graphical user interfaces (GUI), application frameworks, distributed systems, client/server-computing, embedded systems, and heterogeneous component-based systems. The integration of concurrency in Java will have the effect that more and more complex concurrent applications will be programmed in Java besides small-scale Java programs such as web applets. Concurrency does not only play a role for complex applications. It is also used in simple Java web applets. For instance, a time-consuming download is executed in a separate thread because it is not desirable that the applet's main thread is blocked until a download is either completed or aborted. A large number of Java programmers is therefore confronted with concurrent programming. Not only do they have to be trained for this but also software development tools should support their activities.

The concurrency features of Java are often regarded as insufficiently designed [Hol00, Lea03]. The recurring complaints by expert users have had the effect that the features are still evolving, although at a slow pace. While previous versions only made small amendments such as deprecating features, the upcoming version of Java 1.5 adds a few new features [Jav04] in the form of

dedicated classes for more high-level synchronisation concepts, for instances, a lock class. Independent providers have also started to develop more sophisticated libraries [Lea03]. Neither from the additional concepts nor from the libraries it can be expected that they provide error-free concurrent programming or that they completely replace the use of basic concurrency features of Java. Therefore, support for programming with the basic Java concurrency features will always be needed.

### Concurrency Errors

Concurrent programs have a high potential for errors specifically related to concurrency. These errors are not application-specific but can appear in any concurrent program in general and are caused by the programmer. These errors are in the code but they can be successfully compiled. Only when the program is executed, the error has the effect that the program does not behave as desired. Typically, the program does not crash due to the effect of the error but merely exhibits unwanted behaviour. Then often the only possibility is to abort the program.

Concurrency errors can be classified according to their effects on the running program. *Safety* errors have the effect that data becomes inconsistent as a consequence of simultaneous access of data from concurrent threads. *Liveness* errors have the effect that one or more thread stops executing but the rest of the program continues. Those threads stop as a consequence of synchronised interaction with other threads. The unwanted effect in a running program which is caused by an error in the code is also called *failure*. It is often the case that safety and liveness failures have no immediate effect on the system. Only after a while either the system may detect or the user may experience the effects of inconsistencies or blocked computations.

Safety and liveness failures can be very severe. A so-called safety-critical system that acts in an incorrect and unexpected way due to an inconsistency or that does not react any longer due to a liveness error can be a danger to human beings. An erroneous production control system can damage goods or delay delivery. Also non-safety-critical erroneous software can cause severe problems when corrupting data or when not responding to requests.

Safety and liveness failures are a serious problem in any concurrent programming language because of the inherent nondeterminism. An safety or a liveness problem does not necessarily occur in every execution of the program. That makes it extremely difficult to find these failures.

In this thesis we will focus on concurrency liveness failures in Java programs. As they are a consequence of using synchronisation they require different concepts than safety failures. Well-known liveness failures are *dead-*

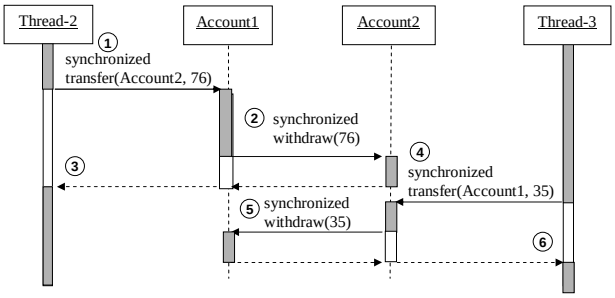


Figure 2.1: Banking Example

*locks*, *nested monitor lockouts* and other monitor problems. Other liveness failures have been rarely documented or not at all. A comprehensive and precise description of liveness failures is still missing for Java.

We distinguish liveness failure due to synchronisation from failures where threads do not make progress due to the *scheduler*. Each Java Virtual Machine has a scheduler component which assigns CPU time to threads, e.g. by time-slicing. The scheduler has a strategy, which it uses to determine when to assign which thread to the CPU. Such a strategy can be unfair and it can therefore have the effect, that a thread is never assigned CPU. This has the same effect on a running programming as one of the liveness failures described above. The failures stemming from scheduling are not our focus and they need to be dealt with differently than those stemming from synchronisation concepts.

### Example

A typical liveness failure is a *deadlock*. A deadlock is an execution state in which threads are blocked mutually waiting for one's other resources. These resources are never released. This state is the result of a specific sequence of interactions between threads. We illustrate this situation with an example. To describe a multithreaded sequence of actions in a program execution we can use a UML *sequence diagram* [UMLa].

We start by looking at an example of a successful bank transfer, a typical scenario in a banking application (see Fig. 2.1). Two threads of control concurrently transfer money between two bank accounts in opposite directions. The sequence of actions in this specific scenario is as follows.

1. The thread object Thread-2 calls transfer(Account2, 76) on Account1 to transfer an amount of 76 units of currency from Account2 to Account1.

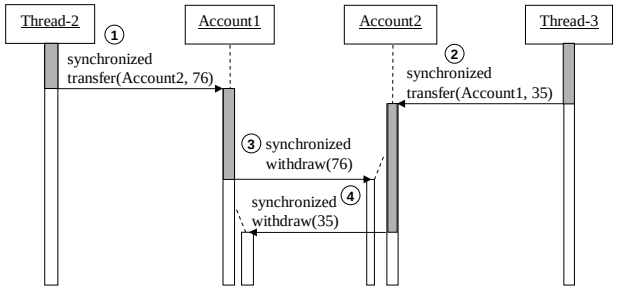


Figure 2.2: Banking Example with Deadlock

2. Account1 calls withdraw(76) on Account2 to withdraw the amount from Account2.

3. When withdraw(76) returns, the amount withdrawn is added to Account1 and transfer() returns.

4. Another thread object named Thread-3 calls transfer(Account1, 35) on the object Account2 to transfer an amount of 35 units of currency from Account1 to Account2.

5. This triggers a withdraw of 35 units from Account1.

6. Again, withdraw() and transfer() return.

The methods transfer() and withdraw() both change an object's state. In order to avoid inconsistencies simultaneous calls to transfer() and withdraw() on the same object have to be *mutually exclusive*, i.e. only one call can be executed at a time on the same object. The prefix synchronized (see Fig. 2.1) denotes that a method has been defined as being mutually exclusive and can only be executed mutually exclusive with all other methods specified as synchronized when executed on the same object. Here, not only a single method but an entire object is a mutually exclusive resource, which can be assigned exclusively to at most one calling thread. When a mutual exclusive method call is granted access to an object we also say that it locks the object. When this call returns the lock is released. When a thread calls a mutually exclusive method on an object which is already locked, it keeps attempting to acquire the lock and cannot progress until it receives the lock. A thread waiting for a lock resource is hence waiting for the thread having this lock to make progress and release the lock.

The result of the activities of the threads in the above example depends on the relative timing of the threads, i.e. on the order in which method calls for each shared object occur. In Fig. 2.1, both calls to transfer() return

successfully because the first thread has completed the call to withdraw() on Account2 before the second thread calls transfer() on Account2. Figure 2.2 gives an example with a different execution order.

1. Thread-2 calls transfer(Account2, 76) thereby locking the first account.
2. Thread-3 calls transfer(Account1, 35) thereby locking the second account.
3. Thread-2 calls withdraw(76) trying to lock the second account and is blocked.
4. Thread-3 calls withdraw(35) trying to lock the first account and is blocked.

Because each account is already locked by a thread both threads are blocked. They cannot proceed without the locks and thus they are waiting for the release of the locks. The blocking of threads on synchronized-calls is indicated by the two additional empty activation bars (see Fig. 2.2). Without the locks the threads cannot complete withdraw(). Furthermore, they will never release the locks which they were granted when calling transfer().

This situation is called a *deadlock* because of the cyclic dependencies: Thread-2 holds a lock on Account1 and is blocked while waiting for the lock of Account2 which is held by Thread-3 which in turn is blocked waiting for the lock of Account1.

A typical means for depicting a deadlock are the following graphs in Fig. 2.3 which have been proposed by [Tan97] to illustrate deadlocks in operating systems. The graph on the left hand side is a so-called *resource dependency graph* depicting which threads hold which resources and wait for which resource. The graph on the right hand side abstracts from the resources involved and is called a *wait-for graph*. It describes which thread is waiting for another thread to release the resource.

Note that the deadlock in a Java program cannot be resolved except by externally aborting the complete program. The deadlock is a failure which has been programmed into the code by the programmer and hence can be avoided on the programming level. In Java, a deadlock can happen exactly as described here.

Note that the two different execution orders in Fig. 2.1 and Fig. 2.2 result from the very same code. The code has a *potential* for a deadlock, and it is also possible to see that potential in a successful execution such as in Fig. 2.1. Here, objects can be locked in an arbitrary order which always bears the potential of cyclic dependencies.

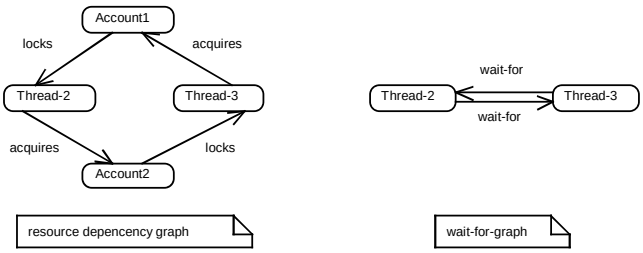


Figure 2.3: Cyclic Resource Dependency and Wait-For Graph

### Liveness Failures Characteristics

The characteristics of the deadlock are typical for other liveness failures, too, such as nested monitor lockout or notification problems, which will both be discussed later. The deadlock characteristics also illustrate the inherent complexity of these kind of failures. As an example consider the deadlock from Fig. 2.2. A deadlock can be categorised according to the following dimensions:

- *Dynamicity*: The deadlock is a state in a running program.

  In the example, the deadlock is only manifest, when withdraw is called for the second time and blocks.

- *Threads*: The deadlock involves more than one thread.

  In the example, the two threads involved are Thread-2 and Thread-3.

- *Interactions:* The deadlock involves more than one interaction per thread involved.

  The interactions involved from Thread-2 are synchronized transfer() and synchronized withdraw(), for Thread-3 analogously.

- *Timing*: The deadlock develops over a certain time span of the program execution and is dependent on the specific timing of the execution order during that time. More precisely, it is dependent on the relative timing of interactions on objects shared between threads.

  In the example, between the first interaction and the second interaction of each thread, there can be an arbitrarily long elapse of time. For the deadlock to occur it is only necessary that the second call to synchronized transfer() on Account2 takes place before the call to synchronized withdraw() on Account2.

- *Dependencies:* The interactions create dependencies such that progress of a thread depends on a resource from another thread which is not released.

  In the example, each thread needs a resource held by another thread.

The dependencies introduced are depicted by the graphs in Fig. 2.3. Either these can be expressed directly as dependencies between threads, or they can be expressed as dependencies between threads and shared objects.

Note that such a deadlock state does not completely describe the order of execution leading into the error. From the given deadlock state we can only derive a partial order. The complete order is missing. Knowledge about the complete order can help to develop an understanding of how the deadlock developed over time.

## 2.2 Problem Description

Dealing with the deadlock and other concurrent liveness failures requires knowledge about all the possible failures and also about their potential indicators in successful executions. The literature does not provide a comprehensive list of concurrency failures in Java. A good starting point for a description of concurrency failures in Java is [Lea00], which informally describes the most important liveness failures.

Knowledge about possible failures, failure-prone executions, and the reasons common to them, is required during testing and debugging. Therefore, the tools used for these tasks need to be able to deal with concurrency failures. Existing tools as found in integrated development environments (IDEs) like debuggers do not provide adequate support for the programmer to detect concurrency failures.

The achievements and deficits of debuggers will be illustrated in depth in the following. We will also illustrate the widespread technique of print-line statements used in addition to or instead of a debugger. We shortly discuss other approaches covering dynamic analysis and static analysis. For improving tool support also a more formal documentation of liveness failures is needed which is still missing.

### Detecting Concurrency Errors

Also in concurrent programming, it is best practice to extensively test a program. The purpose of *testing* is to validate test cases and to discover deviation from them in order to make failures visible. Because of the nondeterminism, testing aims at covering as many different executions of the program

as possible, either by nondeterministic approaches based on inserting arbitrary delays or by deterministic approaches which can execute a concurrent program based on a predefined timing of interactions [CT91, BT98].

Besides testing, programmers often check their code immediately after writing it, using ad hoc test cases. This is called a *code-debug-fix cycle*.

For both approaches it is required to be able to detect and classify failures. Note that by *detection* we mean identifying that there is a failure in a running program or in information gathered from a running program and identifying the kind of failure it is. The term *debugging* does imply detection but goes further. The goal of debugging is to localise the fault area of the code and to develop an understanding of the program so that an adequate correction can be made [FR01]. It therefore includes the steps taken after it has been detected that the program behaves not as specified or as expected and describes the activity of locating the source code causing this failure.

The generation of test cases is not part of the debugging and also not our focus here. For testing Java, there are tools such as JTest [JTe], JUnit [JUn], or even extensions which try to cover also concurrency such as JMThreadUnit [JMT].

### Debuggers

For support in testing, detecting, debugging and understanding failures, integrated software development environments (IDEs) provide *debuggers* with the following support.

- The concept of a debugger is to execute a program *stepwise*. After each step the *program state* is presented. The state consists of heap and call stack.
- The granularity of steps ranges from single statements to user-defined steps between breakpoints and watchpoints.
- Debuggers automatically support detection of address violations or division by zero.

Existing debuggers differ little in these concepts but mainly in the comfort they provide for stepping through the program and for navigating the state [ZK00]. Traditionally, debuggers do not cover concurrency related problems because they were invented for sequential programs.

Using a debugger, a programmer still has to search manually for most errors, especially concurrency ones. The programmer invents and checks hypotheses about errors. Both is usually based on experience and intuition.

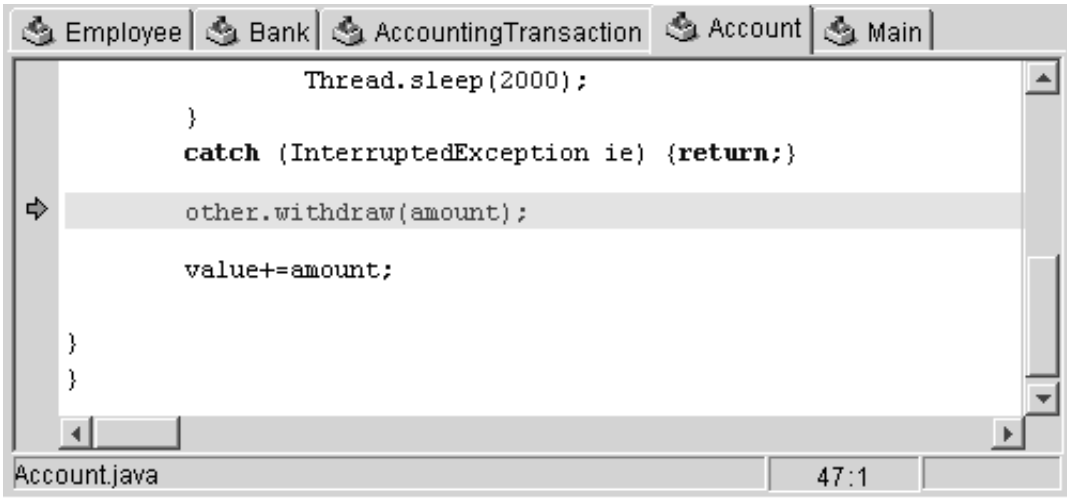


Figure 2.4: Source Code View

The programmer looks for certain patterns in the execution history and in the code. Proving a hypothesis is time-consuming because typically it is not even known when exactly the error has happened. Breakpoints have to be set well before the error is assumed to happen, and then the program has to be executed stepwise. Each state has to be carefully examined by the programmer for identifying known patterns and new situations.

For many decades, debuggers have focused solely on sequential programs. Since the advent of commercial concurrent programming languages like Ada or Java, debuggers have started to make a gradual transition towards concurrent programs. They have been extended to cover also multiple threads. The presentation of program state covers heap, stack, and call stacks for individual threads. The stepping mechanism including breakpoints and watchpoints works on thread-level. Individual threads can be started and stopped while others keep running. The realisation of these concepts do not vary a lot across different languages and tools for the same language. For Java, these concepts have been integrated in IDEs, such as JBuilder [JBd01], VisualAge [Vis00], JDeveloper [JDv01], or Eclipse [Ecl]. In more recent versions of these IDEs, detection of deadlock has become available such as in JBuilder [JBd01]. In this respect, Java IDEs are not behind, compared to other languages. In response to the findings presented in the introduction, COMPAQ started a project to provide deadlock detection in its C/C++ IDE for the Pthread library only in the year 2000 [Har00]. Also recently, a few standalone dynamic analysis tools for concurrency errors have come into existence, which cover also deadlocks such as JProbe Threadalyzer [JPr00].

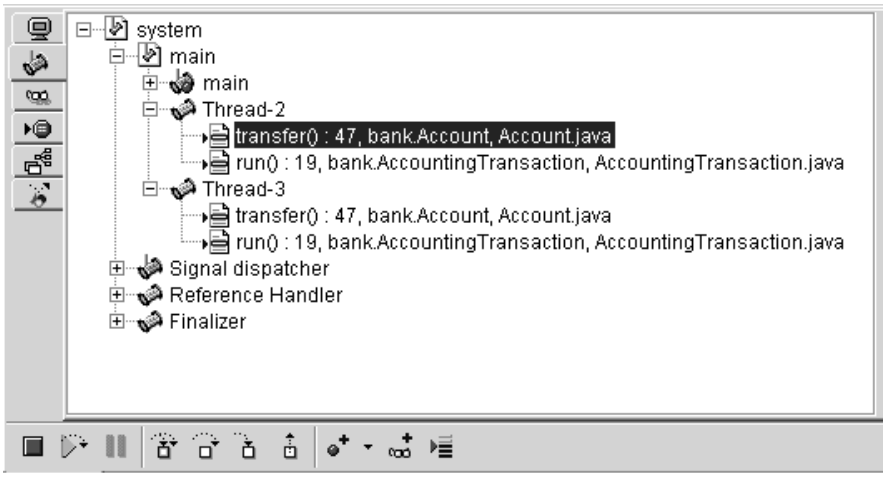


Figure 2.5: Call Stack View

**Example**

To illustrate the existing capabilities and concepts for analysing concurrency errors we have implemented the deadlock-prone bank transfer example from the previous section in a simple Java program, which we execute with the debugger of the JBuilder IDE for Java [JBd01]. JBuilder only serves as an example for a typical debugger in a Java IDE. Other IDEs like Visual Age [Vis00], JDeveloper [JDv01], Eclipse [Ecl] or even the round-trip UML CASE tool Together [Tog] provide similar debuggers. The following three JBuilder IDE views are relevant for describing an execution state and, in particular, a deadlock.

- A view of the source code with a cursor indicating the position of each of the concurrent *control flows* (see Fig. 2.4).
- A view of the *call stacks* of all threads. The call stack of each thread is an ordered list of method calls. The topmost method call in this list is the last one put on the call stack and thereby the active one. A synchronized-method call is only put on the call stack after the method has been successfully entered, i.e. after the corresponding lock has been granted. For each method call the line of code where it is defined is given, the class to which the method belongs, and the package of that class (see Fig. 2.5).
- A view of the mutual exclusive objects, which are also called *monitors*. For each such object, the thread which has access and a list of threads which are waiting for access are given. In the example (see Fig. 2.6),

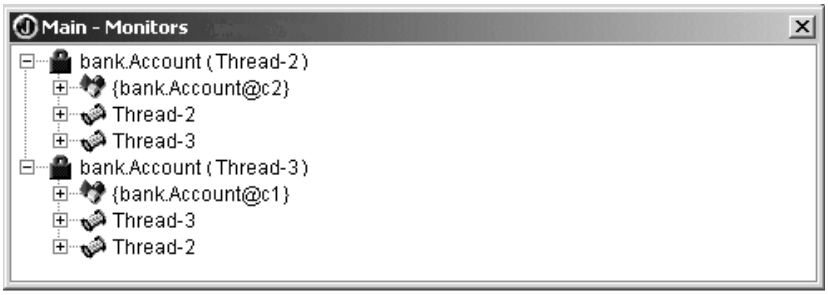


Figure 2.6: Monitor View

Thread-2 has the access for the bank account c2 and Thread-3 is waiting for access, indicated by being second in the list of thread icons. This view indicates a deadlock by highlighting the accessed objects involved in the deadlock. Each thread can be extended to its call stack similar to the second view (see Fig. 2.7).

In the following, we examine how our example is presented along these views. We examine a run of the example code which leads into a deadlock. In our example the *source code view* (see Fig. 2.4) shows the control flow position of a thread by highlighting the line other.withdraw(amount) in the code of class Account.

In the *second view* (see Fig. 2.5), we see the call stack of the two threads which transfer money between accounts. Each thread was started by a call to run(), therefore the bottommost method on each call stack is run(), defined in line 19. The next method on the call stack of each thread is transfer(), defined in line 47. In one of the call stacks, the last method entered has been highlighted, which is transfer(). This is used to denote a correspondence with the first view. The thread in whose call stack the method was called, here Thread-2, is the one which owns the control flow depicted in the first view.

The *third view* (see Fig. 2.6) presents the synchronized objects and the actual deadlock. The synchronized objects in this program are Account c1 and Account c2. For each of these account objects, the locking dependencies are provided. Each object is depicted with a lock icon and a description of its class. In brackets, the thread holding the lock is depicted. The internal ID of the object is given in the next line. We see that Account c2 is locked by Thread-2. After that, the thread holding the lock is depicted again before all threads waiting for the lock are depicted. Here, Thread-3 is blocked waiting for the lock. Similarly, Account c1 is locked by Thread-3 and Thread-2 is waiting for the lock. Each thread depicted can be extended by its call stack similar to the second view (see Fig. 2.7). This third view shows the threads

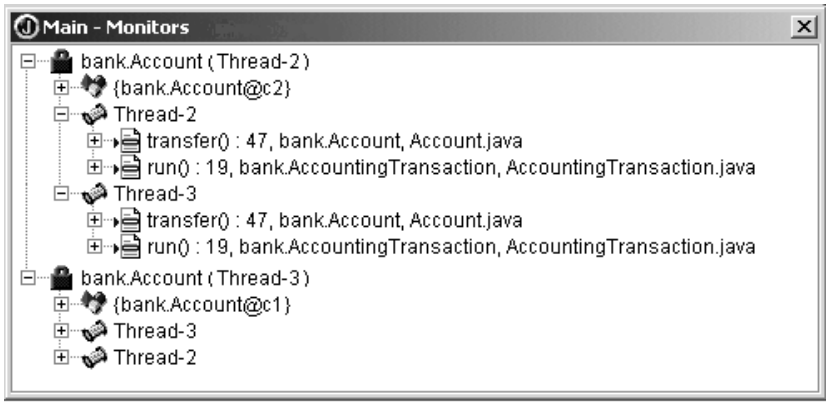


Figure 2.7: Monitor View with Callstacks

and objects involved in the deadlock and their mutual dependencies. The same holds for the extended monitor view. This information is not covered by the other views.

The information about the interactions involved is scattered over two views. The call stacks only show the last successfully entered method which is transfer() in each case. The call stacks do not show which method call is the one which is blocked. This information has to be deduced from the control flow view(s). They are indicating the control flow position, which is other.withdraw() for each thread (see Fig. 2.4).

The execution order is not covered by these views. Though each call stack shows at least the execution order of nested interactions, the call stacks do *not show the relative order between any two call stacks* because the threads are only presented in isolation. The information can again be deduced based on the locking dependencies each call has produced. These locking dependencies restrict the possible execution order. Thereby one can derive the order in which shared objects where accessed.

So far we have described only a snapshot of the JBuider IDE views. We have not explained how these views were generated. To generate these views, breakpoints had to be inserted in each thread before the code involved in the deadlock. Then one had to stepwise execute the program until the deadlock was reached. While this sounds simple, it is only simple for such a contrived example. We knew which threads would get involved in the deadlock and we could select these threads for stepwise execution, however not all at the same time. JBuilder allows only to stepwise execute one thread at a time, which does not provide enough flexibility if one does not know what to look for.

### Debugger Deficits

Historically, debuggers did not provide integration of execution states from different control flows. Their very powerful means for exploring the execution state of a program in detail including heap, stack and the call stack were sufficient for sequential programs. When making the transition to concurrent programs, they did not integrate concepts for a coherent presentation. It is obviously not sufficient to introduce a separate single-state presentations for each new thread as in most debuggers. Another effect of not dealing with concurrency features is the fact, that call stacks do not contain method calls which have been entered but are blocked due to a synchronized. Therefore, debuggers are not able to show that a synchronized-method was called but that the thread was blocked. Another problem is the transient memory of a debugger. Neither a single state nor the history of an execution can be stored. Once the observation of an execution is completed, also the information is gone and therefore the execution order cannot be accessed any longer.

To summarise, two main problems have arisen when examining the concepts of debuggers for the detection and particularly for the understanding of the development of a deadlock.

- Inadequate representation of the interactions involved in a deadlock.
- Missing representation of relative execution order, i.e. the timing on shared objects.

### Manually Generating an Execution History

To overcome the lack of an execution history and to avoid an incomplete reconstruction of it after the execution has taken place, it has always been a widespread debugging practice to manually insert *printline statements* into the source code, for sequential programs as well as for concurrent programs. Although this observation is commonly acknowledged, empirical evidence is rare [Lew03a, ACM97]. This practice shows that programmers have always felt the need for generating and viewing information about the execution history. The output generated provides a time-ordered sequence of program execution steps and is called a *trace*. It should not be confused with the trace-command in a classical debugger, which selects a function or method at which the debugger stops each time it is called.

Using printlines not only saves time compared to stepping through a program with a debugger but also has the advantage that the output is more compressed than the information presented on the debugger screen. Also, this approach is completely tool-independent. Moreover, the information is highly

```
1:Employee@ID#90:Terminal@ID#-1:handOverCheque:false:Exit
2:AccountingTransaction@ID#97:AccountingTransaction@ID#97:run():false:Enter
3:AccountingTransaction@ID#97:Account@ID#79:transfer:true:Acquire
4:AccountingTransaction@ID#108:Account@ID#80:transfer:true:Acquire
5:AccountingTransaction@ID#108:Account@ID#80:transfer(Account(id=79), long 34):true:Enter
6:AccountingTransaction@ID#97:Account@ID#80:withdraw:true:Acquire
7:AccountingTransaction@ID#111:AccountingTransaction@ID#111:run():false:Enter
8:AccountingTransaction@ID#111:Account@ID#80:transfer:true:Acquire
9:AccountingTransaction@ID#108:Account@ID#79:withdraw:true:Acquire
```

Figure 2.8: Trace Example

customisable with regard to the kind and amount of information collected. For instance, one could either choose more printlines with less information or fewer printlines with very detailed information about the program state.

Fig. 2.8 is an example of a Java trace observed during a program run. This kind of output could have been created manually by using printline statements, although typically one would not attempt such a detailed output manually. One line in the trace corresponds to one execution step also called *event*. The information for each event is separated by ":". The first entry is the line number of the trace itself. The second entry is the thread, identified by its class and its object-ID, the third entry is the class and the ID of the object on which the method is called. In case of a method exit, the callee is not identified, which is denoted by a "-1". The next entry is the name of the method called. The following entry is "true" for a synchronized-method and "false" otherwise. The last entry distinguishes method entry, exit or attempt to acquire a lock. Fig. 2.8 is only an excerpt from a longer trace. Typically, the classifier names are preceded by host names and package paths but these elements are omitted here for reasons of presentation.

The deadlock is related to lines 5, 6, 8, 9 in Fig. 2.8. The thread with ID 97 of class AccountingTransaction acquires a lock, which is locked by the thread with the ID 108. However, the thread with ID 108 acquires a lock for ID 79, which is locked by the thread with ID 97.

Manually inserting printline statements into the code also bears disadvantages. It is not only cumbersome but also error-prone in two respects.

- During the insertion of required printlines, important branches of the code can be accidentally omitted.
- The information gathered with printline can be insufficient for a foreseen or unforeseen purpose.

There is not only a risk that important information be missed but also that incomplete output be misinterpreted. In the presence of threads, it is

particularly easy to create a misleading hypothesis. Besides that, printline statements generate textual output that is difficult to read and understand.

Having demonstrated the advantages and drawbacks of manual generation of execution histories, it is obvious that an automated generation is desirable to overcome the drawbacks. However, without thoughtful selection, one could easily generate a large amount of trace data. This requires to also analyse traces automatically because, often, only a small part of the trace contains the failure.

### Other Approaches based on Dynamic Analysis

So far we have discussed debuggers which are integral parts of IDEs. There are also some standalone tools which do not provide the complete functionality of debuggers but instead provide thread-specific dynamic analysis. They focus on profiling, i.e. resource and performance analysis, as well as on detection of a few well-known failures. For instance, the *JProbe Threadalyzer* [JPr00] provides deadlock detection besides profiling [JPr00]. The *Assure Thread Analyzer* [Ass] also provides profiling and deadlock detection. These tools do not put a special emphasis on the presentation of detected failures. To the best of our knowledge, none of these tools is dedicated to an exhaustive analysis of liveness problems. Both tools also detect safety failures.

Test-tools like the commercial product JUnit [JUn] or the open source tool JTest [JTe] analyse coverage during testing. They put an emphasis on automatically generating test cases for object-oriented testing but not for concurrency. Therefore, they do not cover liveness failures or safety failures.

### Formal Methods

Besides the best practice to perform extensive testing, there are also static analysis approaches, which aim at proving properties of concurrent programs. These approaches are either based on source code or on a model of a program.

One of the most sophisticated approaches is the *Bandera Tool Set* [Ban04, CDH$^+$00, HD01], which is an integrated collection of program analysis, transformation, and visualisation components designed to facilitate experimentation with model-checking Java source code. Bandera takes as input Java source code and a software requirement formalised in Bandera's temporal specification language. It generates a program model and specification in the input language of one of several existing model-checking tools. For deriving models from source code, Bandera offers a variety of techniques such as slicing and abstract interpretation. Thereby it wants to leverage the use of model-checking for ordinary software development instead of its previous

use only in dedicated settings, e.g. for safety-critical systems.

Finite-state verification techniques, such as model checking, are attractive because they are capable of exposing very subtle defects in the logic of sequential and concurrent systems. However, it is not feasible to build finite-state models for real-size applications, only for selected parts. Also, bridging the semantic gap between a non-finite-state software system expressed as source code and those tool input languages requires the application of sophisticated program analysis, abstraction, and transformation techniques. The successful choice of techniques requires training and technical background. Therefore, model size and missing expertise are the two main impediments for applying model checking in every day software development.

Formal methods are also applied to check models from which software can be derived following a methodology which keeps the desirable properties. An approach based on model-checking, *Labelled Transition Systems*, has been proposed in [MK99]. This approach suffers also from the problems discussed above.

Because of model explosion and training overhead, alternatives to improve testing are still needed and can be used complementary to static analysis.

#### Prerequisites for Improving Tool Support

In order to implement failure detection, precise descriptions of failures are required. They are needed to determine what kind of data has to be accessed for the failure detection. They are also needed to describe suitable detection algorithms and they are needed to define how results are to be presented.

Classifications of precise descriptions can help to exploit the potential commonalities of failures. During the implementation of failure detection tools this supports reuse of common detection principles for a set of failures.

Precise descriptions are still missing for most Java liveness failures. Literature only provides incomplete informal or exemplary descriptions such as in [Lea97, Lea00, Hol98].

## 2.3 Goals

In the problem description two areas have been identified where the state-of-the art is not satisfying, the description of concurrent liveness failures and their practical support with tools.

First of all, we want to contribute to a better understanding of concurrency failures by

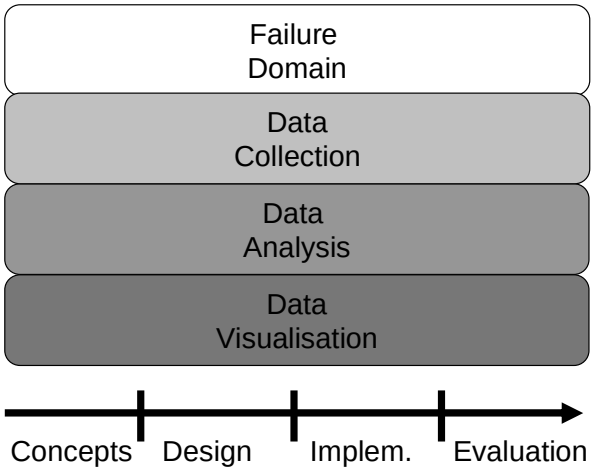


Figure 2.9: Thesis Goals

- systematically and precisely describing the *domain* of Java liveness failures.

Starting from a precise description we want to provide

- concepts for *collecting data* from running Java programs,
- concepts for *automated detection* of Java liveness failures, and
- concepts for *visualisation* of Java liveness failures.

These four goals are depicted on the vertical axis of Fig. 2.9. On the horizontal axis we have depicted the steps which have to be taken to put the concepts to work by designing and implementing corresponding prototypes, which will finally be assed for their usefulness.

The intended prototype shall support the programmer in the detection of liveness failures. Therefore, the detection of liveness failure is intended to be fully automatic and should not require particular background knowledge. The analysis tool shall record test cases or debug sessions and will only give feedback if a failure is found. When a failure is found the programmer is notified. If nothing is found the programmer does not necessarily have to be aware that the automated detection is carried out in the background.

In the next chapters, we will derive detailed requirements from the above listed goals.

## 2.4 Synopsis

In chapter 3 we derive our requirements and outline our approach. In chapter 4 we discuss the concurrency approach taken by Java in greater detail.

In chapter 5 we provide an informal description and classification of liveness failures. In chapter 6 we provide a model of thread synchronisation, which we use to formalise liveness failures.

In chapter 7 we discuss the state-of-the-art in tracing and develop a tracing prototype. In chapter 8 we present selected algorithms for dynamic analysis of failures. In chapter 9 we discuss existing software visualisation approaches and develop a visualisation for concurrent Java and its failures.

In chapter 10 we present the developed prototypes with examples. In chapter 11 we conclude and in chapter 12 we give an outlook.

# Chapter 3

# Requirements for Automated Failure Detection

In this chapter, we will determine the requirements for automated detection of concurrent liveness failures and we will outline our approach. For this purpose we will look again at the main goals for our approach, which we have identified in the last chapter:

- The goal to precisely describe and classify concurrent liveness failures.
- The goal to provide concepts for automatically detecting these failures in a running program.

The first goal is not only important in order to improve the general understanding of concurrent liveness failures but it is also a prerequisite for the second goal. The reason is that the characteristics of the failures determine what information is needed for detecting them and this in turn determines how the detection results should be presented. Because of this dependency we cannot derive detailed requirements for the automated detection until we have described the failures in depth, which will take place in chapter 5. Here, we outline the requirements based on the insights into failures we have gained so far from the motivation. This level of abstraction will suffice for this chapter to describe our requirements and the overall approach.

In section 3.1, we will refine the requirements for a precise description of the failures. Here we will address the issue of classification and formalisation. In section 3.2, we will describe the requirements for automated detection, which consist of determining how the information from a running program is accessed, how it is analysed, and how the analysis is presented. The entire structure of the detection process is determined by the intended usage scenario(s) which will also be discussed here. Section 3.3 summarises our requirements.

## 3.1 Failure Description and Classification

Here we give an overview of the requirements for the precise decription of the failure domain. Thesse requirements are discussed from two perspectives.

First of all we consider the *purpose* of the precise description which has an influence on the choice of the means for the precise decription. Then we consider the *scope* which has to take into account the characteristics of failures.

### 3.1.1 Purpose

For the first goal we require a precise and systematic description of the concurrent liveness failures in Java. This description must support

- the development of a better understanding of the failure domain, and
- the development of definitions which can be used in detecting these failures, i.e. definitions from which patterns or algorithms can be derived to detect failures.

For a better understanding of the failure systematics, it is advisable to identify commonalities and classify failures accordingly. It is desirable that the results from analysing the domain of failures are captured in a clear and unambiguous way. Here, an intuitive formalism can support our goals but it should also be accompagnied by an informal description in natural language. Also, it is useful to base the description of errors on well-known classifications and terminology for errors rsp. failures.

For the development of detection mechanisms, which shall be implemented in a tool, it is even more important to define failures in a formal way. There are different levels of formality. For the goals of this approach, namely automated failure detection and visualisation, we consider it dispensable that the formalism be executable or that the detection part be accomplished by a formal tool. Instead, we strive for a straightforward independent implementation for validating our ideas.

### 3.1.2 Scope

As mentioned in the beginning of this chapter, the failure characteristics determine the requirements for automated support. At this point, we can only give an overview of these characteristics, which will be detailed later:

- Failures are situations which are observed while a program is running, a characteristic which is already captured by the term *failure*.

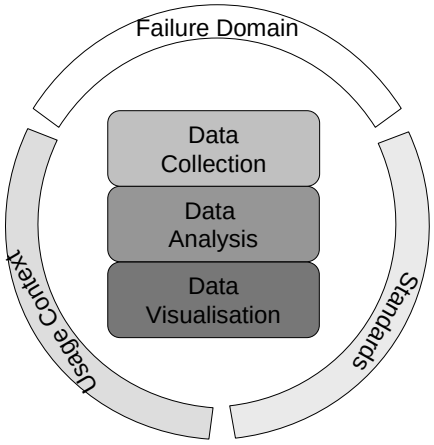


Figure 3.1: Requirements for Automated Failure Detection

- Typically, concurrency failures involve more than one thread and develop over time, involving one or more interaction from each thread involved. These interactions create dependencies among the threads.

Note that all these characteristics were present in the deadlock example in the previous chapter.

The characteristics of the failure domain determine essential features of the desired tool support. From the characteristics it follows that the description of the failures has to cover program dynamics, thread interactions, and thread dependencies. More details on the failure characteristics will be given in chapters 5 and 6.

## 3.2 Automated Detection

When speaking about software, the general idea behind automation is that a software system performs a computation or analysis on a kind of data and informs about the result. This very coarse definition already identifies the three key tasks involved in automation, along which we have to organise our requirements:

- Collecting the input for the analysis *(Data Collection)*.

- Performing the analysis task *(Data Analysis)*.

- Transforming output results into a suitable visualisation *(Data Visualisation)*.

In our case, the analysis task deals with the detection of the concurrency liveness failures. Conceptually, we consider the collection of the input as part of the overall approach, although it might not always have to be covered by a tool, e.g. if collected data already exists. The three tasks are depicted in the centre of Fig. 3.1. Even though the input data depends in its choice on the intended analysis, we will stick to the above order as it is the order in which these tasks depend on each other from the data flow perspective. Note that the above order does not imply that each of these tasks can only be performed once. Indeed, it is possible to perform them incrementally.

The failure characteristics shortly sketched in the previous section determine the analysis task, and in turn determine the necessary input and the resulting output. The failure characteristics are depicted by the arc labelled *Failure Domain* on the outer circle in Fig. 3.1. The outer circle denotes the influence of its arcs on the three key tasks depicted in the centre.

Also, for automated support it is important to reflect the context in which this support is going to be used. This has an impact on the model of how the software interacts with its surroundings, e.g. a user. This is depicted by the arc labelled *Usage Context* in Fig. 3.1.

Orthogonal to the just mentioned tasks is the fact that it is often desirable to use *standards* or de facto standards and conventions. While this can affect anything from data encoding up to the colours on a screen, here we take a software engineering perspective and address exchange formats and APIs in order to increase interoperability and reuse, and we address notations for visualisation as this can improve usability. Standards are depicted as an individual arc in Fig. 3.1.

The ideas outlined above will be examined in more depth below. Section 3.2.1 discusses the contextual requirements. Section 3.2.2 discusses the collection of input data. Section 3.2.3 discusses the analysis and 3.2.4 the visualisation. Standards will be discussed directly in places where they are related to.

### 3.2.1 Contextual Requirements

In this section, we discuss the requirements from the usage context, for which we are planning the automated support. In general, the context is characterised by a process and that process determines where automated support can be used to replace human activities. The automated support is primarily intended to replace part of the process, but also the process itself might

change due to better tools. The process might already employ tools, which has to be considered when attempting to use new tools.

The context in which the desired automated detection is to be used does not only determine the core functionality but also how this functionality needs to be controlled, how it needs to interact with the context, and how data is stored and displayed. The requirements come therefore from the intended *"use cases"* of the automated detection support.

The automated failure detection can be applied in two different cases.

- Automated failure detection can be used to detect failures in test runs, either during or after the *test run*.
- Automated failure detection can be used to detect failures in *code-debug-fix* cycles.

Both cases have to be reflected in the requirements. Both cases require that data from the test runs rsp. the debug cycles can be accessed. In general, for both cases, concepts and techniques have to be determined. There is a common core to both of them.

User interaction plays a role in each of the two use cases and therefore determines the *user control* over the three tasks data collection, data analysis, and data visualisation. The user has to choose the part from the program and the execution phase of the program from which the information is collected. The user has to decide what to do if a failure is detected. The user needs full control about the visualisation.

Not only the use cases have to be reflected but also which *tools* are already used in these use cases such as testing tools in the first case and debuggers or other dedicated dynamic analysis tools in the second case. Either, one might require that it is possible to exchange data between the existing tools and the new tools, or on might go one step further and require that the detection automation be more tightly integrated with existing tools.

Standards will become an issue when we look at the individual requirements for each task in the following.

### 3.2.2 Data Collection Requirements

This section describes the requirements for collecting input data for the intended analysis. In principle, data collection has to provide support for both of the above (use) cases. For this approach, we reduce the requirements to the smallest common denominator. In both of the above cases we need to collect information from a running Java program. One does not only have to describe the format of the data, but one has to actively collect the input

from a suitable data source. Therefore, regarding the data collection, two requirements arise:

- The data format needs to be determined.
- The method how the corresponding data is collected has to be determined.

Before a method of data collection can be defined, the *data format* has to be determined and therefore will be discussed first. The granularity and the level of abstraction of a format have to be such that the desired failure analysis can be performed. Also, it is desirable to design a more general format for the data collection in order to foster reuse. In the same veine, it is desirable to use formats from testing or be at least compatible with them. This could alleviate the first usage issue, i.e. to use our approach for test cases.

For the *data collection method*, a way has to be found how to access information about the execution steps taking place in the running program. One needs precise information about each statement. One way is to hook into the runtime environment to get this information, i.e. the runtime environment can output data about each statement it executes from a program. The hooks can be part of the API of the runtime environment, or one needs to instrument the runtime environment, i.e. augment its code. Another way is to directly instrument the program by using an automatical transformation. However, it is desirable that the source code need not be changed, in order to test or debug a program which is already running, e.g. a webserver. It is also required that the program is not disturbed by the observation. All software solutions which gather precise data will have an impact on the timing of the observed program, e.g. slow it down.

We have already pointed out that we consider a debugger as too limited for our purpose as it is only able to deal with one state at a time. We require that a history of states is available rsp. a history of events and the effected state changes. We next consider a technique which seems apt for our requirements.

### Tracing

We have seen in the motivation that attempts have already been made by programmers to create histories of events by manually inserting printlines into the code. The problems of manually inserted printlines can be overcome by automating the generation of output as produced by printline statements. This is also called *tracing*. Tracing generates a log of interesting events and

states of a running program. Such a log, which is also called a trace file or simply a *trace*, provides an execution history with the global and relative timing of threads. A trace is therefore typically defined as a time-ordered sequence of event-records [KRR98].

The *trace format* defines the trace information, which has to be gathered, i.e. events and states of a program, and how they are encoded. We have already shown an example trace in the motivation in the previous chapter.

It is also required to provide concepts for *controlling* the tracing process and to interact with it. Here, the challenge is to combine ideas from debugging with tracing such as using breakpoints to determine the part of the program to be traced.

A general problem is always that tracing generates large amount of data. Therefore, means for selective tracing are required.

Standards are an issue as well for the tracing format as for the tracing method. For the format we require either to use an existing format or to use a format transformable into other formats. One should not only consider formats from testing but also "formats" for describing software dynamics such as UML interaction diagrams [UMLa]. They are not far from what we require.

For the tracing method, one should look at APIs in the Java context. The Java Platform Debugger Architecture is a de facto standard for building Java debuggers and it is used by many suppliers of Java IDEs. It allows debuggers to collect information about program execution and to set break- and watchpoints amongst many other typical debugger functions. We require that such an interface is taken into consideration.

### 3.2.3 Data Analysis Requirements

The goal is to use the failure characteristics to determine concepts how failure detection can be implemented. Therefore, we do not only require that the data collection be automatic but also the detection. Automated detection for all typical errors in concurrent traces is desirable.

Regarding the analysis, the most important requirement is that the intended analysis can be computed efficiently. It must be feasible to determine algorithms which detect concurrency liveness failures in a running program. For a trace-based approach as outlined in the previous section, it is required that large amounts of data are examined automatically.

The requirements for the algorithms will be refined after we have described the failures in detail.

### 3.2.4 Failure Visualisation Requirements

It was demonstrated in the motivation with the example of the deadlock that failures span over time, involve different interactions from different threads, and involve dependencies among threads. The failure characteristics determine how the results from the failure detection should be presented to the user. Ideally, all of these dimension should be accomodated simultaneously. This can be achieved best by a *graphical respresentation*. While text can only present one dimension explicit, namely, the sequence, a graphical representation can display at least two dimensions and can encode others using colour, shape, size or even text. Therefore, it can display failures with its several dimension better than a texutal trace. In the motivation we have already extensively used different kinds of graphical visualisations to explain the deadlock.

We cannot learn from debuggers how to present several dimension, especially not how to display time. Tracing differs substantially from state-oriented debuggers and therefore needs the dimension of time or order which was not needed for debuggers.

Apart from the missing information about execution history, the main problem of the debugger is the inadequate presentation of the execution state of a concurrent program. Two views had to be considered to determine the current focus of control in the case of a blocked synchronized method call. Moreover, debuggers present each thread in isolation but threads do not act in isolation. Concurrent programs live on interations between threads from which manifold dependencies arise.

Therefore, new presentations different from the ones found in debuggers have to be devised for tracing and for trace-based error detection.

#### The Unified Modeling Language

Describing behaviour and sequences as required when dealing with the execution history of a program is not a completely new issue in the software development lifecycle. The Unified Modeling Language (UML) [UMLa] has become a standard for visually describing software requirements, analysis, and design documents. Amongst others, it provides diagrams for describing behaviour on object level, in particular, complex object interaction and relationships. This is what is found in traces and what is particularly needed to describe concurrency errors. A visualisation is desirable which combines benefits from the history-oriented UML interaction diagrams with the exact deadlock description from the debugger.

The advantage of using UML is that it is known to developers from pre-

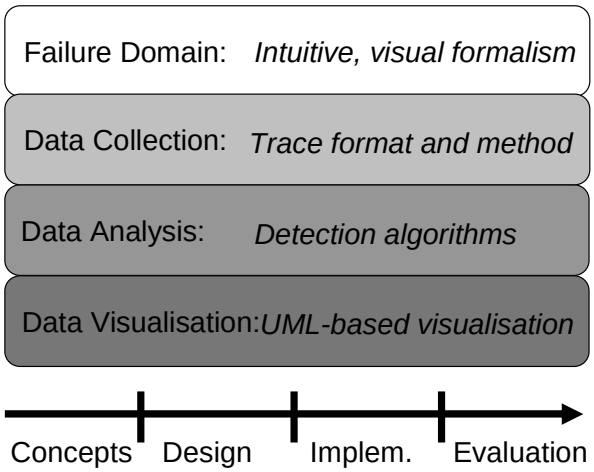


Figure 3.2: First Refinement of Goals

vious phases in the software development lifecycle. Programmers need to be able to read UML design documents. Using UML also during debugging makes a step further towards continuous use of one notation. Scalability is not an issue when we aim at a focused visualisation of failure scenarios using UML. A failure only a involves a limited number of threads, a limited number of shared objects, and a limited number of interactions. Because UML is not primarily designed for visualising program traces or failure detection results, it has to be adapted to capture programming language-specific features. For capturing failures it has to be adapted to capture thread depencencies which play a role in failures. This is made possible through the UML profiling mechanism.

## 3.3 Summary

We summarise the requirements by assigning them to the two goals by which we started this chapter (see also the refinement of goals in Fig. 3.2).
For the goal of describing liveness failures we require

- a natural language description and classification, and
- an intuive, visual formalism, not necessarily executable, from which detection specifications can be derived.

For the goal of automated detection we require

- a data collection using a flexible tracing method and a trace format which supports the detection of liveness failures,

- concepts for implementing detection of the above failures, and
- a UML-based visualisation of traces and failures.

To show the feasibility of our concepts we require to validate these ideas with a prototype. Given the three areas of tool support, for each of these areas a suitable prototype has to be designed and implemented following the decisions which have already been made in this chapter.

# Chapter 4

# Concurrent Programming in Java

In order to improve debugging support and to derive detection characteristics for concurrency failures in Java we need to determine the aspects in concurrency relevant for liveness failures. In this chapter, we introduce the concurrency mechanisms provided by Java. We will motivate the purpose of the concurrency mechanisms with a running example which is an extension of the banking scenario used in the motivation.

The presentation of the Java concurrency mechanisms is based on the official Java language specification (JLS) [GJS97]. This document provides a precise informal description of the syntax and semantics of Java. Not all concurrency mechanisms are covered by the JLS because some belong to standard Java libraries. Then our presentation refers to the official documentation of these libraries.

The Java language is still evolving. Many language features which had been devised have later been *deprecated*. That means that they are still part of the language specification for reasons of downward compatibility but their usage is discouraged. Here, we only describe features which have not been deprecated.

We will present the concurrency mechanisms of Java in great detail because we will formalise many parts of them later.

### The Running Example

The example is a simplified banking application simulating the main components of a bank:

> A bank maintains accounts and engages employees which carry out transfers between these accounts. A transfer withdraws money from

> one account and credits it to another account. All employees work simultaneously. Each employee uses a terminal to enter data for a transfer. Once all data for a transfer have been entered the employee uses the terminal to trigger the execution of the transfer. The employee does not have to wait for the transfer to be completed but can enter new data and trigger new transfers immediately.

In the remainder of this chapter we examine how the example is mapped to Java. Thereby we will introduce step by step multithreading concepts in Java. We will omit intermediate steps of a well-defined software development process such as the design. This chapter is organised according to the stepwise introduction of concepts. In the next section we aim at identifying basic principles of threads and means of interaction between them as provided by Java. In the remainder of this chapter we will present more details of the implementation of the example in Java.

## 4.1 Thread Principles

In this section, we introduce the main principles of Java threads and of interaction among them. We also address nondeterminism, which is inseparable of the notion of concurrency in Java.

### Threads

The textual description of the running example contains hints to inherently concurrent activities, i.e. activities which can take place simultaneously. These are, for instance, the concurrently working employees. Also, entering transfer data, triggering transfers, and transfers themselves are concurrent. Moreover, one can observe that some entities are only active concurrently because they are used by concurrent entities, i.e. a concurrent entity can delegate its activity to another, otherwise passive, entity. For instance, terminals only trigger transfers when they are used by employees.

Conceptually, we can distinguish between

- entities which are the *root and thereby the owner* of a concurrent activity, and
- entities which can be *activitated or used concurrently* but are not the owner of a concurrent activity.

In the example, we identify employees as being the root of a concurrent activity and also transfers because employees can trigger a new transfer

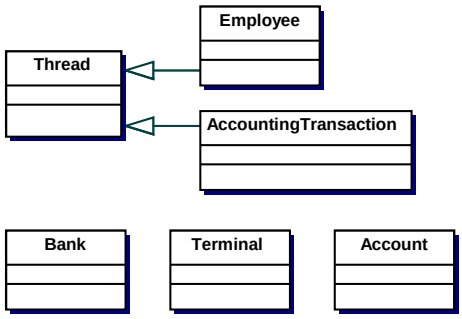


Figure 4.1: Banking Simulation Classes

through a terminal before the previous one has finished. All other entities are not owners of a concurrent activity but they are activitated by owners of a concurrent activity.

Using Java, the conceptual concurrency can be mapped to an implementation level concurrency by means of *threads* [GJS97, Lea00]. A Java thread is a single flow of control which executes its code similar to a sequential program. Concurrent threads execute simultaneously. Each Java thread is also a Java object, in the following referred to as *thread object*. Thread objects are instances of the Java class Thread in the package java.lang.

The conceptual distinction between entities which are the root of a control flow and entities which are only activated by other control flows exists in Java, too. Java thread objects are the root of a control flow and such a control flow can use any other Java objects during its execution.

For our example, we map employees and transfers to Java thread objects. On the type level we map them to Java classes which inherit from class Thread. For employees we specify the class Employee and for transfers we specify the class AccountingTransaction. Those objects not owning a control flow are mapped to normal classes such as classes Bank, Terminal, Account. The resulting classes are depicted in the class diagram of Fig. 4.1, including the inheritance from class Thread.

By providing the concept of concurrent threads, Java frees the programmer from the burden to map the conceptual concurrency to a sequential implementation model. Also, the programmer can potentially take advantage of platforms with multiple CPUs. The Java Virtual Machine (JVM), interpreting the Java byte code, is responsible for mapping the concurrency to multitasking on one ore more CPUs. Concurrent Java programs can run on any JVM and are intended to be as portable as sequential Java programs.

The technical details of how to create thread objects and how to control

their execution will be explained in section 4.2 on the thread lifecycle. In the remainder of this section we will address basic principles of interaction between threads and nondeterminism.

### Interaction

The concurrent activities of the example are not completely independent of each other. For instance, all transfers operate on the same set of accounts, i.e. all accounts of the bank. Another example is that transfers are triggered by employees working with terminals. Conceptually, we can identify two kinds of interactions in this example.

- Activities share data.
- An activity triggers another activity.

Other conceivable interactions are the following.

- An activity delivers data to another activity.
- An activity is waiting for another activity to deliver data, to complete a task, or to finish completely.

Java accounts for the above listed interactions between threads. More complex interaction protocols can be constructed from these kind of interactions. All threads of a Java program run inside the same process and share the same address space. A Java thread is also called a *light-weight* flow of control in opposition to processes running in different address spaces called *heavy-weight* flow of control. The light-weight approach allows threads to share all objects of a program as well as system resources such as files. Threads also exchange data via shared objects. Triggering and waiting also takes place via shared objects.

An interaction which implies a timely coordination between threads is also called *synchronisation*. Synchronisation takes place when threads are triggering each other or when they are waiting for each other for various reasons. Sharing or exchanging data does not imply a synchronisation albeit often synchronisation is needed to prevent data inconsistencies. Synchronisation will be explained in detail in section 4.4 on synchronised interaction.

Unsynchronised but also synchronised interactions bear many problems which will be the subject of subsequent chapters. The reason why any kind of interaction is inherently error prone is the nondeterminism inherent in concurrency.

### Nondeterminism

In the example, it is described that employees work simultaneously. Each action of one employee can take place before, after, or simultaneous with each action from another employee. That is, entering data by using a terminal and triggering the transfers happen without timing constraints with regard to the work of other employees.

When mapping this to Java it is desirable to keep this deliberate indetermination. This is possible because in Java concurrent activities and their parts can, in general, happen in any order with respect to other current activities including overlap.

The paradigm of concurrent threads breaks with the *sequential* paradigm.

- A sequential program specifies a *total ordering* of statements. The order of statements specifies the order of execution. A sequential program has a *deterministic execution order* at runtime and it is *deterministic* with respect to results. Successive statement must be executed one by one without any overlap in time. Given the same input data, a sequential program executes the same sequence of instructions and produces the same result. Considered of its own, a thread has the characteristics of a sequential program.

- A thread is concurrent and nondeterministic only with respect to other threads. Two execution steps from two concurrent threads may either overlap or execute in any order (either one may precede the other). Therefore, concurrent programs are *partially ordered*. Hence, many legal executions are possible. A concurrent program has a *nondeterministic execution order* at runtime. This is also called *nondeterministic interleaving* [Lea00, MK99]. A *scheduler* determines a valid execution order when mapping the threads to one or more CPUs. This scheduling is not predictable for a given program. Therefore a programmer cannot make any assumptions about the absolute or the relative speed of progress of threads. From the point of view of the programmer, a single thread executes in a series of activities as specified by the programmer interspersed with periods of dormancy not specified by the programmer, which occur when the thread is deprived of the CPU.

Note that nondeterministic execution order has to be distinguished from *nondeterministic results*. Different program executions can but do not necessarily produce differing output for the same input. The nondeterminism increases the complexity of a concurrent program and therefore makes it difficult to understand and to find errors in it.

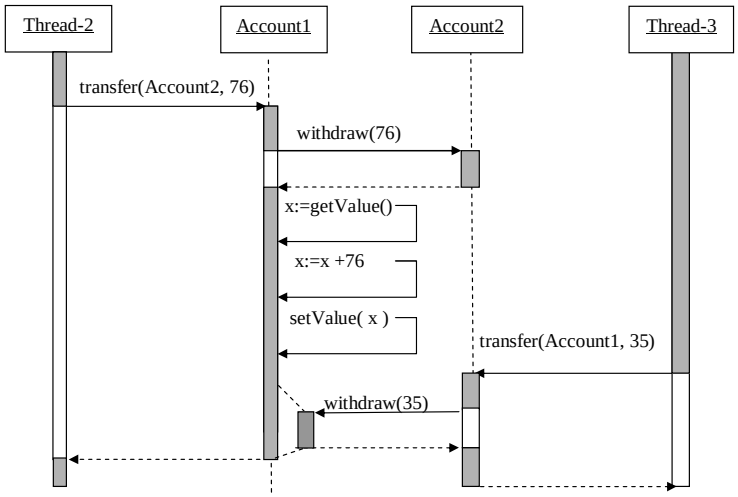


Figure 4.2: Unsynchronised Access of Accounts

We illustrate the nondeterministic execution order with the examples given in Fig. 4.2 and Fig. 4.3. In the examples, two threads transfer money between the same two accounts. It is not predictable in which order the threads access the accounts. In Fig. 4.2, the call transfer(Account2, 76) triggers withdraw(76) on Account2. The current amount of Account1 is queried by getValue() and stored in the local variable x. Then x is set to x + 76 and the new value is stored by calling setValue(x) on Account1. Then, the call transfer(Account1, 35) by the other thread triggers a similar activity which withdraws and adds an amount of 35 to Account2.

In Fig. 4.3, the activities on Account1 interfere. After withdraw(76) on Account2, Account1 is queried by getValue() for the current amount. This value is stored in x. Then, in the control flow of the other thread, withdraw(35) is called on Account1. Then x is set to x + 76. The result of the second withdraw is lost when the the new value of x is stored by calling setValue(x) on Account1.

While the two program runs are an example for nondeterminism they are at the same time an example for its danger. The problem occuring here is also known as *lost update*. The freedom of the nondeterministic execution order is not desirable for the concurrent transfers in our example. Synchronisation, i.e. timely coordination, has the very purpose of introducing temporal constraints between unordered execution of threads and is a means of cutting out unwanted orderings. For example, the concurrent access of accounts can be made mutually exclusive in order to avoid inconsistency of data. While

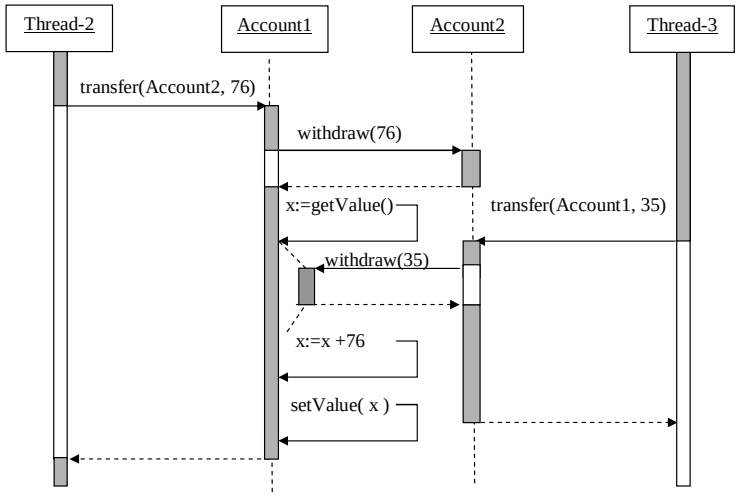


Figure 4.3: Lost Update

synchronisation limits potential orders it is subject to nondeterminism, too. Synchronisation will be explained in detail in section 4.4 on synchronised interaction.

### The Java Memory Model

Chapter 17 of the Java language definition (JLS) [GJSB00] specifies the semantics of Java threads by specifying a set of operations on a memory model and a set of contraints for their occurrence, also called rules. The Java thread semantics are fully described in terms of these model. This model is given in natural language only. We will explicitly discuss the JLS and the memory model in chapter 6. Here we explain how it relates to thread synchronisation from the programmer's perspective.

The JLS does not enforce a total ordering for accessing unsynchronised variables but instead allows partial re-ordering in order to allow compiler optimisations. These issues do not have an effect on the use of the synchronisation concepts on which this approach is based. Therefore, they also are not related to the concurrency failures and potentials. In the presence of synchronisation the Java memory model guarantees sequential consistency and does guarantees the semantics intuitively expected by the programmer.

## 4.2 Thread Lifecycle

In the running example, we can observe that concurrent activities differ in their lifetime. Some concurrent activities are there from the beginning and last as long as the simulation such as the employees. Others come to life triggered by other concurrent activities and have a determined lifetime, such as the transfers, triggered by employees and finished when the transfer is finished.

Java allows the programmer to control the lifetime of a thread accordingly. In this section we will explain all issues of the lifecycle of a Java thread, i.e. its contruction, start, pausing, termination, and the role of priorities. The methods to create threads and to control their lifetime are provided by the Java class Thread and the interface Runnable in the package java.lang. The class Thread provides *non-static* methods, to be invoked on objects of class Thread or of its subclasses, and *static* or *class* methods which are invoked on class Thread.

### Creating a Thread

In the previous section, we have already mentioned that thread objects are created by deriving a new class from class Thread and by creating an instance of the new class. Because Java supports only single inheritance it is not always feasible to inherit from Thread. Another possibility is to specify a new class which implements the interface Runnable. Then an instance of that class is created and it is passed as an argument to a constructor of class Thread.

To provide functionality, in either case the new class has to implement the method run(). The body of the method run() contains the code which will be executed by the thread. In case of inheritance, the new class can override the method run(). In the case of the interface, the new class has to implement run() which is the only method of the interface Runnable. The two possibilities are depicted in Fig. 4.4. For the class Thread we only show method run() from the set of methods because it is the focus here. Other methods are not supposed to be overriden.

In the following, we give an example of a new class derived from Thread. The purpose of the class AccountingTransaction is to transfer an amount of money between a source and target account of class Account. This behaviour is implemented in the run()-method. This method does not have any parameters and its signatur cannot be changed because it is called implicitly, which will be explained later. Apart from specifying run() we therefore have to make sure that the thread is given the data it needs for carrying out a

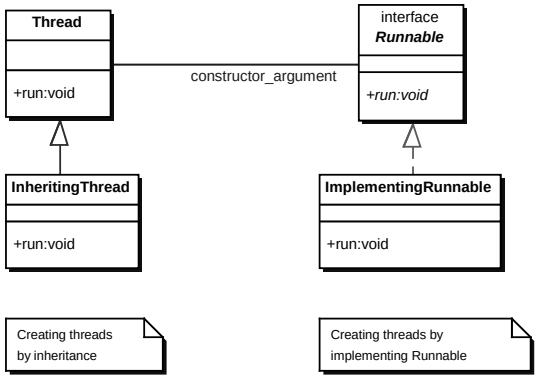


Figure 4.4: Deriving New Threads

transfer. The accounts needed and the amount to be transfered are passed with the constructor and are assigned to fields declared in class AccountingTransaction.

```
class AccountingTransaction extends Thread {
  Account a1;
  Account a2;
  int amount;

  AccountingTransaction(Account a, Account b, int am) {
    a1 = a;
    a2 = b;
    amount = am;
  }

  public void run() {
    a1.transfer(a2,amount);
  )
}
```

Apart from the fields defined in the derived class each thread also has a number of predefined attributes such as *thread name*, *thread group*, *priority* and the *daemon*-property. In so far as they are relevant for concurrency errors, the purpose of these properties will be explained in the sequel. These attributes are only accessible by special methods to inspect and manipulate those attributes, which are declared in class Thread. The thread group controls the access and the lifetime of a thread. A thread's lifetime is bound to its creating thread's lifetime.

### Starting a Thread

After the creation of a thread object, the corresponding thread is started by invoking start() on the thread object. This causes the Java Virtual Machine

to fork a new control flow which executes the thread object's run()-method. Because run() is called implicitly its signature cannot be changed and contains no parameters. A thread is always created and started by another thread, which can also be the main()-thread.

Above, we have specified the class AccountingTransaction for implementing transfers. Now, we want to create thread instances and fork new control flows. We have to identify which thread is responsible for creating and starting them. Transfers are triggered by employees using the terminal. So the code responsible for creating an object of AccountingTransaction is part of the class Terminal. This is an object without its own control flow but it will be used by an employee thread and hence it will be an employee thread which creates an instance of AccountingTransaction. The class Terminal is given in the following. An object of class AccountingTransaction is created in method transferAmount(). Then its priority is set to the maximum. Then start() is called on it. The purpose of priorities will be explained later.

```
class Terminal {
  private Bank bank;
  AccountingTransaction at;

  Terminal(Bank b) {
    bank = b;
  }

  // see Terminal.handOverCheque()
  public void transferAmount(int a1, int a2, int amount) {
    Account A1 = bank.getAccount(a1);
    Account A2 = bank.getAccount(a2);
    at = new AccountingTransaction(A1,A2,amount);
    at.setPriority(Thread.MAX_PRIORITY);
    at.start();
  }
}
```

After calling start until a thread actually runs, and after completion of run until it is cleaned up there is a short elapse of time. During this time, it is not defined, how method calls return. E.g., isAlive() returns true immediately after start was called.

Each thread object can be started only once. Restarting a thread object whose thread is already running will cause an exception. Restarting a thread which was finished has no effect. In a concurrent Java program the number of threads is not limited and not determined at compile time. Every Java program has at least one thread executing the method main() which does not have to be created explicitly.

While a thread is running its purpose is to do calculations, interact with its environment, and interact with other threads, all of which is accomplished by creating and accessing other Java objects. A thread navigating through

the net of objects might need to access its thread object which is typically not known in the object where the thread is currently executing. By calling the static method currentThread() a thread can obtain a reference to its thread object. When the thread object is of a subclass of class Thread the fields of the subclass can only be accessed after casting the object received by currentThread() to the inherited class.

Instead of providing thread instance-specific data via subclassing, the class ThreadLocal allows a thread to create an object with a predefined set() and get() method by which the thread can store and retrieve data independent from where it is currently executing. Each thread has its own copy and is not aware of the other copies for other threads.

### Pausing a Thread

In a concurrent environment, threads will not always have to calculate something but can have explicit idle times before synchronising with other threads or before exchanging data. In the bank example, we want to simulate that employees are not working all the time but can be idle.

Idle time is mapped to Java as follows. With the static method sleep() a thread can pause itself for a specified time. The thread will automatically resume when the time span has elapsed.

| public static void sleep(long ms) throws InterruptedException |
|---|

Table 4.1: Signature of sleep()

Invoking this method can raise an InterruptedException for which an exception handler has to be supplied. The exception is thrown when the interrupt flag of the thread has been set after method interrupt() has been called on the thread but if the thread has not yet polled its interrupt flag. The interrupt mechanism will be explained in detail in a section 4.3.

In our example we can use sleep() in the run()-method of employees (see the next code example) to achieve that they are not active all the time. After they are started they generate data for transfers randomly. Then they are paused before they start the transfers.

The methods suspend() and resume() to pause a thread from another thread are deprecated.

### Prioritising a Thread

While in general the developer should not think about the order in which multiple threads are running, there may be cases where a thread does an

urgent calculation which should be favourised over other threads. Thread priorities are a way to express this.

In section 4.1 we explained that the scheduler is completely responsible for determining a valid execution order. In some cases this is not satisfactory, e.g. if some calculations are more urgent than others. For instance, the transfers from the banking example should be carried out immediately because they need mutual exclusive access to accounts and shouldn't block them for too long.

The Java Virtual Machine supports a simple fixed priority scheduling. Each Java thread is given a numeric *priority* in the range of MIN_PRIORITY defined as 1 and MAX_PRIORITY defined as 10. By default, main() is given the priority 5. Unless set otherwise each thread has the same priority as the one which created it. The priority can be changed dynamically.

The thread with the highest priority is chosen for execution. When that thread stops, or is suspended for some reason, a same-priority or lower-priority thread starts executing. If a thread with a higher priority than the currently executing thread is ready to execute, it is immediately scheduled. This is also called *preemption*. The Java runtime will not preempt the currently running thread for another thread of the same priority [1]. Depending on the underlying system some Java Virtual Machines map the range of priorities to a smaller range with the effect that some threads will not be able to preempt others.

Going back to the example, we will use priorities to ensure that transfers are executed immediately. Transfers are created in the code of class Terminal (see the code example in the previous section. After the creation of a thread object of class AccountingTransaction, setPriority(Thread.MAX_PRIORITY) is called on it.

A thread may, at any time, give up its right to execute by calling the static method yield(). Note that threads can only yield the CPU to other threads of the same priority. In the example we include Thread.yield() after the terminal has started the transfers to ensure that other employees can also issue transfers.

Note the difference between sleep() and yield(). A yielding thread may appear to sleep for a while but this effect is not guaranted because it depends on all other threads of the program.

```
class Employee extends Thread {

  private int amount;
```

[1] The system implementation of threads underlying the Java Thread class may support time-slicing which has the effect that threads of the same priority will effectively preempt each other.

```
  private static int numAccounts;
  private Bank bank;
  private Terminal term;

  Employee(Bank b, Terminal t) {
bank = b;
term = t;
numAccounts = b.getNumAccounts();
  }

  public void run(){
    int i=0;
    while(i<100) {
      //methods to determine randomly parameters for the simulation
      amount = randomizedAmount();
      int a1 = randomizedAccount();
      int a2 = randomizedAccount();

      //simulate idle time
      try {
        Thread.sleep(500);
      }
      catch (InterruptedException ie) {}

      //transfer only between different accounts
      if(a1 != a2) {
      term.transferAmount(a1,a2,amount);
      Thread.yield();
      i++;
      }
  }

}
```

### Terminating a Thread

In the example we decide that a thread for an accounting transaction only carries out one transfer. Then it should be finished. For a new transfer, a new thread will be started[2]. Usually, the control flow of a thread terminates after run() returns. The return is reachable if neither a never ending loop nor an unsolvable lifeness problem occurs. The method IsAlive() called on a thread object allows another thread to test if this thread has been started and has not yet terminated. Also, threads terminate when the thread which created them is terminated. To avoid this, a thread can be given the *daemon*[3] property. The JVM runs as long as there are non-daemon threads. Then it runs as long as the main()-thread or the Java Virtual Machine.

[2]Note that this is a very naive approach. As starting and terminating threads is time consuming, in a real program one would try to reuse a thread to perform accounting transactions one after another.

[3]The origin of the term daemon for background processes is most recently ascribed to a thought experiment by the physician Maxwell in which a hidden daemon selects molecules. The thereby superseded hypothesis says it is an acronym for disk and execution monitor.

Methods to stop a thread were initially part of class Thread but have been deprecated because of potentially dangerous side effects on other running threads which can lead to inconsistent and undefined system states.

A thread can terminate abnormally when an unchecked exception is not handled or a checked exception is re-thrown. This will be discussed in the next section.

### Exceptional Termination

Although the exception mechanism is the same for concurrent Java programs as for a single-threaded Java program its effects are slightly different. While a sequential program is aborted if an exception is not handled, a multithreaded program may continue if the exception does not occur in the main()-thread.

Exceptions are objects that store information about the occurrence of an unusual or failure condition. They are thrown when that error or unusual condition occurs. Java provides a mechanism for handling exceptions also known as *catching* an exception. There are two types of exceptions in Java, *unchecked* exceptions and *checked* exceptions. Exceptions are part of method signatures.

- Unchecked exceptions inherit from class RuntimeException, e.g. the predefined ArithmeticException, NullPointerException, or ArrayIndexOutOfBoundsException. Java does not require that you declare or catch unchecked exceptions in your program code. Unchecked exceptions may be handled as explained for checked exceptions in the following paragraph.

- Checked exceptions do not extend the RuntimeException class. Checked exceptions must be handled by the programmer to avoid a compile-time error, e.g. IOException or InterruptedException.

There are two ways to handle checked exceptions: by *declaring* the exception using a throws-clause or by *catching* the exception. To declare an exception, the keyword throws is added to the method header followed by the class name of the exception. Any method that calls a method that throws a checked exception must also handle the checked exception in one of these two ways. If a checked exception occurs and is not caught before it reaches the main method of the application, the program will crash.

After a method throws an exception, the runtime system leaps into action to find a method to handle the exception in the call stack of the method where the error occurred. The runtime system searches backwards through the call stack, beginning with the method in which the error occurred, until it finds

a method that contains an appropriate exception handler. An exception handler is considered appropriate if the type of the exception thrown is the same as the type of exception handled by the handler. Thus the call stack is searched until an appropriate handler is found and one of the calling methods handles the exception. The exception handler chosen is said to catch the exception. If the runtime system exhaustively searches all of the methods on the call stack without finding an appropriate exception handler, the runtime system (and consequently the Java program) terminates.

When the main thread in a single-threaded application throws an uncaught exception, the stack trace is printed on the standard output, e.g. console, and the program exits. This is also referred to as program crash. In a multithreaded environment not every thread is attached to a console. Therefore, a thread can silently disappear from an application, especially without a (stack) trace, when an exception is thrown and not caught.

## 4.3 Unsynchronised Interaction

Despite the fact that a thread has to be created by another thread we have not addressed interaction between threads so far. Interaction takes place between two threads, directly or via shared objects. Either it is *unsynchronised* meaning that no timely coordination controls the interaction or it is *synchronised* meaning that some timing constraints apply, e.g. mutual exclusion.

In this section we will address unsynchronised interaction between threads, via shared objects and directly.

### Sharing Objects

Threads can use ordinary Java objects to communicate, either by explicitly exchanging data or by using data exchange to encode signals. Shared use of data is the source for inconsistencies in the case of simultaneous read and write accesses. In general, synchronisation has to be applied to avoid inconsistencies. Unnecessary synchronisation should be avoided because it is a bottle neck for a program. Synchronisation is not needed if several threads access the same object by read-only methods as the state of the object is never changed and hence the read-only methods may take place in any order including overlap.

The class Bank creates account objects and can be asked for references to them. For instance, an employee can query the bank for a certain account. Many such queries can happen in any order including overlap. Because they do not change the state there is no constraint on their execution order, i.e.

they do not have to be synchronised. In this somewhat contrived example, the bank interface does not provide methods for changing accounts. The constructor is the only method which could interfere with the other read-only methods. A constructor cannot be synchronised and therefore other mechanisms are needed if threads are assumed to access objects before they are fully constructed.

In the following the code of Bank is given. After creation, the bank can be queried by the two methods for those accounts. With the method getNumAccounts() the total number of accounts can be queried and with the method getAccount(int nr) an individual account can be retrieved to work with it.

Java guarantees atomicity for concurrent access and updates to fields of any type except long and double. Atomicity is thereby also guaranteed for fields serving as references to other objects. These exceptions are saying that the field access to long and double can overlap and thereby a value computed by none of the threads can be result. To avoid these inconsistencies long and double can be made atomic by using the keyword volatile.

```
public class Bank {

  public Hashtable accounts;
  public int numAccounts;
  public int numEmployees;

  Bank(int numEmpl, int numAcc) {
    numEmployees = numEmpl;
    numAccounts = numAcc;
    accounts = new Hashtable();
    employees = new Vector();

    for(int i=0;i<numAccounts;i++) {
      accounts.put(new Integer(i),new Account(i,100));
    }

    for(int j=0;j<numEmployees;j++) {
      Terminal t = new Terminal(this);
      Employee e = new Employee(this,t);
      employees.addElement(e);
    }

    for(int j=0;j<numEmployees;j++) {
      Employee e = (Employee)employees.elementAt(j);
      e.start();
    }

    for(int j=0;j<numEmployees;j++) {
      Employee e = (Employee)employees.elementAt(j);
      e.join();
    }
  }

  public int getNumAccounts() {
```

```
    return numAccounts;
  }

  public Account getAccount(int nr) {
    return (Account)accounts.get(new Integer(nr));
  }
}
```

### Interrupting a Thread

A thread has a built-in interrupt mechanism. Other threads can interrupt a thread by invoking the method interrupt() on the thread object, then internally the interrupt status is set to interrupted. The thread can poll its interrupt status using the static method interrupted(), which returns true or false. The interrupt status of the thread is cleared, i.e. set to not interrupted by this method. Other threads can test, if a thread has been interrupted by invoking the method isInterrupted() on a thread object. The interrupt status of the thread is unaffected by this method. If interrupt() is called on a thread which is in a state where it cannot poll its flag, e.g. sleeping, waiting, or joining, an InterruptedException is raised. While interrupting serves as a means of communication it cannot be considered as a synchronisation. The interrupting thread has no control over the timing when the interrupted thread will handle the interrupt. Also the interrupting thread is not informed when the thread has handled the interrupt. Such an interrupt mechanism can be easily simulated by two programmer-defined methods, one to send a signal to the thread object and one to read out the signal. The advantage of this built-in interrupt is that when the interrupted thread is in a state where it cannot poll the signal it is interrupted by an exception.

| public void interrupt() |
|---|
| public static boolean interrupted() |
| public boolean isInterrupted() |

Table 4.2: Signature of Interrupt Methods

Note that any call to wait(), sleep(), join() has to be encapsulated in a try-catch-block. Each such call will return when an interrupted happens, i.e. each such call can potentially consume an interrupt.

## 4.4 Synchronised Interaction

In this section we look at the synchronisation mechanisms provided by Java. As motivated earlier, concurrent threads need to coordinate their activities

with respect to time. This may or may not include the exchange of data. Note that also start() is a means of a timely coordination between the thread starting a new thread and the new thread being started. In the following we look at further language means.

### Joining

Forking a new thread by calling start() is the simplest means of synchronisation between threads. Similarly, a thread can wait for another thread to finish before it does something. In the banking simulation, the bank is responsible for forking the new employee threads. The bank can finish when all employee threads are finished.

This can be expressed by calling e.join() on each employee object (see the code above for class Bank). The non-static method join() suspends the calling thread to wait until the target thread, on which the method is called, terminates. Method join() throws an InterruptedException if the joining thread is being interrupted by another thread. There exists a variant with a timer. The time in ms can be supplied as a parameter.

| public void join() throws InterruptedException |
|---|
| public void join(long ms) |

Table 4.3: Signature of join()

A thread can only join one other thread at a time. The joining has no effect on the joined thread. It can join any thread provided it has a reference to its thread object. If it needs to join more than one thread it has to do it one after another. In the case that the target thread terminates with an exception, the joining thread returns without an exception.

A thread can join a thread which is already terminated. A thread can join a thread which is not yet started. In both cases the joining thread will return immediately.

### Mutual Exclusion

In a multithreaded environment threads can simultaneously access objects. This can put objects in an inconsistent state also referred to as *data inconsistency*. To avoid inconsistencies the access on these objects must be mutual exclusive. Only if it is ensured that they are not accessed by more than one thread or read-only by all except one thread no mutual exclusion is needed.

In our example the accounts are shared by the threads carrying out the transfers and so access to them needs to be mutual exclusive. We have

illustrated in Fig. 4.3 that concurrent transfers can cause inconsistency in account data. Here we will explain how this kind of problem can be avoided.

Java supports mutual exclusion through *monitors*. A monitor encapsulates data and provides *mutual exclusive* access to observe or modify the data. In addition, a monitor supports *conditional synchronisation*. This permits a monitor to block threads until a particular condition holds, e.g. a condition on its data. In this subsection we deal with mutual exclusion. In the next subsection we deal with conditional synchronisation.

Any Java object can serve as a monitor. Concurrent activations of the same or of different methods on the same object are made mutually exclusive by prefixing methods with the keyword synchronized. A thread executing a synchronized-method excludes other threads from executing any synchronized-methods on the same object. The keyword synchronized grants exclusive access to the entire object, however non-synchronized methods can always be executed.

| |
|---|
| synchronized method_name (...) |
| static synchronized method_name (...) |
| synchronized(o){...} |
| synchronized(this.getClass())(...) |

Table 4.4: Keyword synchronized

Also a class can serve as a monitor if the synchronized keyword is attached to static methods. The class lock has no relation ship to the object locks.

More fine-grained exclusion is achieved by declaring a block of statements within a method as synchronized. Because synchronisation in Java is always associated with a monitor, in the same ways as a synchronized-method, a monitor object has to be supplied when specifying a synchronized-block. This can be the object on which the method which contains the synchronized-block has been called. Then the object specified for the synchronized-block is this. It can also be any other object known in the context of that method. The exclusion rules for methods extends to blocks. We will use the term synchronized-region to refer to both, synchronized-blocks and synchronized-methods.

In the following example of class Account from our banking example, all methods are made mutually exclusive using synchronized. This guarantees mutual exclusive access to accounts in order to keep the amount consistent in a multithreaded context.

```
public class Account {
  private long value;
```

```
  private int nr;

  Account(int n, long v) {
        nr = n;
        value = v;
  }
  public synchronized int getNumber() {
        return nr;
  }
  public synchronized long getValue() {
        return value;
  }
  public synchronized void setValue(long v) {
        value = v;
  }
  public synchronized void withdraw(long amount) {
        value-=amount;
  }
  public synchronized void transfer(Account other, long amount) {
        other.withdraw(amount);
        value+=amount;
  }
}
```

To describe the operational meaning of synchronized, we draw on the metaphor of a *lock* for each object. To be able to enter a synchronized-block or -method a thread needs to obtain a lock on the object. If multiple threads are trying to obtain such a lock, only one thread is assigned the lock. Other threads are locked out and they are blocked, i.e. they cannot proceed. If a thread leaves a synchronized-region, it releases the lock. By nondeterministic choice one of the blocked threads will receive the lock and can start to execute. All synchronized non-static methods inherited and defined for a class use the same lock.

The following is important to know when using synchronized:

- Java locking is nested and re-entrant, a thread can obtain more than one lock and it can re-enter a lock it holds. Re-entrance allows a synchronized method to make a self-call to another synchronized method on the same object without freezing up.

- The keyword synchronized is not automatically inherited when subclasses override superclass methods. A synchronized-method can be overriden by a non-synchronized-method.

- Methods in interfaces cannot be declared as synchronized.

- The object lock does not imply mutual exclusive access to static fields of the corresponding class. Locking of static fields of a class is achieved by synchronized static but only for the class not for its super classes.

- A constructor cannot be synchronized.
- Method run() should never be synchronized.

The monitor approach to mutual exclusion is characterised by the fact that the locking of mutual exclusive objects is not an independent operation. It is always linked to another activity such as executing a method or executing a block of statements. That holds for the beginning of the mutual exclusion as well for its end. When leaving a synchronized-block or -method the mutual exclusion implicitly ends. This avoids errors compared to a style where one would have to specify the beginning of the mutual exclusion and the end separately and without possibility of syntactic checking. The fact that synchronized is always used with a block or method is important for analysing concurrency failures involving mutual exclusion. It is important to determine the block or method which issued the mutual exclusion.

The exception mechanism of the Java platform is integrated with its synchronisation model, so that locks are released as synchronized statements and invocations of synchronized methods complete upruptly. If a thread holding a synchronized-lock encounters a runtime exception and terminates it will release its locks. This ensure that other threads can use these locks but when the locks are released the consistency of the monitor data is not guaranteed.

*Nondeterminism* also plays a role in the presence of mutual exclusion. With mutual exclusion only one of several threads is granted access. However, the order in which contending threads are granted mutual exclusive access is nondeterministic. The programmer must assume that any order can happen.

### Conditional Synchronisation

Sometimes it is desirable to build more complex synchronisation than mutual exclusion. This is the case when a concurrent activity needs a certain condition of the environment to be fulfilled to continue its activities. For the banking simulation we might think of an extension where transfers are only carried out if there is enough money to withdraw the amount. Otherwise, they should not be aborted but be delayed. This results in a number of activities waiting for their completion and we are in charge of finding a way how they will be informed when the condition they are waiting for is fulfilled. Suspending and resuming threads based on the evaluation of a condition is called *conditional synchronisation*. Together with mutual exclusion, it makes up the monitor concept.

Ideally, when mapping to a programming language, as many features of conditional synchronisation as possible should be automated, i.e. keeping a list of waiting threads, watching for the change of some condition, and

finding out which threads need to be notified about these changes. Java does provide some support for conditional synchronisation as part of its monitor concept but the programmer is asked to supply some information such as when a thread needs to wait and also when it needs to be woken up.

The implementation of *conditional synchronisation* is not as straightforward as mutual exclusion. The blocking on conditions and the signalling of changed conditions have to be implemented by the programmer. Therefore, a simple mechanism is supplied with each Java object. Each Java object has a *wait-queue* for queuing threads which have to be blocked on a condition. With the methods wait(), notify(), and notifyAll() of class java.lang.object the wait-queue of the object can be manipulated. These methods can only be used within a synchronized-region. Note that these methods are non-static. These methods are sometimes referred to as monitor-methods.

| public final void wait() throws InterruptedException |
|---|
| public final void wait(long ms) throws InterruptedException |
| public final void notify() |
| public final void notifyAll() |

Table 4.5: Signature of wait() and notify/All()

- When a thread calls wait() it changes its state from running to waiting. The thread is inserted into the wait-queue and releases its lock. The queue is in fact only a set, because Java does not guarantee anything about the order in which the threads are enqueued or dequeued.

- Threads can only be removed from this queue by notifications on the same object, which are issued by another thread calling notify() or notifyAll(). Using notify() removes only one thread from the queue. No assumptions can be made, which thread will be removed. A call to notifyAll() removes all threads from the queue. If notifications are issued and no thread is waiting, the notifications are not kept for threads, which are calling wait() later on. After being removed from the queue, the thread returns from the wait() and it has to re-obtain the synchronized-lock before it can proceed. Releasing the synchronized-lock during wait() and re-obtaining the lock afterwards is both with respect to reentrance, i.e. the lock is fully released and re-obtained as many times as it was accessed before.

Similar to the states sleeping and joining, also in the state waiting, the call of interrupt() causes an exception. Calls to wait() therefore have to be

encapsulated in a try-catch block to handle the InterruptedException defined in the signature of wait(). As explained earlier, the interrupt mechanism will raise an exception if a thread has been interrupted, its interrupt flag has not yet been cleared, and it is in an inactive state such as waiting or sleeping. After having called wait() the thread is in such a state. Then an InterruptedException is thrown in two cases. Either the interrupt has been issued before the thread is calling wait. Then the InterruptedException is thrown immediately. Or the interrupt is issued while the thread is waiting. Then the thread is leaving the waiting state because the InterruptedException is thrown. It is common practice to provide an empty exception handler for the InterruptedException. When the thread returns from wait because it was interrupted, the thread is removed from the wait-queue, and, before the exception is thrown, the thread has to re-obtain the synchronized-lock. If an interrupt and a notification happen almost simultaneously, the thread may leave the wait() either way.

The purpose of wait() and notify(), notifyAll() is to be used for conditional synchronisation in the following way.

1. A thread which has obtained the synchronized-lock tests if the condition necessary for its activities holds.

2. If the condition does not hold, the thread is calling wait(), which puts him in the wait-queue.

3. When it is woken up by another thread through notify(), or notifyAll() it first has to re-obtain all needed synchronized-lock.

4. Then it re-tests the condition before anything else happens. This is implemented by a while-loop.

5. If the condition holds, the thread performs its activities.

6. After this, other threads can be notified.

7. It is important that wait() methods be balanced by notification methods.

The example is mapped to Java as follows. We need to program an account object which blocks threads trying to withdraw money if the result would be beyond a limit. Withdrawing threads are conditionally blocked. Transfering threads always increase the amount and therefore signal to waiting threads that the condition has changed.

```
public class LimitedAccount extends Account {
  private const long limit = 10000;

  Account(int n, long v) {
        super Account(n, v); //???
  }
  public synchronized void withdraw(long amount) {
        while (value - amount <= limit)
            try {wait()} catch (InterruptedException ie) {};
Account.withdraw(amount);
  }
  public synchronized void transfer(Account other, long amount) {
        Account.transfer(other, amount);
        notifyAll();
  }
}
```

Another solution for the notification would have been to call notify() in transfer() to wake up only one thread. Then this thread would have to invoke notify() to wake up the next thread because it could be the case, that the limit is not yet reached. This solution only wakes up one thread at a time to avoid unnecessary wake ups. It is still unfair concerning new threads contending for the lock.

A call to notify() or notifyAll() can only wake up threads which are currently in the wait-queue. If there is no thread in the queue the notify does not have an effect. Also, if after an unused notify a thread enters the wait-queue it is not woken up. That is a notify can not be safed for later use.

Note that compared to the blocking involved in mutual exclusion, here the waiting thread is not automatically woken up. Instead, the waiting thread is dependent on the collaboration of other threads to be woken up. It cannot be checked by a compiler if a corresponding notify() is properly encoded because the notification does not have to be in the same method. It only has to be in the code of the same class or in inherited code. Any notification could become effective at runtime.

Note that the semantics of notify()/notifyAll() in Java differ from the semantics of the analogous functions in a classical monitor by Brinch-Hansen et al. [Lea97]. While the classical definition implies an explicit programming of the wait-queues with more than one wait-queue possible (for several conditions) the Java definition uses a monitor for an entity possessing a lock and a single wait-queue. Also, Java has adopted a so-called *signal-and-continue* semantics which says that the notifying thread continues after the notify while the notified thread has to re-test the condition. The classical *signal-and-urgent-wait* semantics implies that the notifying thread is suspended and that the notified thread can continue without checking the condition. When the notified thread has left the monitor the notifying thread regains the monitor before any other thread. The semantics adopted by Java fit better with

| Concept | Java Syntax | Section |
|---|---|---|
| *Lifecycle control* - Creation | Thread, Runnable | 4.2 |
| *Lifecycle control* - Starting | start(), run() | 4.2 |
| *Lifecycle control* - Termination | return from run, exceptional termination | 4.2 |
| *Pausing* | sleep() | 4.2 |
| *Priorities* | setPriority() | 4.2 |
| *Unsynchronised Interaction* - Sharing Data | shared objects | 4.3 |
| *Unsynchronised Interaction* - Direct Interaction | interrupt() | 4.3 |
| *Synchronised Interaction* - Joining | join() | 4.3 |
| *Synchronised Interaction* - Mutual Exclusion | synchronized | 4.4 |
| *Synchronised Interaction* - Conditional Synchronisation | wait(), notify() | 4.4 |

Table 4.6: Java Concurrency Concepts

the approach to use only one wait-queue.

Nondeterminism extends to conditional synchronisation of monitors. The order in which threads are sent into the queue and are woken up from the wait-queue is not predictable. Newly arriving withdraw()-calls can contend with withdraw()-calls from threads waken up from the queue. Both kind of threads contend for the synchronized-lock. The programmer cannot assume that they are woken up in the same order in which they have entered the wait-queue. If this is not desirable the program has to implement its own synchronisation protocol accordingly.

### Advances in Java Concurrent Programming

As already noted, the Java concurrency features are sometimes regarded as insufficiently designed [Hol00, Lea03]. On the one hand, this has had the effect that libraries have been proposed which provide more advanced synchronisation elements such as locks [Lea03]. On the other hand this has lead to proposals of new language features some of which follow the ideas of aspect-oriented programming [KLM+97] in order to separate synchronisation code from the remaining Java code such as in [MW99].

## 4.5 Summary

In this section we have described multithreading in Java. Our presentation was organised according to the concepts provided by Java. Table 4.6 summarises the concepts we have introduced and the related Java language elements.

The next chapter will build on these concepts to describe, which concurrent liveness failures are possible in Java. In the next chapter, Table 4.6 will be extended with the failures associated with each concept.

# Chapter 5

# Liveness Failures and Potentials in Java

The previous chapter laid the ground for the description of concurrent Java liveness failures by introducing synchronised thread interaction in Java. It also pointed out that concurrency failures are inherent in thread interaction. In this chapter, we will describe this relationship in depth.

The term failure is related to the terminology for error. The term error is widely used and has different connotations. In order to define precisely our notion of error we introduce a standard *terminology for errors* which relates an error to different activities and artifacts during the software development, namely, coding, program, and program execution.

Using this terminology, we will informally introduce liveness *failures* in concurrent Java programs. In the next chapter, these failures will be defined more formally.

In the presence of nondeterminism not only failures but also *potentials* for failures are of interest because they can equally point to problems in the program. We will therefore introduce the notion of a failure potential.

We will introduce the notion of liveness failure using the existing distinction between *liveness* and *safety* properties in concurrent programs. We will elaborate a comprehensive list of conceivable liveness failures. We will focus on the concurrency related failures but not on failures which occur also in sequential programming. For each failure we will discuss the symptoms, the failure state, and the fault in the code.

## 5.1 Terminology

In this section, we will present our terminology for describing concurrent liveness failures in Java. It is based on standard terminology for errors and on existing classifications for errors in concurrent programs.

### 5.1.1 Error, Mistake, Fault, and Failure

The IEEE Standard Glossary of Software Engineering Terminology [I3E90] is a canonical source for software engineering terms and defines the notion of error and related terms, given in definition 5.1. This definition havs not changed since the 1990 version which is a revised version of 1983. Although not stated explicitly in the definition or its context, the term error and its refinements are related to software.

**Definition 5.1 (Error)** *(1) The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition. For example, a difference of 30 meters between a computed result and the correct result.*

*(2) An incorrect step, process, or data definition. For example, an incorrect instruction in a computer program.*

*(3) An incorrect result. For example, a computed result of 12 when the correct result is 10.*

*(4) A human action that produces an incorrect result. For example, an incorrect action on the part of the programmer or operator.*

*Note: While all four definitions are commonly used, one distinction assigns definition 1 to the word "error", definition 2 to the word "fault", definition 3 to the word "failure", and definition 4 to the word "mistake".*

The key essence of an error is that it is dependent on an assumption of what has been expected as correct instead. This is stated explicitly in definition (1) and (3) but it also holds for (2) and (4). Such an assumption can be a specification but also non-externalised knowledge of a human being. For instance, in the case of testing, a so-called test oracle can provide the expected results.

Definition (1) focuses on error as an error measure, i.e. the difference or the deviation between expected and computed result. It is closely related to definition (3), also termed failure, which applies to an incorrect value, result, or state reached.

Definition (2), also termed fault, is used for incorrect instructions in a program. This is also the interpretation of fault used by [FR01] and we will

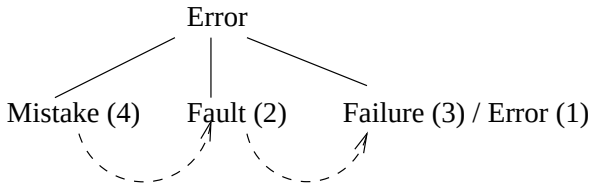


Figure 5.1: Error Terminology

adopt it in the following. Thereby we differ, e.g. from [GJM91] who use the term fault for all incorrect states during the execution of a program.

Definition (3), also termed failure, focuses on incorrect results. We think that the notion of result needs to be broadened. One reason is that programs are not always intended to come to a defined end or to compute output results. Therefore, the term failure should apply to any incorrect execution states of a program (including output). An incorrect state is a state which violates a specification. The error terminology is visualised in Fig. 5.1 The numbers in brackets refer to the numbers used in Definition 5.1.

The definitions (1) to (4) are related as follows. The product of a human *mistake* can materialise in a *fault*, e.g. if the programmer's wrong action is inserting the wrong statement in a specific line of code. Evidence of a fault in the code can come through *failures* at runtime, typically incorrect output values, unexpected program termination, non-terminating or non-progressing execution. The *error* can possibly be determined as a difference between the failure result and the expected result. The above cause-effect relations are depicted by arrows in Fig. 5.1.

It is often the case that the root cause of a failure can be traced to a small but not necessarily coherent area of the program which contains the fault. Sometimes program failures are indications of global problems such as mistaken assumptions or inappropriate architectural decisions. In such cases, it is misleading to assume that editing a small area of the program will prove sufficient to correct a fault [FR01].

Note that this error terminology is independent of a particular programming paradigm but it will be helpful in the following to describe concurrency problems.

### Example

We illustrate the use of the introduced terminology with the example of the deadlock. A deadlock is a program failure because it violates the desired specification that threads should be able to finish their computations. An

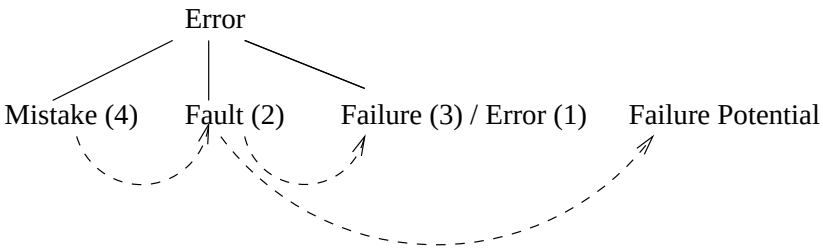


Figure 5.2: Extending the Basic Terminology with Potential

example for a deadlock has been given in Fig. 2.2. The reason for this deadlock is a program fault, i.e. pieces of code which do not work together as intended. Interestingly, there is no corresponding name for the fault related to the deadlock. The code responsible for the example deadlock is the method synchronized transfer(Account other, long amount).

```
public class Account {

  //other members ...
  public synchronized void transfer(Account other, long amount) {
        other.withdraw(amount);
        value+=amount;
  }
}
```

The above method will always be called on the target account of a transfer and will take the source account as an argument. Thus the order in which accounts are locked is determined by the actual transfer only. This allows for cyclic dependencies during locking of the involved account objects. This fault in the code is a product of a human thought process. The programmer made a mistake when not constraining the order of synchronized access to account objects.

### 5.1.2 Potential for Failure

We want to go beyond the IEEE classification by introducing the notion of failure potential. Because of the nondeterminism in concurrent programs a failure might not occur in every program execution, although the program contains a fault. Nevertheless, in some cases, a failure-free execution can contain information about the *potential* for failure. The detection of a potential for a failure is nearly as valuable as a detected failure because it points to the same fault.

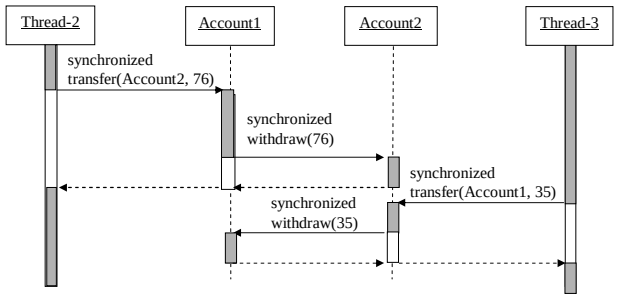


Figure 5.3: Potential for Deadlock

The relation of the potential for a failure to the other terms introduced is depicted in 5.2. In analogy to a failure, a potential failure has its origin in a fault but it is not considered a failure, i.e. failures and failure potentials represent different program executions. This will become clearer with the following example.

### Example

In the motivation, we have contrasted an execution leading into a deadlock (see Fig. 2.2) with a correct execution alternative (see Fig. 2.1). We repeat the latter figure for convenience in Fig. 5.3. The two concurrent threads lock the same two resources, Account1 and Account2, without interfering. We can observe that in this execution the two shared objects are locked in inverted order, i.e. in opposite order. Should the access to the two accounts overlap in time, a deadlock happens. Thus the inverted order serves as a *warning* for a deadlock and is considered a potential for a deadlock (see Fig. 5.3).

### Deriving Potentials from Failures

In the remainder we discuss how potentials for failures can be characterised.

Characteristics for the potential of a failure can be derived from the characteristics of a failure. A failure is detected when all of its characteristical conditions hold. There can be failure-free executions where some of the conditions may hold. Depending on which conditions hold one can take this as an indication that the concrete program bears the potential for a certain failure. Not all conditions are suitable to serve in the definition of a potential. Failure conditions can be classified into those which are fulfilled a priori by the language or the system, those which are fulfilled by most programs, or most program states or executions, and those which hold only for specific program states or executions. If there is a set of conditions which hold only

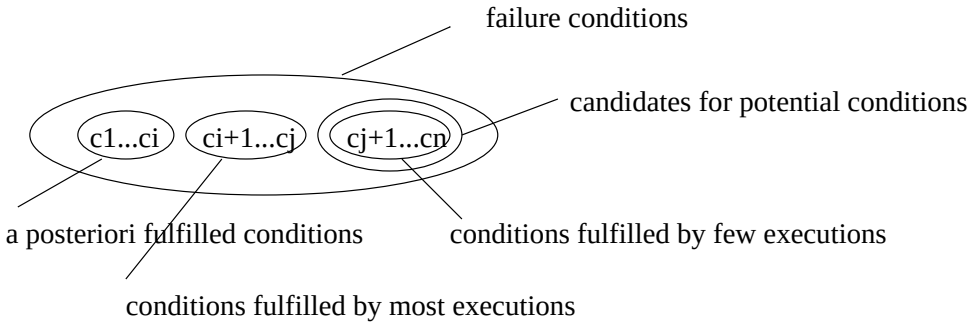


Figure 5.4: Deriving Potential Conditions from Failure Conditions

in specific states or executions, a subset or a set of weaker conditions may serve as candidate conditions to describe a potential. Also, this subset or weaker set should allow to point to a fault in the program. The idea of candidate conditions for potential failure among failure conditions is depicted in Fig. 5.4. Technically, candidates are defined as follows.

**Definition 5.2 (Failure Potential Candidate)** *Given the conditions of a specific type of a failure we can derive candidate conditions for the potential of this failure. From the conditions of the failure, a subset of conditions has to be chosen such that the conditions in the subset are not fulfilled a priori by the language or the system, i.e. they are not fulfilled by each and every program execution.*

*From this set of conditions either subsets can serve as potential conditions or sets of conditions weaker than the original conditions or a mix of both. The set mustn't be empty and weaker conditions mustn't evaluate to true for all cases.*

While this is a very technical view, in reality, not all possible candidates are useful. Some candidates may identify too many executions as failures. The choice of such conditions depends on the concrete failure and will be illustrated by an example in the following. Of course, it can be the case that it is impossible to find such conditions.

Definition 5.2 explains, why failures and potentials are disjunct, i.e. why a potential is not a failure. This is because, it fulfils less conditions than a failure. This definition will be used in later section to establish corresponding potentials from Java liveness failures.

### 5.1.3 Failure and Symptom

A failure will only surface and get noticed trough its symptoms, for instance incorrect or missing output. The term *symptom* depends on a human perspective, i.e. how much the programmer knows about what is supposed to happen and how much the programmer is aware of what is actually happening.

In the easiest case, the symptoms directly refer to the failure state. In other, more common cases, the programmer will make an observation, e.g. performance loss, but does not know what the failure is. The mapping of symptoms to the failures causing them is not one-to-one in concurrent programming and success in identifying failures correctly depends on a lot of experience. Surprisingly, this experience is rarely documented. It is especially rare for concurrent programming, where failures are particularly difficult to track down. Often, authors do not discuss symptoms but prefer to provide lists of typical faults, e.g. for Java [Eng99]. Even books on debugging do not teach the heuristics but present tools [ZK00]. A classification of symptoms is rare as is a classification of symptom-failure relations. An example is [Lew03a], who classifies symptom-failure relations according to their traceability, i.e. whether the symptoms can be traced back to failures or not by a direct cause-effect chain. This classification is independent from concurrency errors, i.e. it is applicable to all kind of errors.

There are symptoms which are characterised by the *presence* of a wrong result or state. This gives a concrete handle to a problem, from where one can try to trace back the cause of the problem. Whether the handle can be successfully used or not, depends on the right tools. When using a classical debugger, there is no means for going back arbitrarily.

There are symptoms which are characterised by the *absence* of a state or result, instead, the missing of the result or the state is the problem. Then, there is no handle from which to start. Also, the first variant might be reduced to the second variant if the wrong result is eventually caused by the lack of some behaviour.

Liveness failures tend to be part of the second category, absence, and are therefore difficult to track in the way described above. Although we take a different approach in the following to classifying liveness failures, our ideas were strongly influenced by [Lew03a] and an memorable presentation of [Lew03b].

## 5.2 Liveness Failures

In the remainder of this chapter we will informally introduce liveness failures in Java. In order to state precisely what we understand by liveness failures we will introduce the classical distinction between liveness and safety failures. We will shortly introduce race conditions, a notion related to concurrency failures.

### 5.2.1 Liveness and Safety

In this section, we will introduce the main classification for failures in concurrent programs. As we noted in the definition of an failure, one can only speak of failures if one can compare against a specification and if that specification is not met. To be able to classify concurrency failures we introduce two desirable properties of a program: *liveness* and *safety* [Lam77, OL82]. A property is an attribute of a program that is true for every possible execution of that program.

**Definition 5.3 (Liveness)** *The liveness property means that a program eventually reaches a good state. In sequential programming, reaching a good state refers to termination which is not always applicable as modern sequential programs can run for a long time or forever, e.g. a server process or an embedded program. In a concurrent program, it means that something eventually happens in an activity, i.e. each activity is eventually making progress until they come to a defined end if any, e.g. activities are eventually granted the resources they are waiting for.*

**Definition 5.4 (Safety)** *The safety property describes that a program does never reach a bad state. More formally, the property says that all object are in a consistent state as specified, e.g. by invariants. Concurrent safety means that data is never corrupted by contending threads.*

Both properties can be violated by sequential programs (inconsistent data, non-terminating execution) as well as by concurrent programs. In both cases, the violation can be avoided by the programmer. The violation will be observable only while the program is executing. Therefore, we speak of safety and liveness *failures* when these properties are violated. Following our general error terminology, it is possible to speak of potentials for liveness failures and of potentials for safety failures but the potentials of a failure themselves do never violate these properties. We depict the integration of safety and liveness failures into the basic terminology of errors in Fig. 5.5.

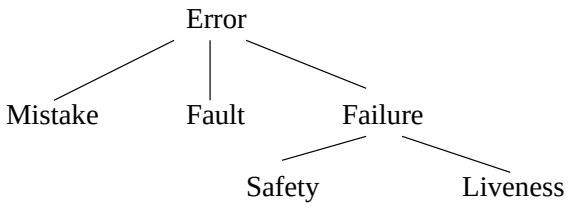


Figure 5.5: Safety and Liveness Terminology

An example for the violation of a liveness property is the failure of a deadlock. Note that a deadlock is sometimes classified as a safety violation because it is considered a bad state [MK99]. We prefer to view it as a liveness failure because that stresses what effect the failure has, namely, that threads involved in the deadlock fail to make progress, and where the reasons for this failure lie and that this failure has much in common with other liveness failures.

The example of the the deadlock also illustrates that liveness failures occur in legal Java programs, i.e. those programs can be compiled and executed. Either some or all executions contain the deadlock although the programmer did not intend this. But the way how the programmer used Java for writing a specific program allowed this to happen. Therefore, it is also said, that liveness failures are *inherent* in Java, i.e. they have their origin in the programming language itself. The same holds for safety failures.

### Race Condition

While liveness problems are due to the various mechanisms for thread synchronisation, at the core of safety problems is the unsynchronised access of data across different threads. As this can happen all the time and every where in a concurrent program, one tries to tackle this problem using formal methods. The potential for a safety problems in concurrent programs is identified by the notion of a race condition.

An unexpected or unintended execution order leading to wrong results based on *unsynchronised* access to variables is also termed race condition. A *race condition*, also termed a *data race*, is given, if two threads are accessing the same shared variable or memory location in an unsynchronised way and at least one thread modifies the variable. Thereby, a data race is the source for a safety violation. A race condition is anomalous behaviour caused by the unexpected dependency on the relative timing of events. In other words, a programmer incorrectly assumed that a particular action would always

happen before another. The following definition for a data race in Java is taken from [CLL+02].

**Definition 5.5 (Data race)** *A data race is defined as two memory accesses which satisfy the following four conditions [CLL+02]:*

*(1) the two accesses are to the same memory location (i.e. the same field in the same object) and at least one of the accesses is a write operation;*

*(2) the two accesses are executed by different threads;*

*(3) the two accesses are not guarded by a common synchronisation object (lock); and*

*(4) there is no execution ordering enforced between the two accesses, for example by thread start or join operations.*

Data races are not the only source of nondeterminism. Races are also involved in competing for access to a synchronised, i.e. mutually exclusive, object. Depending on the order in which threads are guaranteed access the subsequent behaviour of a program might change.

Safety problems can be analysed by analysing race conditions. Analysis of race conditions is a hard problem. Therefore, it is the main goal of research in the area of concurrent safety to improve algorithms for race conditions. Hence, safety failure detection is quite well supported.

The situation for liveness failures is different, because the general conditions for these failures have not yet been established for Java. Typically, only the deadlock is well supported. Therefore, we will focus on liveness failures in the following.

### 5.2.2 Concurrent Java Liveness Failures

In the following, we will informally present a list of concurrent liveness failures in Java. Neither literature nor tools cover all concurrent Java liveness failures. The original source for Java does not provide such material, neither in their books [GJS97] nor on the web. We have consulted a range of books on Java concurrent programming [Lea97, Lea00, Mit00, DD00], online material [Hol98], and also the state-of-the-art tools JBuilder, JDeveloper, JProbe, and Jinsight [JBd01, JDv01, Jin, JPr00]. We found out that the literature about Java in generally does not address concurrency failures in great detail nor do the tools provide support for failures other than the deadlock and deadlock potential.

Here, we will draw on these references to give a comprehensive presentation of concurrent liveness failures. We complete the ones found in literature by those we found missing. Our approach to determine additional failures is based on the idea that the number of synchronisation primitives in Java is

limited and one can informally check, which combinations of primitives are feasible or produce liveness failures. While this is carried out in this chapter on an intuitive level, we will later show how this can be done more formally by testing combinations of thread states of our desired formalisation.

We will use UML sequence diagrams to illustrate these failures. For each failure we will describe the related fault. The symptoms of liveness failure are always the same, the threads involved fail to perform intended computations.

### F1 Deadlock

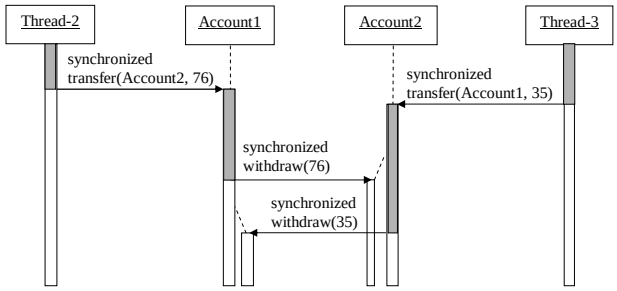


Figure 5.6: Deadlock

The deadlock has been already extensively discussed in Chapter 2. Here we give its precise definition.

**Definition 5.6 (Deadlock)** *A deadlock is a state in a program where two or more threads or processes are mutually blocking each other and can never make progress. A deadlock is described with three necessary conditions (1.-3.) and one sufficient condition (4.) [CES71].*

1. *Serially reusable resources, e.g. resources are mutual exclusive.*
2. *Incremental acquisition of resources.*
3. *No preemption of resources.*
4. *Circular chain of processes such that each process holds a resource which its successor in the cycle is waiting to acquire.*

This situation cannot be resolved by the threads involved. A deadlock with two threads has also been called a *deadly embrace*.

Definition 5.6 is language and system independent. It is also applicable to Java. Java is deadlock-prone because it fulfills the three necessary conditions. Any resource in general is reusable. A resource is made serially usable

through the keyword synchronized. Synchronised resources can be acquired incrementally in Java. The forth condition can be fulfilled by a certain execution. For instance, the sequence diagram in Fig. 5.6 describes an execution that fulfils the four above conditions and hence describes a deadlock.

The fault leading into the deadlock has also been addressed already. It is source code which does not impose order on locking when more than one lock is shared between different threads.

### F2 Wait-induced Deadlock

**Even when ordering on locks is imposed, the use of wait() may change the ordering as in the following example. This may lead into a wait-induced deadlock [Hol98]. The deadlock itself fulfills definition 5.6.**

As an example look at the following code fragments:

```
//Thread 1:

  synchronized(A)
  {   synchronized(B)
      {   //...
          A.wait();
          //...
      }
  }

//Thread 2:

  synchronized(A)
  {   synchronized(B)
      {   //...
      }
  }
```

Based on the above code, a wait-induced deadlock can occur in the following way:

- Thread 1 acquires both locks, enters the wait() and releases A as a side effect of waiting.
- Thread 2 activates, acquires A, but can't get B because Thread 1 has already locked it.
- Thread 1 is notified, through wait() it tries to re-acquire A but can't get it because Thread 2 has already locked it.

The execution is deadlocked. The main issue is that wait() hides the fact that the order-of acquisition isn't the same on both threads. Thread 1 holds the lock on B, but releases the lock on A (when it starts to wait). It re-acquires the lock on **A** when wait returns, but the order of acquisition is effectively B, A. Thread 2, however, acquires the two locks in the order A, B. Note that this is not a new kind of failure but a different kind of fault which produced it.

### F3 Indefinite Blocking

**A thread is blocked having called synchronized but is never granted the lock.**

This can even be the case for more threads regarding the same lock. Here, the threads are however not involved in circular dependencies as in the deadlock. The situation is, that another thread holds the lock but does not intend to release it. Note that it cannot be decided in general if a lock will ever be released by thread. This situation is also called lock *starvation*.

By indefinite blocking we do not refer to a similar situation where a thread having called wait is also blocked waiting for a notification signal which will never be sent.

### F4 Missed Notification

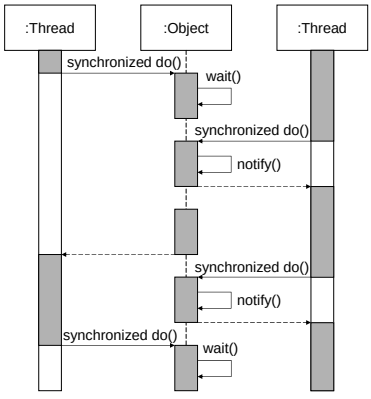


Figure 5.7: Missed Notification

**A thread cannot be removed from the wait-queue, because it entered the wait-queue *after* the corresponding notify()/notifyAll()-call was issued by another thread [Lea00].**

While the deadlock only involves locking, i.e. the use of synchronized, there is a range of failures which are based on conditional synchronisation, which is based on the methods wait(), notify() and notifyAll(). A thread tests a condition and, if it doesn't hold, issues a wait(). The thread assumes that there are other threads which issue a notify() or notifyAll() when the condition has changed. Moreover, one can assume a programming style which uses notifications in an efficient way, i.e. notifyAll() is not used if only one thread should be removed from the queue and if each removed thread will trigger the removal of the next waiting thread.

A sensible use of these mechanisms is to balance wait()- and notify()-calls at runtime. A wait() in one thread is at some time in the execution followed by a notify()/notifyAll() in another thread. Several wait()-calls from different threads are followed by several notify()-calls from one or more other threads or by a notifyAll() from another thread. We depict this principle in Fig. 5.8. If this equilibrium becomes unbalanced, a liveness problem is threatening by threads not being properly woken up.

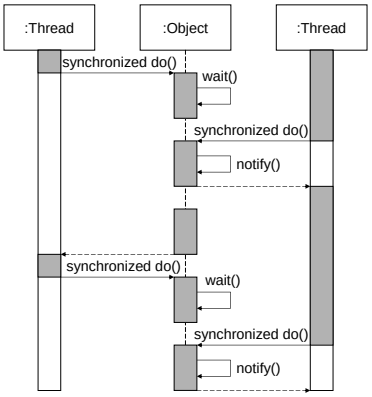


Figure 5.8: Balancing wait() and notify()

Note that this failure describes a state in a program, namely, a waiting thread. But to identify this as a failure, one has to look at the execution history. An example is given in Fig. 5.7. In the upper half of the sequence diagram, the wait()-call within the left thread is balanced by a notify() from the right thread. In the lower half, the notify() takes place before the wait() and is therefore missed from whithin the left thread. Also, no timed wait() is used, i.e. the threads are not removed from the queue simply by time out. The situation could also be extended to the use of notifyAll().

The fault behind this failure is that the programmer has made the (wrong) assumption that the notify() always takes place after the wait() and that the

programmer has not included any synchronisation to make sure that each notify() takes place after a wait().

Instead of missing a notification, the thread could have also missed an interrupt(). This points to the fact that the above heuristic deals only with one solution to wake up the thread, although also the other solution is possible. Also, if timed wait() is used, we must consider that threads are woken up simply by timeout.

The balancing is only a heuristic based on the program's history. However, one could also assume that such a notification is still to come no matter what the history looks like. It is not obvious how long a thread is supposed to be waiting. A timer could be used to trigger the detection.

Note that the situation described here differs from the idea of a potential. While a potential is never a failure itself, the above situation can be a failure following a heuristic although it might be a *false positive*.

#### F5 Missing Notification

**The execution history of the program is such that there has never been a notification for the object where the thread is waiting, even though other threads might have been accessing the object [Lea00].**

This situation assumes the same programming style as the previous one and in particular that no timeouts are used in a wait(). It identifies a situation, where a thread keeps waiting without being notified. Only an interrupt() could change the situation but we do not know if this will happen.

The fault behind this failure is that no thread can issue a notification, because there is no code containing notify()/notifyAll() or the code is never used or not used in this program run because certain conditions are not met. Also here, an interrupt or a timeout could wake up the thread.

This situation could involve more than one waiting thread. E.g., a second thread could call wait on the same object. Note that in the absence of the programming convention it cannot be decided in general, if a notification will be send by another thread.

#### F6 Nested Monitor Lockout

**A thread waiting in the queue of an object does no longer hold the lock for that object but it can hold locks to other objects. We assume that the wait is not a timed wait but that the thread is dependent of other threads to wake it up. This can lead to a situation where any other thread which is supposed to wake up this thread can only access the object at which the thread is waiting**

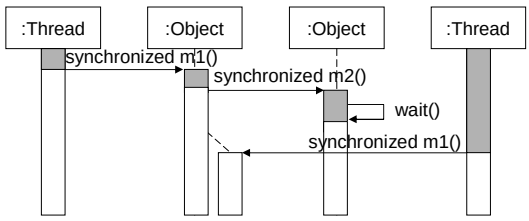


Figure 5.9: Nested Monitor Lockout

**via at least one of the locked object. The locked object is also called the outer object. The object where the thread is waiting is also called the inner object. Because the outer object is locked no other thread can access the inner object [Lea00].**

This situation is depicted in Figure 5.9. The thread on the left obtained a lock to the left object before accessing the right object. Calling wait() on the right object does release the lock on that object but not on the left object. The second thread becomes blocked when accessing the left object. Only interrupting the first thread could change the situation but we do not know if this will happen. Interrupting is feasible, because the first thread can re-obtain the lock for the inner object.

The fault in the code is that the thread is keeping the lock. The question to be answered is how can we be sure that there is no other way to accessing the object but through the outer lock. Generally, this question can only be answered by looking at source code. Determining access relation cannot be efficiently be computed because of aliasing problems. A further hint to a problematic situation could be that other threads start being blocked at the object which is locked by the waiting thread but this cannot be a guarantee that the thread is waiting forever.

This situation bears similarities with the deadlock because of the circular waiting involved. The first thread is waiting for a signal of the second thread while the second thread is waiting for a resource from the first thread. Both threads will not send the signal or release the resource before the desired state change takes place.

Note that in a similar situation where join() is called instead of wait() the thread is not blocked forever. Here, a thread obtains a lock and then calls join() on a different thread. Thus it keeps the the first lock but will return when the joined thread is is terminated.

### F7 Circular Join

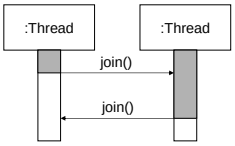


Figure 5.10: Circular Join

**In a circular join, a first thread is waiting via a call to join() for a second thread to terminate. The second thread is waiting via join() for the first thread to terminate.**

This situation is depicted in Figure 5.10. However, a thread not involved in the circular join could interrupt() one of the two threads which would cause the interrupted thread to return from joining immediately. The second thread, still joining, would return once the first thread is terminated. Nevertheless, we consider the circular join a dangerous and unwanted situation because it does not make sense having two threads joining themselves and waiting for a third thread to interrupt() one of them. Note that an arbitrary number of threads could create a circular chain of joins.

On an abstract level, this situation is similar to a deadlock because it involves a circular waiting. The fault in the code is that the parameters of join() allow mutual joining. Also, no timeout is used with join(), which is always a risk. A join() can be aborted through interrupting, but again, one does not know if such interrupt() will be called in another thread.

If a thread joins another thread which is calling wait() this could be a failure if the other thread depends on the joining thread to be woken up. Note that joining a thread which has not yet started or which is already terminated could be a situation not intended by the developer but it is neither a liveness nor a safety failure.

### F8 Self Join

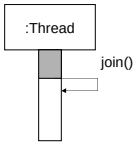


Figure 5.11: Self Join

**A thread joining itself [OW99] can not return from joining. The only exception is that it is is interrupted by another thread.**

For an example, see Fig. 5.11, which depicts one thread joining itself and thus not making progress.

The fault in the code is simply a wrong target parameter of join() rsp. the code which computed the actual thread passed as parameter.

### F9 Join-induced Deadlock

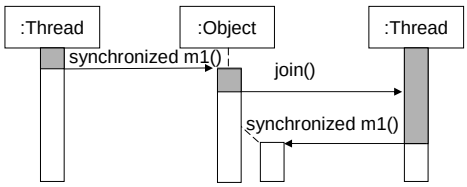


Figure 5.12: Join-induced Deadlock

**A first thread holding a synchronized-lock is joining a second thread and thus will be blocked until this thread is terminated. After the call to join(), the second thread is acquiring the same synchronized-lock. This produces a cycling waiting. The first threads is waiting for the terminating signal before it can release the lock, the second thread is waiting for the lock before it can terminate .**

An example is depicted in Figure 5.12. The thread on the left obtians a lock and then joins the second thread. The second thread tries to obtain the lock which is never released. Only interrupts and timeouts can remove threads from the situation.

The fault is that the joining thread holds at least one lock and that its target was not chosen carefully. The situation would not be different if the second thread acquired the lock even before the first thread calls join(). The same situation also exists if the second thread would still be in a wait-queue of the object which is used as lock. When the thread is notified, it has to re-obtain the lock which would be impossible. This situation cannot be reduced to a normal deadlock such as in the case of the wait-induced deadlock.

### F10 Livelock

**A thread can be actively waiting, usually by looping, for a condition to come true.**

This is also called *busy waiting*. If, however, the condition will never be true and the thread cannot make progress, the situation is called a *livelock*

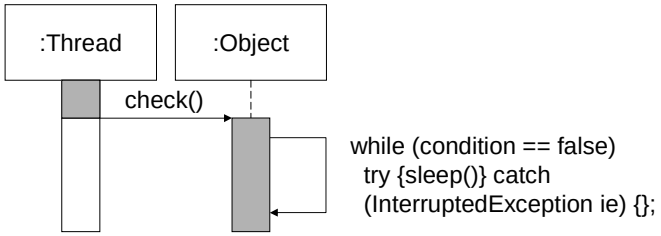


Figure 5.13: Livelock

[Lea00], An example is given in Fig. 5.13. In the loop, the thread chooses to sleep for some time before it tests the condition again.

This failure is well-known from sequential programming. Here, not only one thread but the complete program does not make progress because of a livelock. In concurrent programming though, this failure has the same symptoms as other liveness failures and is therefore not easy to identify. The fault in the code is the looping over a wrongly chosen condition.

To detect the failure, the program history has to be examined for repeated loops where the loop condition is not changing. Whether the loop is a livelock or not can however not be decided in general. It can only be decided for trivial cases where the loop condition is a true constant and where the statements in the loop will never cause exceptions or breaks.

### F11 Blocking on I/O

**A thread could be indefinitely blocked on an I/O method call.**

This could be the case, when a device for input or output is not reacting because it is in an inconsistent state or because the device is constantly used by other threads. This failure is also well-known from sequential programming.

Usually, the code of the program using the device cannot be made responsible for device problems except when it violates preconditions when calling it. Heuristics are needed to decide when a blocking on I/O takes too long.

### F12 Termination by Exception

**In a multithreaded Java program a thread can terminate due to an uncaught *exception* with the effect that the stack trace of the thread is printed to the error console.**

If that thread is the main()-thread the complete program will exit. In the case of other threads, only that thread will terminate but all other threads

keep running. Only in the case of a TreadDeathException, no such trace is printed. The termination might even go unnoticed if the thread does not print a stack trace on the console. It might be noticed only some time after the thread has vanished and then it is not known when and where which exception was thrown. The unexpected termination of a thread might leave the objects used by that thread and shared with other threads in an inconsistent state.

Note that symptoms are not different when threads are blocked or vanish. Both failures may cause inconsistencies, wrong or missing results, or lack of performance.

The fault is the lack of a corresponding exception handler. Exceptional termination itself is not specific to concurrency although some exceptions are caused by concurrency mechanisms such as InterruptedException or IllegalMonitorStateException. The latter exception points to a fault in the code which is only detected at runtime. The fact that the exception is not yet handled is however the fault addressed here. The InterruptedException is used to signal a thread that it was interrupted. The exception itself does therefore not point to a fault but the fact that it is re-thrown and not handled is the fault addressed here.

User-defined exceptions typically signal an inconsistent state, which can be a safety failure. Thereby, a safety failure could turn into an exceptional termination if not caught and hence into a liveness failure.

Because of the various kind of exceptions and the various ways to use it, one cannot interpret all of them as failures. Still, many of them bear the risk of failures. It is essential to provide runtime detection of otherwise unnoticed exceptions and to provide as much information about their occurence as possible.

### Thread Restarting

*Restarting* a thread in Java can either throw an exception or may go unnoticed if the JVM has already cleaned up behind a terminated thread.

This is not a failure in the above senses as this thread never becomes alive and thus cannot fail to make progress. This failure might have the same symptoms as other liveness failures as the thread cannot do the intended computations. Other threads can however not become blocked when trying to interact with this thread.

The fault in the code is that the same thread object is started more than once. It could be the case that the creation of a corresponding new thread object before calling start() is missing.

### Ripple Effects

All of the above failures have the same symptons. Symptoms can be missing or wrong results or computation states or lack of performance. It is also a specific characteristic that these failures can remain unnoticed. When a liveness failure happens the threads involved stop to run and can never continue running. Other threads of the program are not affected by this liveness failure unless they are engaged in a particular interaction with any of the threads involved in the failure.

This can be the case if another thread calls join() on a thread involved in the failure or is waiting for a notification by a thread involved in the failure. If the threads involved in the failure are holding locks other threads can be blocked when using synchronized. Therefore, each of the above liveness failures can have a *ripple effect* on the entire system.

### A Note on Stall, Dormancy, Starvation and Bottle Neck

Finally, we shortly mention a few terms often appearing in the context of concurrency problems which we have neglected so far and which do not play a role in this thesis because they either fall outside the area of liveness failures or because they are less appropriate than the terms we prefer to use.

A *Stall* refers to a situation where a thread cannot proceed for a certain time span. The thread will however be able to continue again. Thus a stall is a delay. Stalls can have many different reasons. The drawback of this term is therefore that it does not exactly describe the situation. A stall is not a liveness failure and thus will not be considered in the following.

*Dormancy* refers to threads waiting in queues that are not being notified. It is not defined whether they will be notified later or never. Because of this and because dormancy can have many different reasons it is not a precise term. Instead, we have classified reasons of dormancy by distinguishing between a missing and a missed notification. This provides a better characterisation than the more general term and this is the reason why we decided not to use dormancy throughout this thesis.

*Starvation* typically refers to a thread not making progress as a consequence of the scheduling policy. Although Java guarantees fair scheduling this problem is not avoided in practice. Scheduling is however an issue on its own and is not discussed in the context of liveness failures because it cannot be influenced by the programmer in the same way as liveness failures. Starvation might also refer to *lock starvation*, which we have already discussed in the context of *indefinite blocking*.

*Priority-inversion* is a special case of starvation. It refers to a problematic

situation where a medium-priority thread prevents a low-priority thread from running and thus from releasing a lock which a high-priority thread needs for making progress. This situation is more accurately termed *unbounded priority-inversion* meaning that the situation cannot be resolved. Priority-inversion can be a problem in Java depending on the scheduling policy of a specific JVM. Java fails to require scheduling-protocols avoiding priority-inversion. The reason for this total cop-out on the part of Java is presumably because of its desire for platform independence, since it has to run on a variety of operating systems. Consequently, multithreaded Java programs can run well in one operating system, and not run at all in another operating system. As said above, issues related to scheduling don't fall in the area of liveness failures and will not be considered in this thesis.

*Bottle Neck* refers to states in a program where the performance is decreased by an increasing number of threads or by issues related to thread interaction. Performance issues are not related to liveness failures. Not only the sheer number of threads running concurrently can affect performance through the context switches or through the cost of creating and deleting thread object, but also the overhead of synchronisation mechanisms. Locks can become bottle necks for threads. The locking itself takes time which is referred to as *Lock Overhead*. Sometimes, programs contain too much locking and locking and even locks can be avoided. Often, there is a trade-off between coarsening lock-granularity to avoid locking overhead and using fine-grained locks to reduce the number of threads competing for a lock. None of these problems is considered a liveness failures and therefore is not an issue here.

### 5.2.3 Potentials for Java Liveness Failures

Some potentials are identified in a system state and others in an execution history. It is possible to determine potentials for some of the above introduced failures. For other failures there are no meaningful potentials. In general it is difficult to establish conditions for a potential if only one method per thread is involved.

For example, in the case of a circular join it is impossible to establish a potential. The dynamic conditions of the circular join are the two mutual method calls. The only condition weaker than that would be one method call but this condition does not make sense because then every call to join() would be treated as a potential.

### P1 Deadlock Potential

In the case of the deadlock its conditions (see definition 5.6) can be classified in necessary conditions which are always fulfilled by a language or system and in a sufficient condition whose fulfillment depend on a concrete program or a concrete state or execution of a program.

- In the case of Java, the condition for non-preemption holds a priori for any program written in Java. Therefore the fulfilment of this condition can not serve as a warning for a concrete program execution.
- Mutual exclusion holds as soon as threads need to share resources with mutual exclusive access using synchronized. Programs can only avoid it when avoiding overlapping concurrency by completely serialising the program which is not very efficient. In practice, systems will use mutual exclusion. Therefore, the use of mutual exclusion is not a good candidate for a warning either.
- Incremental acquisition, i.e. the incremental locking via synchronized is also fulfilled in any Java program which uses synchronized without further restrictions. In a dynamic system where objects are generated at runtime incremental acquisition might be unavoidable. Therefore, it is not a useful candidate.
- The condition for cyclic dependencies is fulfilled if synchronized-objects are accessed in inverted orders and if that access overlaps in time. Typically, programs with dynamic creation of objects can not easily enforce order and therefore, inverse order is a widely used candidate for detection of deadlock potential. The condition of inverse order is weaker than the condition of cyclic dependencies because it omits the overlap in time. Therefore it can be used as a condition for a potential of a deadlock. Also note that detecting inverse order implies that the condition of incremental acquisition and mutual exclusion hold.

### P2 Unused Notification

Another potential can be established based on the problem of missed notification. Disregarding the discussion that it is based on a programming convention, it is possible to consider isolated notifications as troublesome. By isolated notification we refer to notifications which have not been used to wake up a thread because none was waiting. Such a notification can be seen as a potential for a missed notification itself because it was issued at an inappropriate time.

Concerning our method to establish conditions for potentials the conditions for the potential of missed notification are weaker than the one for the missed notification because they do not require that the notification is followed by a wait.

#### P3 Unused Interrupt

The interrupt mechanism bears the potential for an InterruptedException. If a thread was interrupted but has never polled or consumed its interrupt state it will throw an exception when it enters a wait(), join() or a sleep(). We said earlier that this exception has the intent of signaling the interrupt to a thread which is not in a state where it can actively poll the interrupt flag.

If an interrupt is not polled by the interrupted thread after a certain elapse of time this can be seen as a potential liveness failure because it can lead to an exception when the thread calls wait(), join() or a sleep(). As it is best practice to suppress this exception by providing an empty handler in code, it can also be seen as a failure to participate in the interrupt mechanism.

#### P4 Potential for Join-induced Deadlock

There are two conditions for a join-induced deadlock. One thread joins another while holding at least one lock. The target of the join attempts to acquire a lock held by the first thread. Both together are sufficient if they appear in the above order.

There could be an execution, where a thread acquires a lock and subsequently releases it. Later, the thread becomes the target of join(). The joining thread holds the same lock while joining. This situation has the same conditions but a different timing and is therefore a potential for the join-induced deadlock. If the first thread is to acquire the lock after join() has taken place, both threads are blocked.

Joining while holding locks can generally be considered as a potential for failure but this can happen very often in arbitrary Java programs.

### 5.2.4 Classification of Java Liveness Failures and Potentials

We have motivated not only the need for a comprehensive list of concurrent Java liveness failures, but also that it is desirable to analyse their common reasons.

We consider only those liveness failures and potentials as *concurrent* which involve synchronisation mechanisms. Therefore, we do not consider

livelock (see F10) and indefinite blocking on I/O (see F11) as concurrent failures (see also Table 5.1).

It is obvious that concurrent liveness failures and potentials can be categorised based on the language means involved, e.g. synchronized, join() or wait(), based on the number of threads involved, and whether they can be detected in a state or in a history (see also Table 5.1 and Table tab:formalpotclassif).

### General Dimensions

Concurrent liveness failures and their potentials

- manifest themselves while a program is executing,
- involve more than one thread (except for self join and termination by exception), and
- have their origin in the synchronised interaction of threads.

Failures are manifest in either a state or in an execution history. Failure states are legal system states. System states are composed by the states of each thread disregarding for the moment the states of passive data objects.

Failure states differ from other system states in the combination of thread states. In these combinations, threads need resources or events from each other, in order to make progress. To be more precise, at least one thread involved in a failures is depending on other threads.

Failure execution histories are also legal execution histories. The reason why some failure cannot be described as a state but only as a history, is that either the effect of a state change has been overridden by new state changes or that there has not been a related state change at all. For instance, if behaviour is missing, e.g. a missed notification, we cannot observe that in any state. Another example is that behaviour has taken place but has not persisted in a state change such as a call to notify().

### Thread Dependencies

We have already pointed out how failure states and histories are different from other system states because they involves special dependencies. It is common to several kind of liveness failures that these dependencies are cyclic. We can identify three different classes of waiting-dependencies between two threads:

- waiting for a lock, which we also call *blocking*

| Concept | Java Syntax | Liveness Failures | State | History |
|---|---|---|---|---|
| *Lifecycle Control* - Termination | run() | Uncaught Exception (F12) | x | |
| *Unsynchronised Interaction* - Direct Interaction via Interrupt | interrupt() | Uncaught Exception (F12) | x | |
| *Synchronised Interaction* - Joining | join() | Circular Join (F7), Self Join (F8) | x | |
| *Synchronised Interaction* - Joining | synchronized, join() | Join-induced Deadlock (F9) | x | |
| *Synchronised Interaction* - Mutual Exclusion | synchronized | Deadlock (F1, F2) | x | |
| *Synchronised Interaction* - Conditional Synchronisation | synchronized, wait() | Nested Lockout (F6) | x | |
| *Synchronised Interaction* - Conditional Synchronisation | notify/All(), wait(), | Missed Notification (F4), Missing Notification (F5) | | x |
| *Synchronised Interaction* - Mutual Exclusion | synchronized | Indefinite Blocking on synchronized (F3) | x | |
| *General Concepts* - I/O | (not specific) | Indefinite Blocking on I/O (F11) | x | |
| *General Concepts* - Loop | while | Livelock (F10) | | x |

Table 5.1: Liveness Failures

- waiting for the termination of another thread during *joining*
- waiting for a *notification* from an arbitrary other thread

These three dependency can create several different failures involving cyclic dependencies, namely deadlock, wait-induced deadlock, nested monitor lockout, join-induced deadlock, self join, and circular join. The basic pattern is always the same, although sometimes threads wait for a resource and other times they wait for a signalling event.

Hence we can identify blocking, locking, joining, and waiting as the source of failures and of the corresponding potentials. This once more shows, that in concurrent programming, failures are unintended combinations of legal system behaviour.

| Concept | Java Syntax | Failure Potential | State | History |
|---|---|---|---|---|
| *Unsynchronised Interaction* - Direct Interaction via Interrupt | interrupt | Unused Interrupt (P3) | | x |
| *Synchronised Interaction* - Mutual Exclusion | synchronized | Deadlock Potential (P1) | | x |
| *Synchronised Interaction* - Conditional Synchronisation | notify(), wait() | Unused Notification (P2) | | x |
| *Synchronised Interaction* - Joining | synchronized, join() | Joining with Locks (P4) | | x |

Table 5.2: Potential Liveness Failures

### Synchronisation Mechanisms Involved

We have presented a few failures specific to both, sequential and concurrent programs, and many failures specific to concurrent programs only. For the latter it can be observed that they are based on a set of synchronisation mechanisms of Java.

There exists a group of failures directly related to the concurrency concepts of synchronised interaction (see Table 5.1). These failures are *circular join, self join, deadlock, nested monitor lockout, missing notification, missed notification*, and *blocking on* synchronized *or* wait. When a failure occurs,

this is due to the effect of a certain method call, and in some cases due to the absence or wrong timing of a method call.

Two tables extend the table presented at the end of the previous chapter by adding the failure (see Table 5.1) and potentials (see Table 5.2) which can be caused by the language concepts.

### 5.2.5 Failures in Concurrent Application Logic

The Java synchronisation features can be used to construct high-level synchronisation protocols such as protocols which allow concurrent reading threads but only mutually exclusive writing threads (also known as reader/writer protocols [Lea00]).

In these protocols it can also happen at runtime that a thread does not make progress, e.g. if the protocol does not guarantee fairness. Technically, such a thread is either blocked at a lock or waiting in a queue or joining, as there are no other possibilities. Usually, these protocols use waiting to coordinate threads.

In such a case, we would identify the failure that a thread fails to make progress in the presence of the protocol as a missing or missed notification. However, if we took the concrete protocol into account we could give a more precise definition. That is, we are able to detect that something is wrong with a protocol but we cannot provide a protocol specific analysis based on the failures we have defined here. We can however detect if protocols created unintended liveness failures like deadlocks.

## 5.3 Summary

In this chapter we have described a range of failures and potentials. We described why the threads involved did not make progress and we have explained the faults, i.e. the causes or reasons behind these failures. Our focus was on liveness failures inherent in the Java language.

We have described new failures and potentials which we could not found in literature. The join-induced deadlock and its potential have a structure similar to the deadlock and the deadlock potential. We were able to derive them using the model of thread synchronisation which we describe in the next chapter. Failures not documented yet like circular join and self join are more obvious and easier to conceive and were also discovered using the model of thread synchronisation.

# Chapter 6

# A Model of Thread Synchronisation

The previous two chapters have introduced the domain of this thesis by presenting concurrent programming in Java and by informally presenting concurrent liveness failures in Java and their potentials. The reasons for these failures and potentials were analysed and classified. As a result, thread states and dependencies which potentially lead to waiting and blocking were identified. They have their origin in Java methods and statements for thread synchronisation.

The previous chapter used UML sequence diagrams to describe the different failures. Sequence diagrams can however not depict the states and dependencies and describe failures only scenario-based. Therefore, this chapter aims at precisely describing the behaviour of threads on a level which makes states and dependencies explicit. To this end, a *domain model* for synchronisation of Java threads is described which makes states and dependencies of thread behaviour explicit. Based on the model, failures and potentials can be specified more generally.

This chapter starts by deriving requirements for the model from the domain of concurrent liveness failures in Sect. 6.1. It is followed by a discussion of related work in the area of formal Java semantics in Sect. 6.2. Then we will present our statechart-based formalisation of the dynamics of Java thread synchronisation in Sect. 6.3. In Sect. 6.4 we will use the developed model to specify concurrent liveness failures and potentials.

## 6.1 Model Requirements

In this section we will determine the requirements for a model of Java threads which will enable us to capture Java concurrent liveness failures and their potentials.

The model has to cover the characteristics of the concurrent liveness failures and potentials. These characteristics have been informally identified in the last chapter and are summarised here for convenience in the next subsection. Thereafter, we will analyse these characteristics in order to derive requirements for the model.

### 6.1.1 Concurrent Liveness Failure Characteristics

In this subsection we summarise the characteristics of the concurrent liveness failures which have been presented in the previous chapter. There, failures were characterised according to two dimensions, the Java concepts involved and the way how they manifest during a program execution.

In the following we will not consider liveness failures which can happen in the same way in a sequential program such as the livelock and the blocking on I/O. We will only consider liveness failures specific to concurrent Java programs and their potentials.

#### Thread Synchronisation

The last chapter concluded with two tables (see Table 5.1 and Table 5.2) of Java concepts involved in concurrent liveness failures and potentials. The involved concepts are

- joining,
- mutual exclusion, and
- conditional synchronisation.

These concepts have been introduced in Chapter 4 together with the corresponding Java language elements: the methods wait(), notify(), notifyAll(), join(), and the keyword synchronized. Together, they form the Java part dealing with *thread synchronisation*.

#### State and History

Liveness failures and their potentials manifest themselves while a program is executing, either in a specific program execution *state* or in a specific *history*

of program states. All these states are legal system states, consisting of the state of each thread and of its potentially shared objects. Therefore we need to characterise how failures and potentials states *differ* from other states and execution histories.

We can distinguish two kind of failure states. A deadlock-like failure is a legal system state where

- The thread(s) involved cannot make progress and are blocked forever (either blocking on a lock, waiting, or joining). Their interactions have created cyclic dependencies such that progress of each thread depends on a resource not yet released or signaled by another thread.
- One or more interaction from each thread is involved.

The other kind of failure state does not involve cycles.

- Some threads directly involved do not make progress due to blocking after an interaction.
- The reason is the missing of an interaction of other threads. These threads are said to be involved but they are not affected by the problem and can still make progress.

There are also two kinds of potentials.

- One kind is manifest in a state, i.e. if certain conditions hold when a thread makes an interaction.
- The other kind is manifest in an execution history where conditions over the history are identified as potential for failure.

In the following we will analyse the above characteristics in more depth which will allow us to formulate our requirements.

### 6.1.2 Dynamics of Thread Synchronisation

In this section we determine the scope of the model. At first glance, it seems obvious that the scope is restricted to failures. But when we examined failure states more closely, we identified each of them as a legal system state. The Java concepts involved in failures can also be involved in many other system states not identified as failures. This inclusion relationship is depicted in Fig. 6.1. Regarding potentials, they are legal system states or execution histories, too.

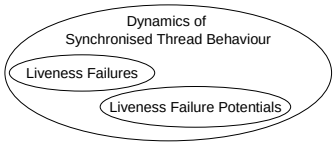


Figure 6.1: Failures and Potentials as Legal System Behaviour

Therefore, capturing failures requires to cover the legal behaviour to which failures belong. Then failures can be specified as specific cases of this behaviour. This *two-step* approach will be followed in the sequel.

Liveness failures and their potentials have their origin in the thread synchronisation. Therefore, the formalisation has to cover *Java thread synchronisation*. As the failures and potentials manifest themselves at runtime we need to model the *dynamics*, i.e. the runtime behaviour of Java thread synchronisation. Note that for the model we are only concerned with the *dynamic semantics* of Java thread synchronisation, but not with the so-called static semantics or grammar.

The model has to provide semantics for the the methods wait(), notify(), notifyAll(), join(), and the keyword synchronized. As the the Java methods wait() and join() can throw an InterruptedException, it is obvious that the interrupt mechanism is intertwined with the concepts of joining and conditional synchronisation. Therefore, we choose to cover also interrupts in the formalisation, implemented by the methods interrupt(), interrupted(), isInterrupted(), and the InterruptedException.

The resulting set of methods, keyword, and exception does not only cover thread synchronisation but almost the entire *thread lifecycle*, i.e. the behaviour common to all thread objects, except for creation, starting and sleeping of threads (see also chapter 4) . For the sake of completeness we decide to cover the entire thread lifecycle with our model. That means, we are going to integrate the constructor new() and the methods start() rsp. run(), and sleep().

Note that these methods and synchronized calls do not only change the lifecycle of the thread who called them but also the lifecycle of the called thread. They can be called concurrently from other threads. Here we are only concerned with their effects on the thread whose lifecycle we describe, but the model has to respect these multithreaded calls because the lifecycle also depends on the other threads behaviour.

### 6.1.3 Control Flow States

In this section we determine the level of abstraction for modelling the thread lifecycle. It is important to identify, in which states of the lifecycle a thread is unable to proceed. Therefore, one has to analyse how a thread changes state. In order to better understand what characterises the states of the lifecycle which are involved in failures we discuss the characteristics of *concurrent program state*.

In general, a program state manifests itself in the state of its *data* and its *program counter*, i.e. the pointer to the position of the control flow respectively point of execution. In the case of a concurrent object-oriented program the data state refers to all object states including thread object states. The program counter consists of a counter for the main()-thread and for all other threads. In a concurrent program, the continuation, i.e. the next execution step of each thread, is not only determined by the counters but also by additional information about the state of each thread with respect to concurrent activities such as synchronisation or other interactions between threads. Depending on this information, a thread can react differently to a method call. Moreover, synchronisation and other interactions can prevent the continuation of a thread and put a thread in a state where it cannot proceed.

We therefore make the following important distinction between two kinds of thread states:

- the state of the *thread object* itself, determined only by attributes either defined in its class or inherited from super classes, including Thread.

- the state of the *control flow* associated with the thread object, i.e. the program counter *and* the thread state with respect to synchronisation and any other information about a thread which determines the continuation. This state can be changed by synchronisation and interaction mechanisms used by the thread itself but also by other threads.

Note that the thread object state and the control flow state are *orthogonal*, i.e. independent of each other. The object attribute values are not used for encoding control flow states. The control flow states of Java threads are part of the language and cannot be extended. Each thread, whether an instance of Thread or of a subclass, is always in one of these states.

The thread lifecycle can be fully characterised in terms of control flow states. Regarding the list of methods, keyword, and exception which serve to implement the thread lifecycle, their behaviour is dependent on the control flow states and they can change it. Therefore, a model has to describe the

semantics of the methods implementing the Java thread lifecycle *in terms of the control flow states*.

Many control flow states serve to identify pre- and postconditions of method calls. However, threads engaged in a method call, i.e. threads which have not yet returned from a call, can also react in different ways to incoming concurrent calls or events because they can leave that method in different ways. That means, that *intermediate states* are also control flow states because they influence the behaviour of a thread by having an effect on state changes. However, these states never occur as pre- or postconditions but they are states reached only *during* a method call.

### Example

An example for an intermediate control flow state is the state, where a thread is in state *blocking* while waiting for a lock on an object after having called a synchronized-method or after having entered a synchronized-block. When the lock is eventually granted, the thread changes its state to *running* while holding a lock. A thread in state *blocking* reacts differently than, e.g. in state *waiting* to an interrupt()-call from another thread. In state *blocking*, the interrupt flag is set. In state *waiting*, the thread immediately returns from its call to wait() and the flag is not set.

### Control Flow States and Failures

Only control flow states which describe different kinds of blocking are involved in failures. In order to describe failures, these states have to be identified by our formalisation. Therefore, the required *granularity* of the model has to be that of control flow states.

From the list of failures we can identify which states need to be distinguished. Failures involve *blocking* on a lock, *waiting* and *joining*. Therefore, we require that these states are made explicit by the formalisation. It can be observed that the blocking states are all intermediate states of some method execution or synchronized-block execution.

A control flow state for a concrete thread can bear *additional information* which is specific for this thread. For instance, a thread in state *locking* has locked a certain object and a thread in state *blocking* is waiting for a lock on a certain object. This is the kind of information needed to describe failures involving cyclic dependencies. Therefore, the model has to cover *thread instance-specific* information associated with a control flow state.

### 6.1.4 Summary of Model Requirements

It is the primary goal to describes the states in the lifecycle where a thread cannot make progress and is potentially blocked forever. Control flow states have been identified as a useful abstraction for capturing the characteristics of such states. Therefore, the model has to describe how the Java methods and calls to synchronized involved in the thread lifecycle are dependent on control flow states and how they change control flow states.

We can conclude the following requirements:

- The scope of the model is the thread lifecycle.
- The model has to capture the behaviour of methods and statements having an effect on the thread lifecycle in terms of control flow states.
- Control flow states are chosen at a level of granularity which allows us to distinguish all states involved in failures, especially intermediate states during method executions.
- The model has to capture instance-specific information of control flow states.
- Based on this, failures and potentials have to be formalised as specific states and sequences of states.

Thus, we are only going to model a dedicated part of Java.

Note that for our model it is not required that it is executable or that it can to be formally analysed. We provide the model in order to precisely and systematically describe concurrent liveness failures in Java and their potentials and to develop support for their automated detection and visualisation.

## 6.2 Related Work

Aiming at a model of the behaviour of Java thread synchronisation is related to efforts in the area of formal Java semantics. It seems obvious to consult the official *Java Language Specification* (JLS) [GJSB00]. However, this specification provides only an informal yet precise description of Java syntax and semantics. Nevertheless, this is the only source available from which formalisations can be derived, and equally important, the only source against which any formalisation can be proven or validated. Existing Java implementations, i.e. compilers and runtime environments only serve as reference implementations, such as the ones by Sun Microsystems [Jav01], but not as a specification.

As we plan to provide our own model we discuss the JLS in this section under the perspective of deriving a model, in general, and also with respect to our requirements.

Not surprisingly, many thorough attempts have been made in research to overcome the lack of formal semantics for Java. Because they cover most parts of Java it seems practical to reuse one of these. In this section, we will discuss briefly some state-of-the-art approaches and argue why we have chosen to provide our own model of thread synchronisation in Java.

### 6.2.1 The Java Language Specification

We start with a few general observations related to the organisation of the presentation of *Java threads* within the official Java language specification (JLS) [GJSB00]. The semantics of threads is scattered over a few chapter in the JLS. The reason for the scattering is that the JLS is driven by the syntax. This requires to deal with the method modifier synchronized in a different place (JLS chapter 8) than with the synchronized statement used to specify a block (JLS chapter 11).

A more unifying presentation is given in the JLS chapter 17 titled "Threads and Locks" which discusses the semantics of unsynchronised and synchronised shared data using synchronized and also wait() and notify/All(). This chapter provides a complete treatment of locking, i.e. with respect to the use of the synchronized keyword. It provides a basic description of wait() and notify/All(), but this description is incomplete with respect to the interrupt mechanism involved. The built-in exception InterruptedException, thrown by wait(), is not specified.

Not only wait() and notify/All() of class java.lang.Object are not completely covered by the JLS but also all other methods, classes, and interfaces dealing with threads which are part of java.lang, most importantly the classes Thread, ThreadGroup, and the interface Runnable. Their semantics are described in the documentation of the package java.lang. Although these language elements are closely related to the Java language itself, these packages are not documented in the current version of the JLS [GJSB00]. This clearly has a disadvantage for understanding one of the most complex parts of Java. An integrating presentation is therefore desirable but needs a thorough analysis of the relevant chapters of the JLS and of the documentation of java.lang.Thread.

The specification style of the JLS and of the related package java.lang does not exploit commonalities among the preconditions and postconditions of method behaviour. The specifications of preconditions and postconditions are per method (or per statement). Thereby, the JSL and also the Thread

related libraries do not make control flow states explicit. Analysing commonalities is not only difficult because of the scattering but also because the JLS is pure prose.

There is one part of the JLS which is not pure prose but contains well structured textual *rules*. This is the so-called *Java Memory Model* (JMM) which is the major part of the JLS chapter 17 [GJSB00]. The rules specifies how unsynchronised and synchronised data (by means of synchronized blocks and methods) is shared among threads between *thread local memories* and one *global memory*, using a set of *primitive operations* for data transfers and for locking and unlocking an object. The keyword synchronized is mapped to the *lock* operation primitive, followed by the *unlock* operation primitive, both operations are carried out by a thread and by the global memory *together*, i.e. "synchronised". Formally analysing the JMM reveals some problematic cases for hard to detect safety failures which has been the focus of several formal approaches [RM02]. However, safety is not our focus. For our approach we are interested in the behaviour of locking. The semantics of synchronized is covered by rules which guarantee that all lock and unlock requests in a program are totally ordered. Also, lock requests are serialised, i.e. simultaneous requests for the same lock cannot interfere. Only one lock request can return successfully while the other one is blocked.

The way how the JLS deals with the specification of locking is not suitable for our requirement. In analogy to the Java language, also the primitive operations *lock* and *unlock* of the JMM, which lay claim to a lock of an object and which release exactly this claim, do not allow to distinguish the *intermediate* state where a thread is blocked having called *lock* because the lock is not available. Essentially, the call to *lock* returns only after successful locking, what happens internally is hidden. We want to be more explicit and go beyond this level of abstraction, i.e. we want to identify state changes inside the call to the operation *lock*.

### 6.2.2 Formal Java Semantics

We briefly present some well-known rigorous approaches and discuss why they do not completely meet our requirements. The chosen approaches cover multithreading in Java, however to different extents. As already mentioned, many approaches dealing with multithreading aim at formalising the Java Memory Model, which is not our focus.

Existing operational semantics [CKRW99, BS99] based on Structured Operational Semantics (SOS) and Abstract State Machines (ASM) respectively, fall short in completely covering thread synchronisation. The concepts of joining, sleeping, and interrupting, which are tightly coupled with conditional

synchronisation, are not covered. Another missing feature is the discrimination of successful from unsuccessful locking. The blocking on locks, i.e. the phase when a thread is acquiring a lock until it might be granted, is not made explicit as a separate state. Instead, this behaviour is encapsulated in a function which returns only when the lock is granted, similar to Java synchronized and similar to the JMM *lock* operation. The executable operational semantics by [RM02] only address the JMM primitive operations using Guarded Commands. They also do not distinguish intermediate phases of locking. Moreover, they have reduced complexity by working with only one lock for the entire Java program which locks a set of objects. Even when these approaches are semantically correct they are not at the desired level of abstraction. We need to discriminate the different phases involved in locking and other blocking activities in order to allow precise description of failures.

The denotational approach by [WM00] using CSP is also not yet complete with respect to thread synchronisation. The present version does not yet deal with waiting and notification. Another drawback is that translating Java into CSP changes not only the level of abstraction but also the concepts, e.g. with regard to object-orientation. Hence, the failures of the original program take a different shape in the CSP domain. This prevents an intuitive understanding and requires to map forth and back. This is not necessarily the case with any denotational semantics but it is a typical effect. Because of these disadvantages a more intuitive formalisation is desirable.

In any case, working from an existing formalisation would make it necessary to *extend the scope* and, in many cases, *extend the handling of the locking state*. We do not consider integrating the missing features into an existing approach as unfeasible. However, because we aim at formalising only a dedicated part of Java, namely thread synchronisation, implemented by a set of methods and statements, we do not want to take the burden of extending an entire Java formalisation. Therefore we decide to provide our own model dedicated to thread synchronisation.

## 6.3 A Statechart Model for Thread Synchronisation

Given our requirement that the model of the dynamics of the thread lifecycle should make states as pre- and postconditions and as intermediate states explicit, we decide to use *statecharts*. Thereby we can achieve the following:

- Statecharts make control flow states explicit at the desired level of granularity.

- They capture how execution of methods depend on control flow states and how they change these states.
- Statecharts thereby capture commonalities in the pre- and postconditions of method call events.
- Statecharts are a visual and intuitive formalism for capturing state-driven behaviour.

In comparison to other approaches based on states and transition, the hierarchical mechanisms supplied by statecharts support factorisation of common behaviour of a group of states. Most statechart formalisms allow specification of side effects on variables, which can be used to model the instance-specific state information. Because we do not intend to use the statechart for formal reasoning, the chosen statechart approach does not have to be executable.

A Java thread is object-oriented. Therefore, we decide to use an *object-oriented* variant of statecharts. This allows us to assign statecharts to instances of Java classes. Also, directed communication between instances is supported in object-oriented statecharts. Object-oriented statecharts support for communication not only by asynchronous *events* as in classical statecharts but also by *method calls*.

In the following we first present the chosen object-oriented statechart approach. We then further classify the Java concepts involved in the lifecycle, so that we can identify which concepts can be mapped in the same way to statecharts. Then we will map the thread lifecycle to the chosen statechart approach. Based on this we will formalise system state and history.

### 6.3.1 The UML Statechart Approach

The Unified Modeling Language (UML) [UMLa] provides an object-oriented approach to statecharts. UML statecharts can be used for modelling behaviour of classes but also for other behavioural elements such as operations. Here, we intend to use them for modelling the behaviour of a class. We do need to model in which state a method call can be processed and we need to model the effects of calls. These effects may include actions in order to encode instance-specific behaviour. Therefore, a protocol statechart simply defining order is not sufficient but we need to use a behaviour statechart, following the terminology of the UML.

In the following we describe the semantics provided for UML statecharts [UMLa] on which we rely for our formalisation. Where the semantics is left open or where it imposes problem we will refer to other standard semantics

such as the semantics for Rhapsody statecharts [HK04, HG97], in order to clarify the situation. The syntax will be given while presenting our statechart solution.

### Event Processing

A queue is associated with each statechart. It enqueues all events arriving at a statechart. The UML does not specify exactly in which order events are removed, in order to leave room for, e.g. priority-based strategies. For our formalisation we assume that events are enqueued immediately after they were sent in the order in which they were sent from each statechart. They are dequeued in the same order as they were enqueued. In addition, events can be prioritised, meaning that they will be enqueued at the head of the queue. If several prioritised events are enqueued we do not make assumptions about their order.

Events are processed one at a time, i.e. an event can only be processed if the processing of the previous one has completed entirely (*Run-to-Completion Semantics*). This includes all actions generated during the transition. If an action is synchronous, then the transition is not completed until invoked objects complete their own run-to-completion transitions.

An event can only trigger one transition. If several transitions are possible, one is arbitrarily selected except for hierarchical states where a precedence mechanism is defined. If there is no corresponding transition for an event it is ignored but not stored. In order to store an event, a state can declare an event as *deferred*. Whenever the event occurs in that state, it is put back in the queue.

An ignored event can be interpreted as normal behaviour or as illegal behaviour. Often, the latter semantics is associated with protocol statecharts. Here, a synchronous method call event which cannot be dispatched is considered illegal (unless it is deferred). We will adopt this interpretation here.

If the statechart is attached to an active object, e.g. a thread, the events for that object can be processed by its thread, potentially concurrent with event processing in other threads. Events for passive objects are processed by a system thread. This is a semantic variation point of the UML. Having a queue per active object simplifies avoidance of concurrency conflicts during event processing.

Although we will use hierarchical statecharts and a history mechanism, we do not rely on a specific transition selection mechanism because we use only one level of nesting and no conflicting transitions.

### Event Types

How event processing affects the sender of an event depends on the kind of event. There are four kinds of events.

- An *asynchronous signal event* denotes the reception of an asynchronous signal. The sender does not have to wait for the event to be processed. This event types can have parameters.

- A *synchronous call event* denotes the reception of a request to synchronously invoke an operation. The sender has to wait until the call event is dequeued and the transition is completed. This event types can have parameters.

- A *change event* is an event that occurs when a boolean expression becomes true.

- A *time event* occurs after a specified elapse of time.

In the presence of event-queues per active object, synchronous events can cause a deadlock in the following way: A first thread sends a call event to a second thread. The event is queued in the queue of the second thread. The first thread becomes blocked waiting for the call to be handled. While the call event arrives, the second thread processes an earlier event and sends a call event to the first thread. This event is queued and cannot be handled because the first thread is already blocked. The second thread also becomes blocked.

At present, there is no convincing solution for this problem. The semantics of Rhapsody statecharts [HK04] specifies that call events are sent to the receiver directly instead of queuing them. It is also specified that the receiver serialises incoming concurrent calls and that they can only be handled when the thread does not handle other events. This blocks the callers in a similar way as in the UML semantics. Deadlocks generated by these semantics are considered as design errors. Both approaches, UML and Rhapsody enforce a kind of rendezvous concept on thread communication in statecharts, i.e. both threads have to be available to handle a call event from one thread.

Both proposed statechart semantics for synchronous events differ from what is actually happening during a method call in a programming language like Java. Therefore, they are not suitable for documenting the exact behaviour of programming languages like Java.

**Guards and Actions**

A guard is a boolean expression which controls whether a transition for an event will fire or not. A transition can have an associated action which executes when the transition fires. It must execute completely before the transition is completed. This includes synchronous method calls and synchronous call events. Both, guard and action, can access parameters of the event and attributes, links, and operations of the object which is associated with the statechart. It is not specified whether a transition is atomic with respect to guard and action, i.e. whether guard and action are carried out as an atomic transaction on the respective data structures. Therefore, we will not assume atomicity.

Actions can send call events and signal events to other objects. The target has to be specified with the event itself. An event generated during a transition is put in the queue of the target. If no target is specified, the event is sent to the object which has issued the event, i.e. the implicit target is "this". UML proposes a framework for action semantics and supports the design a suitable action language.

### 6.3.2 Classification of Thread Lifecycle Methods

Before we can design the states and transitions of the Java thread lifecycle, we will discuss specific characteristics of methods of the thread lifecycle whose pre- and postconditions and intermediate execution states we want to model with states.

Method calls are are synchronous by definition. They can be further characterised according to the fact whether they are static or non-static. There are also some system events which trigger a return from a method call instead of an ordinary return. Both, calls and system events, are potential events on transitions.

The potential candidates for events can be further characterised as follows.

1. A thread can invoke *non-static methods* of a thread object (including itself). These calls are executed in the control flow of the calling thread. When a method returns, the state of the calling thread might have changed and the state of the called thread, too.

   Methods new(), start(), interrupt() only change the state of the called thread object. Method new() creates a thread object. Method start() creates the corresponding control flow, the thread object changes its state to running. Methods new() and start() can only be invoked once

for one thread object. Method interrupt() can be invoked concurrently from concurrent threads. It changes the thread's interrupt flag to interrupted.

Method join() and wait() only change the state of the calling thread. Also the call of a synchronized-method or block only change the state of the calling thread.

Methods notify/All() do not change the state of calling or called thread. The called thread is used to signal other waiting threads whose state is potentially changed.

Methods join(), wait(), notify/All() and a synchronized-method or block can be invoked concurrently. Note however, that wait(), notify/All() also require a synchronized. Therefore, all methods except join() are in fact mutual exclusive when invoked on the same thread object.

From the point of view of the thread object, on which these methods can be called, these calls are also referred to as *external* calls.

The behaviour of a thread on which wait(), notify/All() and a synchronized-method or block is called is the same as of a passive Java object. They will be explained in more depth in item 3. In the following we will not deal with these methods specifically for thread objects, because the behaviour of thread objects regarding these methods can be derived from the behaviour of objects. This is admissible, because these methods do not interfere with the thread-specific behaviour of thread objects.

All other methods which can be invoked concurrently on a thread object have no effect on the lifecycle nor on synchronisation. In order not to make the statechart more complex than necessary, they will not be considered here.

2. A thread can invoke *static methods* of class Thread. These methods have an effect not on the state of the class but on the thread itself who is calling them.

   A thread can invoke one of these methods concurrently with invocation of its own non-static methods by other threads.

   Either the state of a thread changes upon return of the method, e.g. for interrupted(), or the state of a thread changes *during* the execution of the method, and the state is reset upon return. For example, sleep() causes a thread to change to *sleeping* until it returns from the call. While a thread is executing method sleep() it can react differently to

concurrent invocations from other threads. If method interrupt() is called on a sleeping thread, the thread returns immediately.

3. The *thread* under consideration can invoke any *non-static method* on any other Java object.

   Calling methods wait() or the keyword synchronized has an effect on the calling thread but not on the Java object. Method notify/All() has no effect on caller or callee but on other threads waiting in the queue of the object.

   In the case of wait() or the keyword synchronized, the thread changes its state *during* the call and it changes its state again upon return. Again, these states are considered important.

   It is important to distinguish the state waiting because a thread can return differently from this state, via timeout, via notification or via interrupt(). It is important to distinguish the blocking for a synchronized-lock because this is the source for failures.

4. A thread changes state due to *system events* such as notifications or unblocking or timeouts.

   In a state where a system event occurs the thread is engaged in calling specific methods but cannot proceed from these methods unless other threads make certain calls.

   Hence, these events are only concurrent with respect to calls from other threads. They can never be concurrent with a method call from the thread in which they occur because that thread is in a state where it cannot make any more calls.

A statechart modelling the behaviour of one thread has to model how it reacts to calls issued by its own control flow, how it reacts to calls from concurrent thread on the corresponding thread object, and how it reacts to system events for its own control flow.

### 6.3.3 Principles for Designing States and Transitions

In this section we will derive a set of design principles for designing state and transitions from the above characteristics of methods for thread synchronisation. These principles will be applied when we identify how the behaviour of a method is described in terms of states and transitions.

The principles are chosen such that they support our main goal, which is to use the statechart model to specify failures and potentials. Therefore,

the statechart must model not only the behaviour of the class Thread with respect to synchronisation but it must also model instance-specific behaviour such as a *dependency* between two threads.

The goal of the statechart is to describe how a thread *reacts* to method calls which it issues itself and which it receives from other threads. We do not model, how the method calls issued by a thread are generated. We make the assumption, that, when a thread of control works through the code, the corresponding events will be generated.

### Designing Events

We have already pointed out that there are some key differences between Java semantics on the one hand and UML statechart semantics on the other hand. When attaching a statechart to a thread, the events sent to the thread are queued. Because the thread is considered an active object it can process the queue using its own thread of control. If another thread synchronously calls the thread and if that call maps to a call event that call (event) is queued. To the contrary, in Java, the call would be executed immediately in the thread of control of the calling thread. Also semantics which do not queue synchronous calls are not equivalent because they make multiple concurrent calls mutual exclusive. It is not the goal of this thesis to provide a general solution, either as an extension or a mapping algorithm, for this problem. Instead, we will try to determine the most appropriate solution for our Java synchronisation model.

Because of the mismatch we will not rely on the use of synchronous events. Instead we consider *asynchronous* events as an alternative. There are several reasons why this is feasible.

First of all, we do not want to model the behaviour of arbitrary method calls. Next, for some method calls we want to model intermediate execution states which would not have been possible with synchronous events but requires to split method calls in asynchronous *enter* and *exit* events. Also, we have noted that there is more than one way to return from these states. This can easily be modelled by several exit events.

For those method calls, which we do not split in this way, we consider a single asynchronous event as appropriate. The nature of these method calls is such that they will always return immediately and that they have no return value. Therefore, they are very similar to asynchronous events, such as the method interrupt(), only that Java does not support asynchronous calls.

The justification will be made clearer for the different cases in the following. When we model calls with asynchronous events we also have to consider that these events are queued and we have to ensure, that the queuing does

not yield a semantics which is not feasible in Java.

Using asynchronous events we *loose* of course *synchronity*. We cannot specify in the statechart that a certain exit event has to follow a method enter event. However, we gain expressiveness with respect to those states we are interested in for failure detection. There is no way to map a synchronous call in two asynchronous events by means of a statechart with the above semantics. We have to assume that the corresponding entry and exits events are generated.

### Designing the Event-Queue

Following UML, we assume that there is a queue for each statechart, i.e. for each object for which a statechart exists. We do not use priorities and we do not use deferred events.

Each thread receives events due to its own calls and events from other threads due to their calls. Both are queued in the same queue. Therefore, we have an interleaving semantics for the concurrency semantics of Java which allow both, interleaving and real parallelism. For the calls which a thread executes in its own control flow, we assume that the system generates them. We assume, that at most one such event is generated and that at most one such event is in the queue at a time. This is correct, as the thread will be able to execute only one piece of code after the other. We do not model how these events are generated.

### Designing States

Designing states involves three key issues,

- identifying intermediate states, i.e. states describing the execution of a method call,
- factoring out commonalities, i.e. hierarchical states and common transitions, and
- designing how instance-specific information per state has to be kept.

The last item takes us to the issue of the design of an action language.

### Action Language

We have already said several times, that in addition to the state of a thread there is also state-specific information which can not be captured by a statechart. This additional information is needed for two purposes:

- It constrains the executions of method calls.
- It can be used in expressing state-based dependencies between threads which is needed for failures.

An action language provides means to deal with such additional information. For the first case, we need to be able to use that information in guards, and therefore actions have to perform suitable side effects. For the second case, we need to be able to query that information for the entire system state and we need to be able to navigate it arbitrarily.

We assume that for each statechart (instance) a list of variables can be specified. We use variables of type $N$ (integer rsp. natural numbers including 0) and of type set of $N$, also denoted as $\mathcal{P}(N)$. Each statechart (instance) has its own data space and cannot access the data of another statechart. We assume that the action language can perform assignments on integers using simple arithmetics, and assignments on set types using standard set operations.

We use the set N to depict unique identifiers of threads and objects. Similarly, $\mathcal{P}(N)$ is used to depict sets of unique identifiers. This is based on the assumption, that the unique identifiers are either integers or can be mapped uniquely to integers.

### Direct Mapping of Method Calls to Asynchronous Events

Here we describe the one-to-one mapping of a synchronous call to an asynchronous event.

This applies to the non-static methods new(), start(), interrupt() (see item 1.) and notify/All() (see item 3.). It applies to the static method interrupted() (see item 2.). Here we describe the effect of using an asynchronous event and we conclude by defining the corresponding event for the resulting statechart.

- The external calls new() and start() both return without problem. Concerning the thread on which start is called, the queuing should not have an effect as this call does not compete with others by nature. The queuing has the beneficial effect, that concurrent calls assuming that the thread was started can not be made too early.

  The two events introduced are NEW and START. They have no parameters.

- Methods notify/All() are mapped to single asynchronous events. They have no influence on the caller and no return value. The method notify/All() is mutual exclusive because it requires synchronized. Therefore, the queue is not a problem.

The two events defined are NOTIFY(ob) and NOTIFYALL(ob). They have a parameter for the target object.

- For interrupt() we have chosen to handle it as an asynchronous event to avoid deadlocking. This is admissible, as the method interrupt() cannot block because it is not synchronized and also it does not return a value. Therefore, the interrupting thread usually does not depend on the synchronous handling. As a result of using an asynchronous event, the calling thread is not waiting longer than necessary because it is not waiting at all.

  Regarding the queuing itself, we argue that each interrupt changes the control flow state of the called thread and therefore has to be carried out mutually exclusive with other events handled.

  The asynchronous event defined is IRPT.

- Also method interrupted() is mapped to an asynchronous event. This is problematic, as it has a return value. In addition, the method changes the state by clearing the interrupt flag. We can accept to replace the method with an event, if we argue that the return has no effect on synchronisation and we ignore its value.

  The asynchronous event defined is INTERRUPTED.

Table 6.1 gives an overview of the mapping of method calls to asynchronous events, which are depicted in capitalised letters. Parameters are omitted in the table.

### Splitting Method Calls in Asynchronous Events

Here we describe the mapping of a synchronous call to a combination of an asynchronous entry event, modelling the entering of the method call, and at least one asynchronous exit event.

This applies to the non-static methods join() (see item 1.) and wait() (see item 3.), to non-static methods with a keyword synchronized and to blocks with the keyword synchronized (both item 3.), and to the static method sleep() (see item 2.).

- The call to join() is split, because it can be involved in failures and it can return via interrupt or when the joined thread is finished or via timeout.

  The entry event is JOIN(t) or the timed variant JOIN(t,d). The parameter t is the target of the join, d is the duration of the timeout. The events defined for return are EXIT, IRPT or a time event after d.

| Mapping | Java methods/ statements | Event name |
|---|---|---|
| Single asynchronous events | new(), start()<br>interrupt(), interrupted()<br>notify/All() | NEW, START<br>IRPT, INTERRUPTED<br>NOTIFY, NOTIFYALL |
| Asynchronous entry events | join, join(d)<br>sleep()<br>wait(), wait(d) | JOIN<br>SLEEP<br>WAIT |
| Asynchronous entry and exit events | synchronized | LOCK, UNLOCK |
| Asynchronous system events | timeout after timed wait(d), join(d), sleep(d)<br>notification of a thread<br>unblocking of a thread after lock release<br>return from run | after d<br>NOTIFICATION<br>FREE<br>RUN_EXIT |

Table 6.1: Mapping Java Behaviour to Statechart Events

- The call to sleep() is split analogously, although never involved in failures. It can return via timeout or via interrupt.

  The defined entry event is SLEEP(d) with a parameter for the duration of the sleep. Returns are via a time event after d or IRPT.

- The call to wait() is split. Waiting can be involved in failures and has several possible ways of returning, via interrupt, timeout or notification. As with notify/All(), the queue is not a problem.

  The entry event is WAIT(ob) or WAIT(ob, d). The parameter ob is the object on which the wait is called, d is the duration of the timed variant. Return is via events IRPT, after d, or NOTIFICATION.

- Intermediate states of calls to synchronized are involved in failures. However, synchronized deserves a more complex treatment than a simple splitting. Firstly, the call itself is split in an explicit locking and unlocking, because the release of a lock has an effect on other threads which has to be modelled. Secondly, locking has to be differentiated because it can involve an intermediate state blocking before the thread obtains a lock and continues to run.

  Calls to synchronized are split into the events LOCK(ob) and UNLOCK(ob) with the parameter ob for the object on which they are called. When a thread tries to obtain a lock via LOCK(ob), it has to be signaled by the system if it can obtain the lock via the event FREE(ob).

  Because synchronized is mutual exclusive by definition, the queuing of the corresponding events is not more restrictive than Java.

The resulting events are depicted in Table 6.1.

### Direct Mapping of System Events

The system events (see item 4.) map directly to *asynchronous events*. We will not model the system and therefore we will not model how system events are generated. The following events have already partly been introduced above.

- We model timeouts occurring after certain method calls with a *time event*, denoted as after time.

- When the target thread of a join() is exiting, an event EXIT is generated.

- When a thread calls notify() or notifyAll(), the system determines which threads have to be notified and sends an event NOTIFICATION.

- After a synchronized-lock is released, one blocking thread has to be signaled via a FREE event.
- Finally, a thread returns from run() via RUN_EXIT.

See the resulting events in Table 6.1.

### 6.3.4 The Resulting Statechart

The main goal is to show how the thread changes state due to calls it makes itself and due to incoming calls from other threads. We have already described how we define events for each method, statement, and system event from the Java thread lifecycle.

Because of the complexity of the thread lifecycle we will introduce the resulting statechart step by step. Each statechart models the behaviour of *one* thread. In the following we will refer to this thread as the *observed* thread in order to distinguish it from other threads it interacts with.

We first describe, how the thread reacts to calls from other threads. Then we describe, how the thread reacts to self calls, i.e. we show how self calls change the control flow state of the calling thread.

Note again that we omit all methods which by their nature have no influence on the lifecycle but provide handling of independent thread properties such as methods to set/get the thread name, group, priority, or interrupt flag.

The resulting statechart will describe the part of the behaviour of a Java thread related to the thread lifecylce, especially including synchronisation. Therefore, the statechart can be seen as providing operational semantics for a set of Java method calls with respect to a limited set thread states. The statechart describes an interpreter which takes a thread state, an event, and the state of the queue as input, and generates a (new) state as an output.

The following statecharts were created with the UML CASE tool Together [Tog].

#### Reacting to External Calls

After its instantiation via NEW the thread is in the state *created*. After the call of START it is in the state *runnable* (see Fig. 6.2). We have chosen this label to express that it is under the control of the scheduler whether the thread is actually running or not, although it has no effects on the semantics presented here.

While start() returns immediately, after starting the thread by calling run(), run() will not return immediately. The entering of run() is therefore

modelled as the transition to a new state. We depict the return with the event RUN_EXIT which triggers the transition to state *terminated*. This event covers ordinary return from method run() as well as exceptional return with no handler found. For our purpose, it is not necessary to differentiate between ordinary and exceptional return. When first entering *runnable*, the observed thread enters *cleared* because its interrupt flag is not set.

In a state like *runnable* and also in other states to come, we have to provide transitions with events for external calls. Absence of an event means that such a call is illegal. The event NEW does not make sense since new() can be called only once for one instance. Method start() could be called again, and this is illegal.

Any thread can send an IRPT. The called thread has to change from *cleared* to *interrupted*. When IRPT is received in state *interrupted*, the thread remains in this state. Other effects of IRPT will be explained when sleeping, joining, waiting, and locking is described.

External calls to synchronized and wait(), and notify/All(), and join() will not be considered here, as they do not change the state of the called thread. Instead, we will consider these methods, when the thread itself is calling them on other (thread) objects, and we will model the effect which they have on the calling thread.

### Reacting to interrupted()

With a static self call interrupted(), the observed thread can change from *interrupted* or *cleared* to *cleared*, depicted with the event INTERRUPTED (see Fig. 6.2).

### Reacting to sleep()

When the static method sleep(d) is called by the observed thread, the observed thread reacts to it by sleeping for the specified amount of time, then the call returns. The call also returns when another thread calls interrupt() or when the interrupt flag has already been set.

With the event SLEEP(d) the thread enters *sleeping* if it was in *cleared*. It returns via after d or IRPT (see Fig. 6.2). If the thread was in *interrupted*, it changes immediately to *cleared*.

We do not distinguish between ordinary return and exceptional return. If a thread returns via an interrupt, the corresponding exception will be caught by a handler at some point upward in the callstack, or the program exits. If it is caught by a handler, the program will continue normally. Therefore, we do not introduce a new substate of *runnable* but show the different events

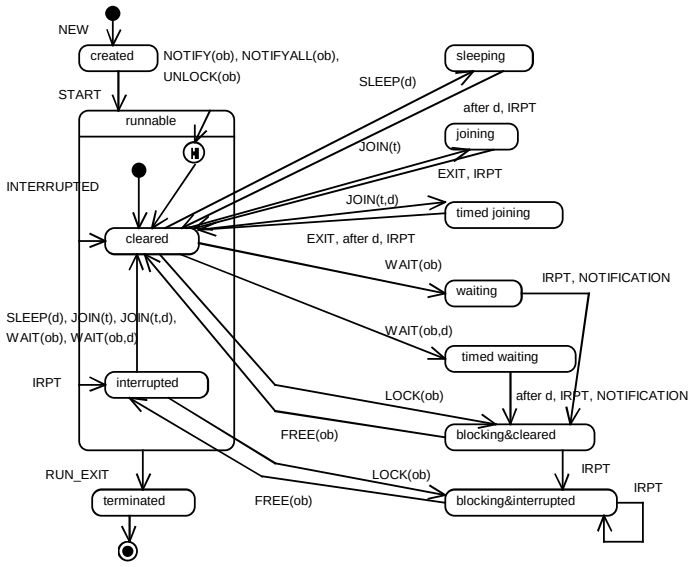


Figure 6.2: Thread Lifecycle Statechart

on the transitions. In any case the thread will be in the state *cleared* when it returns (see Fig. 6.2).

### Reacting to join()

The non-static method join() can be called by an observed thread on another thread but it has an effect only on the calling thread, i.e. the observed thread. We distinguish the states states *joining* and *timed joining* (see 6.2) because the thread can return differently from them.

The observed thread enters *joining* by JOIN(t) where t stands for the target. On termination of the target thread the event EXIT is sent to each joining thread. Also, the observed thread can return if an IRPT is sent. The timed variant is depicted by JOIN(t, d). From this call the observed thread can return via EXIT, by IRPT and by timeout after d.

### Reacting to wait() and notify/notifyAll()

Similar to joining, we distinguish the states *waiting* and *timed waiting* which are entered via WAIT(ob) or WAIT(ob, d) from state *cleared* (see Fig. 6.2).

From both states, the observed thread can return via IRPT or via a NOTIFICATION generated by the system when another thread calls notify() or notifyAll() on the corresponding object. From *timed waiting*, the thread can also return via timeout after d.

In Fig. 6.2 it is shown, that the thread then changes to *blocked&cleared* because it first has to re-acquire the lock which it released upon entering waiting. Locking will be explained in the next paragraph.

One NOTIFICATION can cause the leaving of *waiting/timed waiting*. If there are more notifications in the queue than threads, they cannot be matched and therefore, they have no further effects. This corresponds to the Java semantics of notification.

If the observed thread receives WAIT(o) or WAIT(o,d) in state *interrupted*, the wait returns immediately with an exception and the interrupt flag is cleared, and therefore the thread enters *cleared*.

A thread can issue notifications only in state *runnable*. It reacts to them by return to the previous state, which is modelled using the *history* state.

### Reacting to synchronized-Calls

A synchronized-method or synchronized-block claims a lock for the specified object in order to proceed. It proceeds if the lock is granted or blocks if the lock is possessed by another thread. We have to distinguish, whether the thread was in state *cleared* or *interrupted* because it has to return to the previous state when the lock is granted. We cannot use the history mechanism because the thread can receive an IRPT while blocking on the lock.

In a simple solution (see Fig. 6.2), the thread changes to state *blocking&cleared* when it receives an event LOCK(ob) in state *cleared*. This state has two meanings. It is a state where the thread tests if the object is free. If the object is free, an event FREE(ob) is sent, and the thread can leave the state immediately. The state also has the meaning, that the thread is blocked because the object is already locked when it tries to acquire it. Then it also leaves via an event FREE(ob) which is sent when another thread releases the lock and when the observed thread is the one granted the lock.

Analogously, when in state *interrupted* the thread changes to *blocking&interrupted*. Note that the thread can change from *blocking&cleared* to *blocking&interrupted* due to IRPT.

Locks are released with the event UNLOCK(ob) in state *runnable*. After this event, the thread returns to the previously hold state, indicated by the *history* state.

The solution presented so far does not yet handle instance-specific information such as which locks are held and how often each lock was entered, i.e. re-entrance. Therefore, the solution does not support to test whether a lock is held when calling wait() or notify/All(), modelled by the corresponding events WAIT, NOTIFY, NOTIFYALL. It also does not distinguish in state

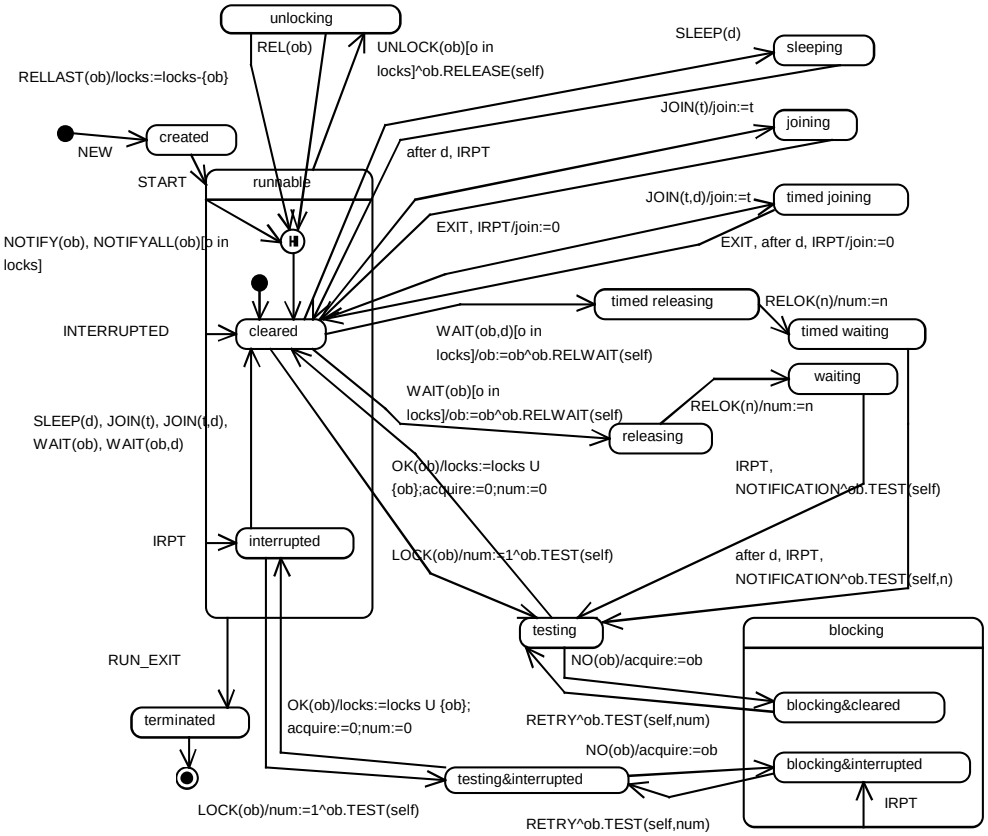


Figure 6.3: Thread Lifecycle Statechart with Guards and Actions

*runnable* whether a thread is running with locks or not. Also, the molding of testing for a free lock and the actual blocking into one state is not really satisfactory.

Based on the statechart in Fig. 6.2 we cannot yet specify failures or potentials because we cannot yet express which locks a thread is holding and at which threads or objects it is waiting. The next paragraph describes, how the missing information can be added. We will also provide a more appropriate solution for describing blocking and locking.

### Adding Instance-Specific Information

The thread statechart is augmented by a definition of data fields holding the instance-specific information needed to express dependencies involved in failures and potentials.

We first describe the straightforward extension to capture dependencies involved in joining. We need to keep track of the target of a thread in state

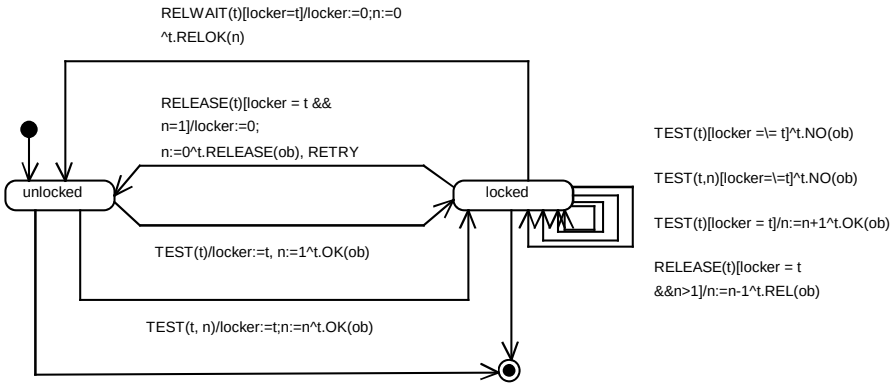


Figure 6.4: Object Synchronisation Behaviour Statechart

*joining* or *timed joining*. Therefore, the thread statechart has a field `join : N` of type integer (also depicted in the field declaration below). On entering the above states, the target thread is stored with the action join:=t. Target t is obtained from the event parameter. When returning, the target is cleared with join:=0. This extension is depicted in the statechart in Fig. 6.3.

The following code snippet declares the data fields used by a thread statechart. Note that for each thread instance, there is a separate instance of all these fields. The thread identifier is stored in the field `self`. The remaining fields will be explained in the following.

```
// data fields for thread statechart

self : N      // unique thread identifier
join : N      // target thread of a join call
ob : N        // target object of a wait call
acquire : N   // target object of a blocking synchronized call
locks : P(N)  // set of locks hold by the thread
num : N       // number of locks to be acquired during re-entrance
```

Next we deal with blocking and locking. Here we have to keep track of the following details:

- We have to track the state of a lock object, either locked or unlocked.
- We have to track for each lock the owner and vice versa.
- We have to count re-entrance of locks.

That means, for each LOCK(ob) event we have to check whether the object is already locked by this thread. Depending on the result we either change to

state *runnable* or to *blocking&cleared*. The check must be mutually exclusive with checks of other concurrent threads for the same lock object. As threads do not share data, this can only be modelled through a statechart for lock objects. Also, the checking and the obtaining of a free lock must be atomic. As this involves communication with the object's statechart, this cannot be modelled with a single transition in the thread statechart. In addition, the object statechart must guarantee that the effect of the communication guarantees atomicity.

Therefore, we propose a solution, where a thread first tries to grab a lock and in a second step checks whether it was successful or not. Depending on the result it enters different states. This is implemented using an additional state *testing* (see Fig. 6.3). On the transition from *cleared* to *testing*, the thread sends an event TEST to the object it wants to lock and then waits for the answer of the object. The thread can receive a NO meaning that the object was already locked. Then it enters *blocking&cleared* and stores the acquired object in the field `acquire : N` using acquire:=ob. The thread can receive an OK meaning that is has locked the object successfully. Then it adds the lock identifier to the set of its locked objects hold in the field `locks : P(N)` with the action locks:=locks $\cup${ob}. It also deletes the acquired object with acquire:=0. Note that re-entrance is counted in the object itself. The thread can leave the *blocking* when it receives a RETRY generated by the lock object. It then sends again a TEST.

Again we require a separate state *testing&interrupted* for a thread which has been interrupted. The transitions are the same and have already been explained.

The thread releases locks via the event UNLOCK. This event can now be guarded using [o in locks]. (Note that in the diagrams generated with the CASE tool Together the symbol $\in$ is depicted as "in"). Also, here communication with the object statechart is required to indicate the release of the lock using event RELEASE(ob). The object statechart checks whether the lock is finally releases. If so, the object sends an event RELALL event and the thread removes the lock from its list using locks:=locks - {ob}. Otherwise, the objects sends a REL event. The thread enters via the history state to return to its state prior to the unlocking.

Note that for a complete model it would be necessary to introduce for each of the states *unlocking*, *testing*, and *testing&interrupted* an additional transition for receiving IPRT which could happen while the thread is in these states. We have omit this in order to keep the statechart readable. Alternatively, one could assume that the events from the communication with the object are prioritised. For convenience we have grouped all states related to blocking into a super state *blocking*. This avoids having to distinguish states

when describing failures involving blocking.

The lock object behaviour relevant for the above described communication with the thread statechart is depicted by the statechart in Fig. 6.4. It is associated with the following data.

```
// data fields for object statechart

locker : N     // thread currently locking the object
n : N          // re-entrance counter, i.e. number of times thread has locked the object
```

The object has a field for storing the thread which locks the object and it has a field for counting the re-entrance of that thread. Threads blocking at the object do not have to be stored because the model does not cover the generation of notifications. Instead it assumes that NOTIFICATION events are generated by the system.

The statechart from Fig. 6.4 distinguishes states *unlocked* and *locked*. In *unlocked*, the object can grant access to the first thread whose TEST event it receives. The object handles TEST events mutually exclusive and therefore the object is only locked by one thread. The object stores this threads and sends an OK to it. When the thread tries a re-entrance, access is granted. Other threads are sent a NO. When the thread releases the lock, the objects checks for final release and sends messages correspondingly. Note that the same behaviour as for the object also applies to thread objects as they can serve as locks, too. Here we have not modelled this aspect of threads as this would make the resulting statechart too complicated. The missing behaviour, as we have argued before, is completely orthogonal to the thread behaviour specified so far.

The extensions also requires to make changes to notifications and waiting. We can use the information to guard the events WAIT, NOTIFY, and NOTIFYALL with [o in locks]. On entry, the lock has to be completely released by sending a RELWAIT event to the object. The object has to acknowledge the release by a RELOK(n) providing the re-entrance number.

In addition, states *waiting* and *timed waiting* are linked with the new state *testing* (see Fig. 6.3). It is entered via events NOTIFICATION, IRPT, or after d. In addition, the event TEST is sent to the acquired object. Therefore, the object reference has to be store when entering *waiting* or *timed waiting*. This is done by an assignment to an additional field ob:=ob.

Method wait() or the timed variant wait(d) can only be called when the corresponding synchronized lock is hold by the thread, this is already checked when the call is generated. We have to keep track of the fact that the thread has to release the lock completely (and regain the lock after returning from wait). When a thread releases a lock on entering wait(), the object statechart sends the re-entrance count to that thread and releases its lock completely.

Also from state *releasing* we would need a transition for an IRPT which we also omit in in analogy to the states *unlocking*, *testing* etc. When a thread returns from *waiting* it claims multiple access to the lock. Either they are granted or the thread enters *blocking&cleared*. From there, a thread returns by claiming a lock a fixed number of times. For a thread which claim the lock for the first time, the number is set to 1. Leaving *waiting* we can reuse the existing state *testing*. An event TEST with an additional parameter for the re-entrance number required is sent. Only if the corresponding object is unlocked, the thread enters *runnable* with the action that it acquires the lock as many times a held before. If the corresponding object is locked, the thread changes to *blocked&cleared*. A thread changes via after a time event to *blocking&cleared* if the corresponding object is locked. If unlocked the thread changes to *runnable* with the same actions taken as for notification.

Note that also other solution are conceivable. For modelling re-entrance, one could also maintain a re-entrance counter in the thread data, one for each lock in the set of locks. Here, our goal was to keep the data types as simple as possible.

#### Limitations of UML Statecharts

At present, the formal semantics of UML statecharts have not yet been completely defined. Of cause, there will not only be one formal semantics, but different semantics will concretise some of the extension points of the UML [HK04].

We do not want to contribute to the area of defining formal semantics of UML statecharts. Instead we are interested in using visual formalisms for describing models useful for understanding a certain domain. Nevertheless, we have tried to keep the statechart simple, especially with respect to the action languages, so that it would be straightforward to translate it into a simpler state-based formalism such as Labelled Transition Systems as used in [SHS03] or Kripke structures [LMM99] for which formal semantics exist. Also, the hierarchical states and the history mechanism could be easily transformed into non-hierarchical states.

### 6.3.5 System State and History

The statechart provides the lifecycle for one thread. In order to formally define failures, we have to formalise the states of all threads involved, as a failure has been defined as a system state.

We intend to simulate a system of concurrent threads by instantiating the statechart for each thread of the system.

The set of all objects existing throughout the whole system lifetime is denoted by the set of natural numbers $N$. Each number is used only once. This set includes also thread objects. The set of all control flow states of threads is denoted by $C$.

$C = \{created, runnable, cleared, interrupted, terminated, sleeping,$

$joining, timedjoining, waiting, timedwaiting, blocking,$

$blocking\&cleared, blocking\&interrupted, unlocking, releasing,$

$timedreleasing, testing, testing\&interrupted\}$

We do not need to formalise the object states for the description of failures and potentials because it is sufficient to refer to objects through the instance-specific information of each thread. We define the system state as a mapping of the set of objects to the set of control flow states.

**Definition 6.1 (System State)** *A system* state *is a partial function*

$sys : N \rightarrow C \cup \{\perp\}$ *with*

$n \mapsto c \in C$ *, if n is a thread and the thread is alive*

$n \mapsto \{\perp\}$ *, if n is a thread that is not alive.*

Note that only objects that are threads are mapped to thread states. The bottom symbol denotes that a thread does not have a system state, because it is not yet created or already deleted. The state of a thread also includes the instance-specific values which are stored in the Java data structure. For each thread state there is a snapshot of this data. We assume that the data fields can be accessed in a given system state using the field names.

The *execution history* of a system is a sequence of events generated by the statecharts of all threads of the system.The set of possible event occurrences is denoted by $E$. Here, we need occurrences and not types, because we need access to the actual parameters of events. Set $E$ is infinite because of the infinite possibilities of actual parameters. Event occurrences are derived from the event types given in Table 6.1 and by the additional event types defined while constructing the statechart in Fig. 6.3. The event types are instantiated with potential parameter values. All parameters are natural

numbers. Depending on the signature of the event defined in our statechart model, a number may identify an instance or a timeout.

$$E := \{IRPT, LOCK(1), UNLOCK(1), ...\}$$

The execution history also keeps track of the threads which generated the events. An event in the history is therefore a tuple $(e, n)$ where $e \in E$ is an event occurrence and $n \in N$ represents the reference to a thread instance.

The chosen definition of an execution history provides a total order. That means, concurrent events are eventually serialised. The result is an order with an interleaving semantics. This is sufficient for our purpose. The history is a mapping from a point in time to an event in the history. The time axis is represented as a set of natural number, denoted as $I$ for index.

**Definition 6.2 (History)** *The* history *is a partial function*

$his : I \rightarrow (E \times N)$ *with*

$i \mapsto (e, n)$*, also denoted as* $\binom{e}{n}$*, if event* $(e, n)$ *happens at the point in time represented by* $i$*.*

Here we have defined a history simply as a *trace* of events but not as a sequence of states. For our purpose, this will be sufficient. The above mapping is partial because a program might generate only a finite number of events.

Note that the term trace as used here refers to a different kind of trace than in the motivation. The term used in the motivation stands for a trace generated from a running Java program where an event has a different format from here consisting of the method name, parameter names and values and a lot more. Therefore, it should not be confused with the execution history based on statechart events as defined here. The trace we have defined here can be seen as an abstract model of the execution of a Java program and thus an abstraction of the trace as defined in the motivation.

In Sect. 6.4, the above formalisation will be used to specify failures and potentials in a declarative way. We assume that for each thread or object created, an instance of the corresponding statechart exists. We further assume that the data fields associated with each statechart are initialised. The `self` field of the thread-specific data has to be set to the identifier (natural number) of the corresponding thread. All other fields of type $N$ should be set to zero. Fields of type $\mathcal{P}(N)$ are assigned with empty sets.

## 6.4 Formalising Concurrent Java Liveness Failures and Potentials

Failures which involve dependencies will be formalised using the system state formalisation (see Definition 6.1) because dependencies can only be expressed with these kind of formalisation.

Failures not involving dependencies and potentials are more conveniently formalised by the notion of an execution history (see Definition 6.2).

### 6.4.1 Formal Description of Failures

We formalise the failures which have been informally introduced in Chapter 5. There, typical failure scenarios were depicted by a sequence diagram. Here, we describe a formalisation which aims at matching all possible scenarios of one kind of failure in a running Java program.

In the following we do not consider the timeout variant of events and of states, i.e. the state from which one can return with timeout, because including them would make the specifications less understandable. It is possible to replace the variants with no timeout with a timed variant in the following.

There are two failures which are not formalised. Indefinite blocking on synchronized (F3) is not covered because there is no pattern, besides detecting a blocked thread, which allows to detect it. Detecting it depends only on heuristic using a timeout after which the blocking is assumed to be a failure. Missing notification (F5) cannot be covered because it is related to source code.

#### Deadlock (F1, F2)

Here we give a deadlock definition for two threads. The definition can be extended to match three or four threads, etc. Note that the definition covers also the wait-induced deadlock.

$$\exists t_i \in N \exists t_j \in N : t_i \neq t_j \wedge sys(t_i) = blocking \wedge sys(t_j) = blocking \\ \wedge t_i.acquire \in t_j.locks \wedge t_j.acquire \in t_i.locks \quad (6.1)$$

#### Missed Notification (F4)

This failure involves one thread not being able to make progress because it is in state waiting. This thread is dependent on other threads because it is waiting for a notification signal. Assuming single notifications, a missed

notification is a wait() not followed by a notify(). Either, there has been only a notify() before but no other wait(), or there have been two subsequent occurences of notify() with no wait() in between.

Because the failure deals with calling notify() on an object and not with a thread receiving the corresponding notification after the system has determined the threads to be notified, we describe a formula based on the event NOTIFY not on the event NOTIFICATION. We assume total ordering of the history trace with the smallest number being the first in the trace and hence the oldest event. We omit the type of the variables in the following formula to improve readability. Variables $s, t, u \in N$ etc. denote threads, $o, p \in N$ etc. denote objects, and $g, h, i \in I$ etc. denote indices from the time axis.

$$\exists m \exists s \exists o : his(m) = \begin{pmatrix} WAIT(o) \\ s \end{pmatrix} \wedge \forall t \neq s \forall n > m (\neg (his(n) = \begin{pmatrix} NOTIFY(o) \\ t \end{pmatrix}))$$

$$\wedge$$

$$(\neg (\exists l < m \exists t : his(l) = \begin{pmatrix} WAIT(o) \\ t \end{pmatrix}) \wedge \exists k < l \exists u \neq s : his(k) = \begin{pmatrix} NOTIFY(o) \\ u \end{pmatrix}$$

$$\vee$$

$$\begin{aligned} & (\exists k < m \exists t \neq s : his(k) = \begin{pmatrix} NOTIFY(o) \\ t \end{pmatrix} \\ & \wedge \forall u \neq s \forall l : k < l < m \Rightarrow \neg (his(l) = \begin{pmatrix} NOTIFY(o) \\ t \end{pmatrix}) \\ & \wedge \exists i < k \exists v \neq s : his(i) = \begin{pmatrix} NOTIFY(o) \\ v \end{pmatrix} \\ & \wedge \forall w \forall j : i < j < k \Rightarrow \neg (his(j) = \begin{pmatrix} NOTIFY(o) \\ w \end{pmatrix}) \\ & \wedge \neg (\exists x \exists h : i < h < k \Rightarrow his(h) = \begin{pmatrix} WAIT(o) \\ x \end{pmatrix}))) \end{aligned} \tag{6.2}$$

In order to identify that the thread has missed a notification we must deal with different cases. The first line of equation 6.2 states that there is a wait() not followed by a notify() from a different thread. The second line states the special case that this wait() is preceded by at least one notify() and

no other wait(). The rest of the lines matches the two closest occurences of notify() which happened before the wait(). The last line states that there has not been a wait() between the first (in execution order) of the two occurences of notify() and the observed wait().

**Nested Monitor Lockout (F6)**

$$\exists t_i \in N \exists t_j \in N : t_i \neq t_j \wedge sys(t_i) = waiting \wedge sys(t_j) = blocking \\ \wedge t_j.acquire \in t_i.locks \quad (6.3)$$

**Circular Join (F7)**

$$\exists t_i \in N \exists t_j \in N : t_i \neq t_j \wedge sys(t_i) = joining \wedge sys(t_j) = joining \\ \wedge t_i.join = t_j \wedge t_j.join = t_i \quad (6.4)$$

**Self Join (F8)**

$$\exists t \in N : sys(t) = joining \wedge t.join = t \quad (6.5)$$

**Join-induced Deadlock (F9)**

$$\exists t_i \in N \exists t_j \in N : t_i \neq t_j \wedge sys(t_i) = joining \wedge sys(t_j) = blocking \\ \wedge t_i.join = t_j \wedge t_j.acquire \in t_i.locks \quad (6.6)$$

### 6.4.2 Formal Description of Potentials

Here, we formalise the potentials described in chapter 5. Again, we omit the type of variables to improve readability. Variables $s, t, u \in N$ etc. denote threads, $o, p \in N$ etc. denote objects, and $g, h, i \in I$ etc. denote indices from the time axis.

### Deadlock Potential (P1)

Here we formalise the deadlock potential for two threads. The deadlock potential is captured by checking for existence of two locks which are locked in adverse orders in different threads. The last two lines check that there is no so-called gate lock, which is always used before the other locks and therefore avoids deadlocking.

$$\begin{aligned}\exists s \exists t \exists i \exists j \exists o \exists p : i < j \Rightarrow (his(i) = \begin{pmatrix} LOCK(o) \\ s \end{pmatrix} \wedge his(j) = \begin{pmatrix} LOCK(p) \\ s \end{pmatrix} \\ \wedge o \neq p \wedge s \neq t \\ \wedge \exists k \exists l : k < l \Rightarrow (his(k) = \begin{pmatrix} LOCK(p) \\ t \end{pmatrix} \wedge his(l) = \begin{pmatrix} LOCK(o) \\ t \end{pmatrix})) \\ \wedge \neg(\exists q \exists m \exists n : m \neq n \wedge his(m) = \begin{pmatrix} LOCK(q) \\ t \end{pmatrix} \\ \wedge his(n) = \begin{pmatrix} LOCK(q) \\ s \end{pmatrix} \wedge m < i \wedge n < k)\end{aligned} \tag{6.7}$$

### Unused Notification (P2)

An unused notification is one that follows another one. Between the two notifications there is no wait().

$$\begin{aligned}\exists u \exists s \exists o \exists n : his(n) = \begin{pmatrix} NOTIFY(o) \\ s \end{pmatrix} \\ \wedge \\ (\exists l < n \exists t : his(l) = \begin{pmatrix} NOTIFY(o) \\ t \end{pmatrix} \\ \wedge \forall u \forall m : l < m < n \Rightarrow \neg(his(m) = \begin{pmatrix} WAIT(o) \\ u \end{pmatrix}))\end{aligned} \tag{6.8}$$

**Potential for Join-induced Deadlock (P4)**

A thread acquires a lock and subsequently releases it. Later, the thread becomes the target of a join(). The joining thread holds the same lock while joining.

The following potential can be checked when a join() takes place. This is feasible because from the point of the time where the join() happens, the history is searched backwards for a claim of the lock by the other thread.

$$\begin{aligned}\exists t \exists o \exists k \exists m : his(k) = \begin{pmatrix} LOCK(o) \\ t \end{pmatrix} \wedge (\forall l > k : his(l) \neq \begin{pmatrix} UNLOCK(o) \\ t \end{pmatrix}) \\ \wedge m > k \wedge his(m) = \begin{pmatrix} JOIN(o) \\ t \end{pmatrix} \\ \wedge \exists s \exists n : n < k \wedge his(n) = \begin{pmatrix} LOCK(o) \\ s \end{pmatrix}\end{aligned} \tag{6.9}$$

Another risky situation is a join() with locks. This is not a potential in the previous sense but it is a risky situation. It can be described in a system state.

$$\exists t \exists o : sys(t) = joining \wedge o \in t.locks \tag{6.10}$$

## 6.5 Summary

In this chapter we have provided a statechart model to capture the behaviour of threads with respect to synchronisation on the type level. Based on this statechart we have captured individual failures and potentials more formally than in the previous chapter. This chapter marks the end of the presentation and analysis of the domain of concurrent Java liveness failures. The forthcoming chapters will describe support for automated detection of failures.

As a first step towards a model of thread synchronisation, we analysed the domain of Java concurrent liveness failures and potentials for their scope and their characteristics. The key results were:

- The scope of Java concurrent liveness failures is covered by Java *thread synchronisation*.

- The failures can be classified as specific combinations of *control flow states*, a definition we have introduced to identify the states in the thread lifecycle.
- These specific combinations of states can be further characterised by *dependencies* between threads in these states.

In order to cover control flow states and thread dependencies we have proposed a statechart-based model. This model has captured the thread synchronisation in a specific way. The key decisions were the use of:

- *Intermediate* states to model significant behaviour while a method is executing.
- *Asynchronous* events to model the entry and exit to intermediate states of otherwise synchronous method calls.
- *Instance-specific* information for expressing thread dependencies caused by waiting, joining, locking, and blocking on a lock.

The use of the statechart imposed *trade-offs*. The advantage is a visual formalism depicting the core idea of control flow states explicitly. The disadvantage is the problem to capture control flows from object-oriented programming in the world of statecharts which typically process events using queues. The conclusion is that for our restricted application area, where only a restricted part of Java had to be modelled, the use of a statechart is feasible. The simulation of a synchronous communication with mutual exclusive objects guarantees the correct behaviour also in the presence of asynchronous events for behaviour which is synchronised. For behaviour which is not synchronised, like the interrupt, the asynchronous mechanism is not a problem.

It is cumbersome to model the details involved in thread synchronisation with a statechart. On the one hand, the mechanisms are complex involving different pre- and postconditions and involving different ways of returning from methods. Also solutions using other formalisms will become lengthy. The drawback of the statechart is the mixture of explicit states and implicit states encoded in associated data fields. Still, we think that making the states explicit fosters better understanding of the Java synchronisation mechanism, e.g. when provided in addition to the purely textual language and library documentation.

Based on the statechart, we have formalised the notion of system state and execution history. These formalisations where used to formally specify failures and potentials. Failures implying dependencies are formalised using

the system state formalisation which makes the state of each thread and its instance-specific information explicit. For failures based on history it is more convenient to express them in terms of an execution trace. The same holds for potentials.

The developed statechart also helped us in determining failures we had not thought of before and which have not been documented elsewhere. This was the case with the join-induced deadlock, its potential, and the circular join from the previous chapter. Because the statechart makes the states involved in different kinds of synchronisation explicit, one can easier check different configurations, i.e. different combinations of thread instances and their states, and hence one can determine possible failure states.

# Chapter 7

# Trace-based Data Collection

The previous chapters have presented the domain of concurrent liveness failures. They concluded with their formal specification as an answer to our first main requirement (see chapter 3). This chapter and the following two chapters will present concepts for automated detection of these failures, thereby addressing our second main requirement. These concepts are implemented in our prototype, the Java Visualisation environment *JAVIS*.

In chapter 3 we have already structured the requirements for automated detection of concurrent liveness failures into the tasks data collection, data analysis and data visualisation (depicted once more in Fig. 7.1). Each task is influenced by the failure domain, whose characteristics have already been presented, by task-specific requirements for user control, and by task-specific standards. Now, that we have informally and formally presented the failure domain in the previous chapters, we are able to refine our requirements for the three tasks depicted in the centre. For each of them, we will discuss the refined requirements, related work, and our proposed solution in a separate chapter.

In the motivation in chapter 2 we have already argued that the failures are insufficiently covered by analysis tools and, when they are supported, not adequately visualised. Before developing new concepts, we will present the state-of-the-art in tracing, dynamic analysis, and visualisation of concurrent object-oriented programs to determine which concepts may be re-used. Most of the concepts stem from existing tools. These tools typically cover all three tasks. However, each tool is only of interest for our goals regarding one or two of the three tasks. The contribution to these tasks is then the reason why the tool was chosen for discussion and therefore the tool will be discussed in the respective chapter.

This chapter deals with *data collection* in JAVIS, which is indicated by the highlighted part in the centre of Fig. 7.1. Chapter 8 will deal with data

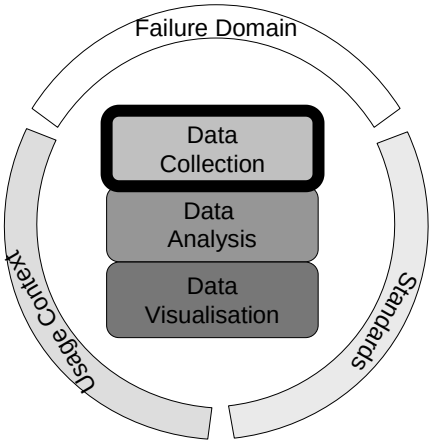


Figure 7.1: Requirements for Data Collection

analysis of the JAVIS environment and chapter 9 will conclude with data visualisation of the JAVIS environment.

## 7.1 Tracing Requirements

In chapter 3 we have already outlined that our approach to collecting data from a running program will be based on tracing, i.e. we will collect data about the execution history of a running program.

Although many failures are detected in a state, the state information is not sufficient to reconstruct the execution order which lead into the failure state. Instead, the execution history can provide better insights in how a failure developed. At the same time, the execution history can provide the same information which can be found in the individual states. For the detection of a few failures and for many potentials, the execution history is needed. Therefore, tracing is the approach which can provide information for both, state-based and history-based failures and potentials.

We will draw on the failure specifications in order to define the granularity and level of abstraction of the collected trace data. This is dealt with in the next section. In the section to follow we will refine requirements for the tracing method generating the required trace data.

### 7.1.1 Format, Schema, and Encoding

Here we will determine the requirements for a *schema* for trace data, i.e. a definition of the structure and the contents of a trace. We do not impose specific requirements on the *encoding* of the trace data, i.e. how the trace is represented and compressed. Schema and encoding together are called *format*. In the remainder of this section we will present our requirements for the schema.

A *trace* is a time-ordered sequence of event-records [KRR98]. The order of the traced events must be kept, either explicitly, e.g. using time stamps, or implicitly, e.g. by a relative position. Time stamps are needed if events can happen and be collected in parallel, e.g. in the presence of true concurrency or in a distributed setting. If the concurrency is implemented by interleaving, i.e. by executing one thread at a time and changing between threads, time stamps do not add information about concurrent execution apart from performance information. Therefore, time stamps are optional and we do not require a specific format for them. As most JVMs are used with single processor machines or do not use an additional processor, we assume interleaving semantics for our trace implementation.

The data to be collected is mainly determined by the failure characteristics. Concurrent liveness failures are a subset of synchronised behaviour. For the purpose of failure detection, only method calls with an effect on synchronisation have to be traced, including synchronized-methods and -blocks.

Note that the trace we intend to define here will defer from the notion of trace introduced in our formal model in the previous chapter. This is not surprising, because now we need to define a trace in terms of an executing Java program while in the formal model we were defining a trace in terms of events from a statechart. The statechart had been designed to simulate a part of behaviour of Java thereby mapping Java behaviour to a different paradigm.

In order to be able to detect the state-based failures in an execution trace, the trace has to contain an event which indicates a blocking method call. With such an event, we are prepared for offline analysis of failures involving blocking. Without such an event, blocking can only be decided with the following workaround. If a blocking takes place, the trace would contain the last successful method entry or exit taking place before the call to synchronized. One would need to look at the source code following the last traced event to decide whether the blocking is caused by synchronized or by other code such as a loop or I/O.

For each method call or synchronized-block we require to store

- **caller object and class** (optional because it can be computed from

previous events)

- **callee object and class**
- **method name**
- **parameters** (optional)
- **thread of control object and class**

  (if necessary for identification also its virtual machine and the host, on which the virtual machine is running)
- **timestamp** (optional)

Having discussed in depth the importance of control flow states of threads during method call execution, we require that for each method call observed the trace stores

- when it was **entered**,
- when it was **exited**,
- and in addition whether it was **blocked** on a lock before it could be successfully executed.

A synchronized-block can be treated as a method call except for the method name that is omitted. Caller, callee, and parameter objects can be stored by an identifier and class. We do not need to store the complete state, i.e. fields and values, of each object. Parameters can be stored by their name and value. Note that here we consider also constructors as method calls.

While it is sufficient for failure detection to store only selected method calls in the trace (those dealing with synchronisation), with this approach it is however difficult to provide a coherent picture of the program execution, as not every call in a sequence and not every nested call is traced. The navigation paths between objects are incomplete. This inhibits the understanding of the general behaviour and can be counterproductive when trying to understand the execution leading into a failure and especially when trying to understand potentials. Failures and potentials are restricted to synchronised behaviour but they are embedded in non-synchronised behaviour.

These deficiencies can be easily overcome by tracing all method calls, not only those involved in synchronisation. Only innermost methods of a sequences of nested method calls can be omitted without loosing coherence of control flow. Innermost methods are often calls to standard Java packages. Omitting calls to framework packages on the other hand can break the control

flow because a framework is often characterised by the fact, that it makes calls to application code. Other kinds of statements can be neglected. Thereby, we achieve a **general purpose** trace format include arbitrary method calls.

Moreover, a general purpose tracing format is also important to allow the tracing of method calls to libraries providing synchronisation. Such libraries can be used in addition to or instead of the Java synchronisation constructs. Method calls to such libraries have the same effects on threads as already described. If only the internal calls to synchronisation methods would be traced, the overall behaviour would be difficult to understand. Therefore, it is desirable that such method calls can be included in detection of failures and hence in tracing. We have already discussed, that future versions of Java will include synchronisation libraries. This situation makes a trace format with arbitrary method calls even more important.

From the perspective of failure detection it would make sense to declare all synchronisation methods as required for the format and all other methods as optional. But tracing all synchronized-methods may provoke a large space and time overhead. In order to give the user more flexibility we allow that only selected parts of a program be traced.

The described format for tracing method calls on objects is almost equivalent to UML interaction diagrams [UMLa].

Although the requirements for our format are fixed, we desire that the schema is **extensible** regarding entries for one call, e.g. time stamps or performance annotations, and regarding additional events, e.g. statements.

We can determine one such extension already from the intended use of the tracing facility. When an already started program is traced, the trace will contain method returns for which no entry has been traced. However, the traced program has the corresponding entries on its call stack. Therefore, we allow the trace to contain the **call stack** of each thread. This part of the trace will therefore contain method entries in our trace format but they are only ordered with respect to one thread. To avoid misunderstanding, we need an extra entry for these calls which identifies them as call stack method entries.

### 7.1.2 Trace Generation

In principal, the required trace data format is the essential requirement which determines the choice of method. Only the optional features can be neglected.

Every trace method has to face a problem which cannot be avoided. It disturbs the running program either because the program is transformed and therefore can change behaviour, or because the runtime environment executes the program differently due to the runtime environment's additional task to

generate traces. This problem is known as *probe effect* or Heisenberg effect. It is especially severe for concurrent programs [Kra98]. Here, the observed program can behave differently due to different scheduling in the presence of a changed program or a runtime environment with additional tasks. This sometimes also provokes failures, also called "Heisenbugs", but is likely to hide as many failures as provoked. The ultimate answer to this can only be deterministic testing.

For using tracing in different scenarios such as testing and debugging we have also determined the core requirement that we need **independent** trace data collection facilities. This choice was made in order not to be dependent on a tool which allows tracing only in combination with another functionality such as testing or stepwise debugging. Therefore, our only core requirement is, that tracing requires that the program is executed. It should not be dependent on a specific tool executing the program.

Also the tracing should have a loose coupling with the subsequent tasks of analysis and visualisation to foster interoperability with different analysis and visualisation components. The general question in this respect is, if the data are **online** or **offline** analysed or visualised. Online can mean either initiated by the tracing tool or more loosely coupled via streaming to shared memory or to a communication channel from where another tool can read the trace immediately. The highest degree of decoupling is reached if the trace is stored and subsequent tasks can access the trace *post-mortem*, i.e. after its generation. Therefore we aim at trace file generation.

Usually one is looking only for application-specific method calls and typically one can limit the search for errors to specific packages. Non-related information should be discarded. As already mentioned, a tracing method involves time and space overhead. Therefore the tracing method must be **configurable**. We require that it must be possible to select which method calls are traced, and that optional information about events such as parameters can be omitted. For method selection, it must be possible to either select individual methods, or to select methods by part of the name, class or package. This is also called *filtering*. It should be easy to extend the tracing method in order to extend the trace format.

Tracing itself should be as flexible as possible regarding when and where it is connected to the observed program, i.e. **time** and **location**. The user has to control the starting and the stopping of the trace. For analysing problematic program behaviour it can be required that this is done also after a program has been started. It can be required to pause the tracing (but not the program), also termed drive-by-analysis [PMR+01]. The user may also want to specify other kind of starting and stopping criteria such as criteria expressed in terms of the program behaviour. For instance, a trace could be

stopped after a failure has occurred and has been detected. In the presence of the internet also distributed debugging scenarios are conceivable where a non-local program needs to be traced remotely.

## 7.2 Related Work

When we started in 2000 to develop concepts for supporting automated detection with a tool, tracing was the first component to be designed and built. At that time there were no reusable general purpose tracing tools for tracing concurrent Java programs nor general purpose tracing formats. Only very few tools for dynamic analysis of concurrent Java based on tracing were available.

Since then the research has increased exponentially in this area with almost a hundred academic and a few commercial tools based on tracing being published. A few reusable tools and frameworks have been proposed and recently, the need for a standard trace format has been expressed more often.

In the following, we discuss related work with respect to trace formats and trace methods. We also discuss the degree of flexibility in user control. We start by introducing the basic techniques on which all these approaches rely. Then we present in detail how different approaches choose among these techniques to achieve their goals. The presented approaches are from the area of Java programming and concurrent programming.

When data about performance of these approaches is available we discuss it.

### 7.2.1 Basic Techniques

#### Schemata and Encoding

It is desirable to reuse trace schemata in order to foster interoperability. At the lowest level, a schema is specifying which bit or byte of a file contains what information. Schemas may also be defined as the streaming result of runtime memory objects. In both cases, the schema includes the encoding and the distinction between the two is blurred. More advanced schema definitions contain pairs of name and value thereby not imposing orders inside an event. At the highest level is a schema defined with a data description language (DDL) which is executable, i.e. supported by a tool to generate useful components from the data description. Depending on the requirements, both, low-level and high-level techniques, can be useful. Extensibility of schemata can also be important in the fast evolving programming world.

Encoding is important and influenced by factors such as compression, consistency or ability for network streaming. It must also serve purposes such as fast readability or analysis. Sometimes also legibility can be required. Since the advent of more widespread mark-up languages, sometimes XML [XML00], is proposed as encoding. XML can be advantageous because it simplifies exchange, storing, and retrieving traces in XML databases. However, developers of tracing facilities consider XML too verbose and therefore too inefficient [BDE+02].

None of our requirements seem to exclude any of the schema or encoding techniques.

### Generation of Execution Information

In the motivation, we have already coarsely sketched the two main approaches for collecting information from a running program, namely *code instrumentation* generating file output, or calls to helper classes generating output, and *instrumentation of the runtime environment*. Instrumentation always requires a modification step. Instrumentation means, that this modification typically augments the program by inserting statements. It is different from program transformation which means more arbitrary modifications where existing code can be replaced, re-ordered etc. Often, instrumentations of the runtime environment are already part of the environment and can be used via dedicated APIs of the environment.

In the following, we have a more detailed look, at what levels source code instrumentation and runtime instrumentation or interfaces can occur with respect to Java (see Fig. 7.2). We explain each level and discuss advantages and disadvantages in more depth.

- Java source code, i.e. the *Java program*, is not always available. Therefore, instrumentation of *Java byte code* is more general. It might be even the case that Java byte code is only instrumented at load-time. The general advantage of source code and byte code instrumentation is that it is faster than interaction with the runtime environment.

  While method entry or exit can easily be instrumented by inserting code before the first line and before each return or after the last line, the synchronized-keyword is problematic. If code is inserted after the method call or block, the blocking situation can never be traced. If code is inserted before a synchronized-method or -block, the blocking appears in the trace but one does not know if the call is executed immediately or ever. In the byte code, the problem remains the same. The keyword synchronized is mapped to a pair of monitorenter and

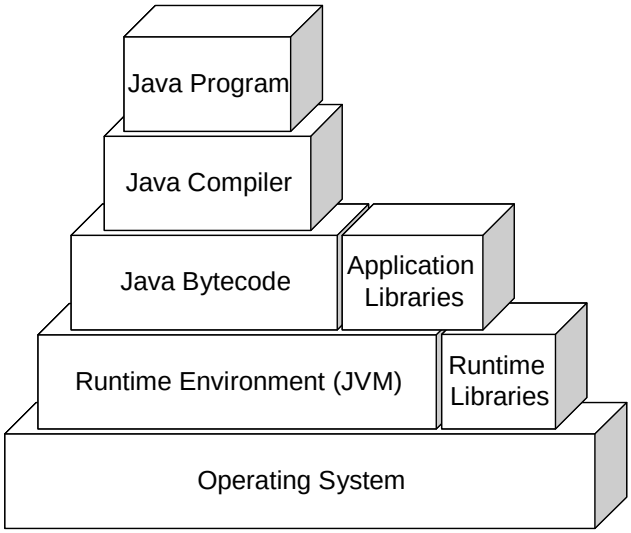


Figure 7.2: Levels of Instrumentation

monitorexit statements in byte code. Hence, also here code can only be inserted before or after monitorenter. Here, the JVM has the advantage that it knows when a call was done and when it was blocked.

Another problem is that, when an exception occurs, the synchronized-method or -block is left immediately and the release of the lock cannot be traced via an instrumentation. The same is true for the byte code. Therefore, [GH03] proposes to add a try-catch-block around each monitorenter/exit.

- Instead of a separate transformation tool to achieve the instrumentation at source code level before compilation or at byte code level after compilation, one could consider modifying the *Java compiler* for integrating this task. This is similar to compiling a Java program with a debug option which generates debug information otherwise not included in the byte code. The limitations are however the same as for code instrumentation.

- Depending on what kind of information is needed, it could be more useful to instrument the *application libraries* usually given as byte code. The advantage is that only a library has to be modified but not each program using the library. This is not feasible, as we are not using a third party library for synchronisation.

- Solutions for the runtime environment have the advantage that they

can be installed once and the repetition of the transformation step is superfluous. Here, one can think of different options, namely instrumenting *runtime libraries* or instrumenting the *runtime environment (JVM)* itself. The first would of course not yield a general tracing approach but could be interesting for our synchronisation domain. It is however not possible for Java, as the synchronisation primitives are only partly in a runtime library java.lang.Thread but also partly proper language elements such as the synchronized-keyword.

Therefore, the Java alternative to code instrumentation or compiler instrumentation lies only with the Java Virtual Machine (JVM) itself. The JVM can be instrumented to generate traces which means that a modified JVM has to be provided. Luckily, for the most applications which want to observe a running JVM this is not necessary. The Java runtime environment already provides a set of runtime application programming interfaces (APIs) for observation tasks. This includes method calls and blocking on monitors. It has also be pointed out by an approach for deadlock detection in a different setting (pthreads) that it is necessary that the blocking can be explicitly queried. The JVM can also pause a running program which cannot be achieved using code instrumentation. In general, runtime-based solution have a higher performance overhead than code instrumentations.

- For special purpose data collection one could even consider using the *operation system* (OS), e.g. if Java only provides an interface to using services or libraries of the OS. Often, the OS provides already APIs for observing such behaviour. Instrumenting the OS should be the last thing to be considered. It is the deepest layer and modifications will effect also other programs using it. This could however make sense when a JVM uses extensively OS-level threads and their synchronisation concepts.

With regard to user control, all approaches using an instrumentation of the program itself need to provide means to insert and remove instrumentation (or keep uninstrumented copies). This might not suffice when tracing is switched on and of while the program continues to run. The more flexible the tracing should be, the more functionality has to be built into the instrumentation. This is not a problem for the direct interaction with the components having complete control about the execution of a program.

Although some major disadvantages of instrumenting the Java program itself in different levels have been pointed out it cannot be said in general that this approach is unfeasible.

### 7.2.2 Trace Formats

In this section, we look at explicit proposals for an exchangeable format. The runtime APIs presented in the next section and some of the tools presented in the following sections also provide formats but they do not explicitly address interoperability.

#### Compact Trace Format

A recent yet very initial proposal to standardisation for object-oriented traces, *Compact Trace Format (CTF)* [HLL03], proposes to base traces on UML sequence diagrams [UMLa]. The schema contains concepts for eliminating recurring sequences and loops. It does not explicitly deal with synchronisation nor with blocking method calls.

#### STEP Framework

More mature work has been provided recently by the *STEP* project [BDE+02]. It addresses standardisation at the level of schemata and encoding, and provides a framework for efficient encoding of trace data. For definition of trace schemata, STEP provides a data definition language. It also provides configurable trace encoding. STEP has been used for example for defining a schema for events from the Java Virtual Machine Profiler Interface (JVMPI), which will be introduced in the next section.

### 7.2.3 Code Instrumentation

Flexible instrumentation tools are rare. The following approaches do not provide solutions for dealing with the synchronized-problem.

#### JSpy and JTrek

Only recently the *JSpy* [GH03] Java byte code instrumentation tool was proposed. It was built for the *Java PathExplorer JPax* [Hav00], a tool for runtime analysis of Java programs, which is part of the *NASA Java Pathfinder* project which develops model checking techniques for Java [JPF03]. JSpy declaratively specifies what should be traced, under what condition, and with what effect Amongst others, JSpy reports all successful acquisitions and releases of locks but not the blocking. JSpy support streaming and file output.

JSpy is a customised interface to the byte code instrumentation tool *JTrek* by Compaq SRC [JTr] which also supports the conditional tracing but allows

to insert only restricted kinds of codes. A performance evaluation is not available.

#### Aspect-Oriented Programming (AOP)

The *aspect-oriented programming (AOP)* approach, especially the language **AspectJ** [AJ98, KLM+97], can be seen as an approach for instrumenting method calls and therefore seems to meet the requirements for tracing method calls. However, at the time when we examined it, it was not flexible enough because it could only instrument method bodies. This would not allow to insert code directly before a synchronized-method. This was also observed by the developers of JSpy [GH03] which additionally noted that not enough information can be gathered about the running program. More recent versions allow also method calls to be instrumented, not only bodies. Still for counting locking and releasing locks this method is not apt.

As for any instrumentation approach, it is cumbersome to implement interactive control over the instrumentation.

### 7.2.4 Runtime APIs

The Java Development Kit (JDK) provides several APIs for developing programs which observe a running Java program and interact with it through the JVM on which it runs [Jav01].

The *observation principle* is always the same: The API defines a set of events which can be observed. Via the API the observing program can register for selected events and can provide callback functions which are called when the events occur. With the call, also an event record is provided containing details about the event occurrence. The level of detail is configurable via the API. There are several different APIs which we present in the following.

#### Java Virtual Machine Profiler Interface (JVMPI)

The *Java Virtual Machine Profiler Interface* (JVMPI) is intended to support the collection of profiling data from which statistics about time and space consumption and performance bottle necks can be computed.

The JVMPI is a native interface in C. A profiler has to be developed as a separate C program.

Specifically for multithreading, the JVMPI allows profilers to observe a set of predefined events such as starting and ending of threads, contention of monitor objects and waiting of threads. The event records are however

designed for statistical evaluation and not for reconstructing the execution. For instance, the event record for method entries contains only method name and identifier of the called object but not the thread or the caller identifier.

### Java Platform Debugger Architecture (JPDA)

The *Java Platform Debugger Architecture (JPDA)* (since JDK 1.3) contains two interfaces and a protocol intended to develop debuggers (see Fig. 7.3). It allows for more detailed observations than the JVMPI.

The **Java Virtual Machine Debugger Interface** (JVMDI) is a native interface implemented by each JVM, typically in C.

It provides basic debugging functionality such as inserting breakpoints and watchpoints, starting and stopping the debugged program, and exhaustive querying of the state of a debugged program. The implementation directly using the JVMDI is written in the native code and runs on the same JVM as the observed program.

The **Java Debug Interface** (JDI) is a Java interface which internally uses the JVMDI. It provides the same functionality in Java as the JVMDI but can be used on a different JVM.

This is achieved by the **Java Debug Wire Protocol** (JDWP), a standard for using the JVMDI. A backend using the JMVDI runs on the same JVM as the observed program. The frontend which implements the JDI runs on a different JVM, either on the same host or on any other host. In each case, the communication is implementing the JDWP.

The JDI provides detailed enough event records for method entry and exit events from which program execution can be reconstructed. Synchronized method call entries which are blocked do not generate events, only successful method entries. However, the JDI provides a method currentContendedMonitor which determines for a given thread the object it attempts to lock. The JPDA does not provide time stamps with the events.

The JPDA seems to fulfill the requirements for our trace format requirements. It allows method entries and exits including callee and control flow thread to be traced and it allows the state of objects used as monitors to be traced. The complete information about entries and exits is not available via the JVMPI. The JPDA allows debuggerst to trace already running programs and supports also remote tracing. Also, the JPDA allows not only to generate trace files but to couple arbitrary programs to its provided interfaces which will be executed while the observed program is executed. This allows tracer to provide additional functionality, namely optional online analysis and optional online visualisation. The JPDA also allows flexible control during tracing.

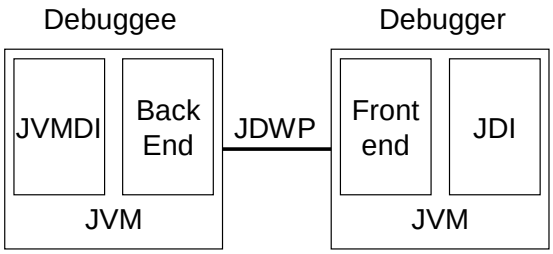


Figure 7.3: The Java Platform Debugger Architecture (JPDA)

These varieties makes the JPDA more flexible than instrumentation. Therefore, we prefer such a solution over the instrumentation-based approach. More details on the JPDA will be provided when presenting our solution.

It is noteworthy, that for Java tracing there is no specific interface of the Java runtime environment, only the JPDA can be used. It has not been designed for tracing, albeit the interface is increasingly used for these kind of applications. We have already noted, that performance is a problem. As performance evaluations are not available we have conducted our own experiments.

### Packages Runtime and Log4J

The class Runtime of package java.lang provides a method traceMethodCalls. When this method has been called, the JVM outputs all method entries and method exits. Regarding source and target of a call, only the class names of the called methods and the thread of control are in the output. Note that this class cannot really be considered an API, as it is tightly integrated with the observed Java program itself.

For very simple tracing there are also libraries such as *Log4J* [L4J] but none of the tools discussed here is using it. With Log4J it is possible to enable logging at runtime. Logging behaviour can be controlled by editing a configuration file. The Log4J package is designed so that the additional statements for logging can remain in shipped code without incurring a heavy performance cost. The logging output is not as detailed as we require. It does not support locking.

### 7.2.5 Debuggers

Regarding classical debuggers we have already shown their deficiencies with respect to the analysis and visualisation of concurrent program execution. Here, we will shortly discuss their approach to data collection and to user control. Then we will discuss extensions such as history-oriented debuggers and deterministic debuggers.

#### Classical Debuggers

Java debuggers of today are using the Java Platform Debugger Architecture (JPDA) such as JBuilder [JBd01] and others we have mentioned.

The JPDA was designed with only debuggers in mind and supports breakpoints, watchpoints, stepwise execution, complete inspection of the program state and the like. For this kind of functionality, performance is not a critical issue. Debuggers demonstrate what information is accessible with the JPDA. They access more information via the JPDA than what it required for our trace format, e.g. the complete object-graph. However, debuggers do not store any of the displayed information. Because the JPDA provides remote functionality, debuggers can also attach to remote programs. Debuggers can also attach to already running programs. Modern Java debuggers support this to a certain extent.

An interesting aspect of data collection in debuggers is the highly user-interactive process. The user has fine-grained control over starting and stopping the execution with conditional breakpoints and watchpoints.

As we said we do not want to be dependent on a given tool functionality for producing traces. However, it would be feasible to extend a debugger with tracing as a new functionality. A problem is, that for commercial tools source code is not available.

#### History-based Debuggers

The key idea of tracing, namely access to execution history, is also pursued by a set of trace-based tools which see themselves in the tradition of debuggers. Their mission is to consider what information will help the programmer instead of what information can be provided while the program is running [Lew03a].

The **Omniscient Debugger** [Lew03a] provides typical debugger functionality but generates a trace. It still adheres to the stepwise presentation of the execution by a source code view. The Omniscient debugger is a post-mortem tool, which limits its usefulness concerning interactive program execution. It first collects the trace and then allows flexible trace navigation.

It does not provide any more analysis capabilities than a normal debugger. The Omniscient debugger instruments Java byte code at load-time to generate the tracefile. Tracing uses filters and the level of detail can be configured. The trace format is not dedicated to locking problems and cannot identify blocking.

The slowdown factor is 10-300. It takes 10$\mu$s to store a method call and 1$\mu$s to store an assignment. The average is assumed to be 2$\mu$s. In 20s, a 2GB trace file is generated.

Only an early predecessor, **ZStep95**, a debugger for LISP [LF98] also provided a large variety of visualisations of the history itself at different levels of abstraction. At the time of its publication, the feature of reversibility was considered to be the most important new feature. Reversibility means that the trace can be navigated forwards and backwards.

Also tools like **JProbe Threadalyzer** [JPr00], and **Assure Threadanalyzer** [Ass] need to store data about the execution history. However, they collect data highly selectively governed by their intended analysis. They do not store or output it as traces but use it for statistical results and for identifying individual failures such as the deadlock. They collect data via the JVMPI and JPDA. These tools existed when we started with this thesis but as they are not open source they are not reusable or extensible.

### Deterministic Debuggers

Tools for deterministic tracing of concurrent Java programs can improve the testing and debugging considerably. The **Delta Debugging** approach [ZH02] starts from a nondeterministic or user-defined thread execution schedule which is then automatically examined and narrowed down to a more significant thread schedule. The **Dejavu** tool [CZ02] supports execution of a given deterministic thread schedule using a modified Java virtual machine called *Jalapeno*. Dejavu was first used to isolate failure inducing thread schedules [CZ02].

Deterministic replay tools also have a long tradition in parallel and distributed computing such as the **collect-and-replay** tool by [CT91].

We consider this a feature which is more related to the testing area and we envision that approaches like ours can be used together with these kind of tools.

### 7.2.6 Trace-based Tools

#### Java Dynamic Analyzer (JaDA)

The *Java Dynamic Analyzer (JaDA)* [BT98] is an academic approach to concurrent Java tracing. It traces only synchronisation events and it supports deterministic tracing by prescribing synchronisation sequences or replaying already recorded synchronisation sequences. This approach relies on code instrumentation. (It is not indicated whether on source code or byte code level.) For the tracing of synchronisation, the approach assumes that either specific libraries are used which can be instrumented or that the code is transformed to use such libraries. This is to circumvent the problems with the instrumentation of the keyword synchronized. For the deterministic execution, thread classes are instrumented such that they inherit from a specific new class. With this technique the main thread and the GUI event handling thread cannot be transformed. The approach also lacks flexibility, as the tracing cannot be switched on and off while the program is running.

#### Jinsight

IBM *Jinsight Version 2.1* is an industrial prototype [Jin, PKV98] and the most prominent example of a tool combining tracing, analysis and visualisation. Its purpose is the detection of anomalous behaviour in Java with a focus on object-oriented time and space performance. The tool first collects data and then post-mortem visualises individual method calls of different threads.

Jinsight provides two ways to collect data, behaviour tracing and object population snapshots. For tracing, Jinsight uses the JPDA albeit with an instrumented JVM for generating trace files. The trace files are in a proprietary format in a non-legible encoding and their definition is not published. This makes it difficult to reason about there suitability. Because none of the analysis facilities or the views allow to detect a deadlock at the level of granularity of identifying blocked method calls we conclude that the trace format does not contain these kind of entries. The tool only traces the locking and release of locks.

Jinsight provides flexible control supporting remote drive-by-analysis, e.g. in order to trace Webservers to identify problematic behaviour.

The trace implementation has changed over time. Jinsight 2.1 provides instrumentation via a profiling agent which uses JVMPI interface. Jinsight 2.0 provided instrumentation as entire replacement JVMs.

Trace files grow at a rate of 15-30 MB/minute. A few tens of megabyte contain over 1 million events. One event hence seem to take a few tens of

byte in a file. Traces are read/visualised at rate of 12 MB/minute. Jinsight visualises traces up to a few tens of megabyte. Filtering while reading allows visualisation of traces otherwise too large.

Jinsight itself is not developed further but its concepts are integrated in commercial tools now such as IBM Websphere [WbS] or IBM Hyades for Eclipse [Hya].

#### Tracing in Distributed Environments

The Siemens **Test and Monitoring Tool (TMT)** is a noteworthy approach aiming at tracing different kinds of middleware in distributed systems, for instance Java programs with remote method invocation (RMI) [CGR01, BCH+00]. It collects trace data via a very low level "runtime environment", namely the network layer. It listens to the network traffic and re-combines network packages sent, for instances into remote method invocations.

It is important to point out that tracing is not a new concept. Tracing has a long tradition in the area of **telecommunication**. Here it is used with domain-specific languages. As a consequence, tracing is often a configurable facility built into the domain-specific operating systems. Since the telecommunication area faces new challenges such as reverse engineering their long grown systems new concepts are needed. In the E-CARES project [MW03] reverse engineering tools using tracing are proposed.

### 7.2.7 Comparison

We summarise the discussion of related work with Table 7.1. For each approach we classify the trace schema and encoding, the kind of trace output (file or only runtime data), the tracing method (instrumentation or runtime modifications).

We did not include performance in the table because it is only published for few approaches. We will discuss performance for the approaches which provide measurements and for our proposed solution at the end of this chapter in more depth. We draw the following conclusions from the related work. Firstly, no tool provides a trace format schema which fulfills our requirements. Even those schemata which provide the tracing of synchronized do not distinguish between blocking and successful locking. At the time when we built our tracer, there were not extensible frameworks. Nowadays, JSpy or STEP provide useful support.

The only more high-level API is the JPDA with its interfaces. It has gained practical relevance, has already been used for tracing in Jinsight, and

| *Approach* | *Year* | *Schema* | *Encoding* | *Output* | *Method* |
|---|---|---|---|---|---|
| CTF | 2003 | method calls, multithreaded | trees | independent | independent |
| STEP | 2002 | definable | definable | definable | independent |
| JSpy/JTrek | 2003 | multithreaded, locking and unlocking | unknown | file/socket streaming | byte code instrumentation |
| AspectJ | 2004 | executes code before and after method call no blocking | user defined | user defined | byte code instrumentation |
| JVMPI | - | method calls, multithreaded | independent | independent | JVM API |
| JPDA (JVMDI, JDI) | - | thread identifier/class, method entry/exit, callee identifier/class, blocking on locks | independent | independent | JVM API |
| Runtime | - | thread name, method entry or exit, class name | ASCII | file, console | runtime library |
| Log4J | - | thread name, method entry/exit, class name | ASCII | file, console | runtime library |
| Omniscient | 2003 | calls and statements, multithreaded, no locking | compressed by coding in bytes | file | byte code instrumentation |
| ZStep95 | 1995 | LISP paradigm, not multithreaded | unknown | file | reflection |
| JProbe, Assure | - | multithreaded, partial trace information for profiling | - | none | unknown |
| Delta Debugging | 2000 | multithreaded | unknown | file | modified JVM |
| JaDA | 1998 | interface operations of synchronisation concepts, monitor lock and release, time stamps | unknown | file (per synchronisation object) | application library and source code instrumentation |
| Jinsight | 1998-2001 | multithreaded, method calls, no blocking | unknown | file | JPDA, modified JVM |
| TMT | 2000 | Java RMI | unknown | unknown | network listener |

Table 7.1: Trace Collection Approaches

seems to be more flexible than instrumentation and flexible enough for our tracing goals. It especially seems to be more flexible and more usable than the JaDA architecture.

## 7.3 JAVIS-Tracer for Concurrent Java Programs

In this section we define the trace format and the tracing method used by the JAVIS tracer. We conclude by presenting the resulting architecture of our tracer.

### 7.3.1 Trace Format

#### Schema

The schema is a linear list of entries. The traced events are method entries and method exits, including constructor calls. Each event record contains the following entries:

1. a unique **name of the thread of control** in which the event takes place
2. the **class of the thread of control** and a unique internal **reference**
3. the **class of the call target** and a unique internal **reference**
4. the name of the **called method with optional parameters**, given by class and reference
5. a boolean **synchronisation flag** indicating whether a method is synchronized
6. a string indicating the **state of a method call**, i.e. "enter", "exit" or "acquire" (for a thread being blocked when attempting to acquire a lock)

Note that for the concrete values we rely on the conventions used by the JPDA. Threads and objects are given by the references provided by a JVM which are integer IDs. In addition, each thread has an string with an ordinal number in the JVM, except the main thread which only has a string "main".

To make these IDs and strings unique they have to be qualified by the host and the JVM instance of the running program because we allow that

```
beethoven_VM#1_main:java.lang.Thread@beethoven_VM#1_ID#1:Employee@beethoven_VM#1_ID#92:
  <init>(Bank@beethoven_VM#1_ID#74), Terminal@beethoven_VM#1_ID#91)):false:enter

beethoven_VM#1_main:java.lang.Thread@beethoven_VM#1_ID#1:Employee@beethoven_VM#1_ID#-1:<init>:
  false:exit

beethoven_VM#1_Thread-0:Employee@beethoven_VM#1_ID#90:Employee@beethoven_VM#1_ID#90:run()
  :false:enter
```

Figure 7.4: Trace Format Example

data from different JVMs can be collected in one trace file. The host is given by its name if accessed within a local network or by an IP address if accessed via internet.

Note that we do not store the caller within an event record because the caller can be inferred from the preceding event record within the same thread of control. It is the target object of the preceding method entry. The method call entry can also be a constructor call, denoted by "<init>", or a synchronized-block, denoted by "<block>". in the latter case, the target denotes the object used as monitor for the block.

A method entry which is not traced but found on the call stack has the same format as a normal method entry, only the method name is preceded by a label "<stack>".

### Encoding

At present, the trace format is encoded in ASCII. This has the advantage, that it can be read by humans without any tools. Also, it is our choice having seen that the general discussion is still open ended and that there are more arguments against an XML-based solution than pro.

An event record does qualify its entries by position but not by a qualifying field name. Entries are separated by ":". Within an entry, IDs are separated from class names by "@". Each line contains one event record. The sequence of event records presents the timing order of the events. We do not use time stamps.

Figure 7.4 gives an example, but note that this format was not designed for ease of reading but for completeness of information. The example contains three subsequent lines from a generated trace file. For this trace, parameters have been suppressed, and are only given with constructors which are depicted with <init>. The first line traces the constructor of an Employee object, the next line the return, the third line the entering of the run()-method on the created object. Note that start() cannot be trace with the JPDA, only run().

### 7.3.2 Trace File Generation

Here, we describe how the trace file is generated using the JPDA. We will describe our choice according to the dimensions of remoteness, execution time control, event selection and callback behaviour, and filtering options.

#### Location

Our solution uses the JDI because it provides more comfort and greater flexibility than the JVMDI. That implies, our trace program runs on a different JVM than the traced program but potentially on the same computer (see also the right hand side in Fig. 7.3).

For the JDWP we have chosen TCP/IP [JPD] because this solution is applicable for the case that the same host is used but also for the case that a remote host is used. This allows remote tracing.Remote tracing is implemented simply by choosing from the different initialisation capabilities from the JDI API. The chosen connection uses ports [JPD]. Therefore, the only requirement is that the observed program opens a port when started.

#### Time

Here we describe how the execution phase during which a program is traced can be controlled. There are two ways to start the tracing:

- start the observed program from the tracer
- attach the tracer to an already running program (implying re-attaching)

This is implemented straightforward by connecting to an existing port or by starting the observed program with such a port.

We provide different ways to stop the tracing.

- stop the tracing by user choice
- stop the tracing when a failure is detected
- stop the observed program

The conditional stopping uses the possibility that via the JPDA not only output can be generated but also arbitrary calculations can be carried out while observing events. We provide the option that the tracing can be stopped when a failure is detected. Details on the online detection will be provided in the next chapter on trace analysis.

During tracing we also provide the means to pause a trace simply for convenience. That means that the tracing is stopped because the observed program is paused by the tracer.

### Event Selection

Via the JDI API we chose events for which we provide callback methods. The following events are chosen:

- METHOD_ENTRY
- METHOD_EXIT

With these events, additional information is transmitted to the callbacks. The information provided contains the name and the ID of the thread of control, the class of the call target, the name of the method, and whether the method was synchronised. In the case of METHOD_EXIT, the target is not provided but can be inferred from the METHOD_ENTRY event. It has to be noted that the ID of the target is also missing.

For accessing information which is not part of the events generated, the JDI API allows suspension of the JVM when an event is generated. When the events are chosen, the suspend policy can be determined. Either all or no thread can be suspended, or only the thread in which the event happened can be suspended.

```
...
//vm is a handle to an attached vm

EventRequestManager mgr = vm.eventRequestManager();

MethodEntryRequest menr = mgr.createMethodEntryRequest();
menr.setSuspendPolicy(EventRequest.SUSPEND_ALL);
menr.enable;
...
```

When the JVM is suspended, the callback method can use the thread reference provided by the event to examine its stack frame, where the missing ID of the target object can be found. When the callback returns, the suspension is finished and the program continues running.

There are no events for blocked method calls. Our idea was therefore, to use the suspend modus to query the JVM for the missing information. Experiments with different JVMs have shown that the JDI method ThreadReference.currentContendedMonitor(), which is supposed to return the acquired lock for a given thread, is not supported. The method ThreadReference.status() returns whether a thread is blocked but not the acquired lock. However, other experiments have shown, that some JVMs already provide the stack frame for the method call which is still waiting for the lock. In this stack frame we can find the ID of the call target and hence the ID of the lock. The Solaris JVM for Java 1.4 (see also the description in [Wey01]) provides

this kind of information while the JVM for Java 1.4 from SUN [Jav01] does not.

Combined with querying the thread status we can hence determine blocked calls. These queries are of cause triggered by other events. During tracing, references to all threads are stored. Whenever an entry event for a synchronized-method happens all other threads are queried for their monitor status. For each thread blocked, we determine the object. We can only do this for threads which have changed their status. Therefore, if a thread only shortly waits for lock, we might not be able to trace it. But for the purpose of detecting blocking involved in failures this approach is sufficient.

Entry and exit of synchronized-blocks are no predefined events in the JDI. Unfortunately, the aforementioned solution cannot be used with synchronized-blocks because they do not appear in the stack. In general, one could extend the JVMs with the desired functionality but this was not the focus of this thesis. As the Java technology is steadily evolving and especially its APIs we hope that better support will be provided in future. For these kinds of problems, feature enhancement requests can be sent to designers of the Java runtime environments. Therefore, here we cannot cover synchronized-blocks.

### Generating Trace Output

We have already described how the callbacks determine the needed information. Each callback writes the information to a file, to a console and to a text view in the tool.

The default is that also traces obtained from different JVMs are written to the same file. They can be identified by the JVM and host identifier.

Our solution could be easily extended with a streaming facility, e.g. writing the file output to shared memory or to a socket [JPD].

### Filtering

In order to reduce trace data we provide two kinds of filtering

- default filtering of all java packages (java.*, javax.*) and of JVM related packages (ibm.*, sun.*, com.sun.*)
- customisable filtering of arbitrary packages

Filters are added as exclusion filters to the event requests:

```
...
String[] exludes = {"java.*", "javax.*", "sun.*", "com.sun*", "ibm.*"};
for (int i = 1; i<excludes.length; i++) {
  menr.addClassExclusionFilter(exludes[i]);
}
...
```

In later Java versions [Jav04], it became also possible to define filters dynamically. We were not able to use this feature when we built our prototype.

#### Performance and Size

As no performance data was available for the JPDA, we conducted a measurement experiment [Meh03]. Note that the following measurements do include the data gathering remotely via the JDI but on the same machine. The settings for the experiments are the same as for the intended usage scenario of the tracer.

- 1000 method calls take 80 seconds when the full trace information is gathered, i.e. when the JDI suspend mechanism is used.

  1000 method calls take only 4 seconds when simpler trace information is gathered, i.e. a trace with the same number of calls but without the identifier of the called object and without detection of blocked method calls, i.e. if no suspension is used.

- 1000 method calls generate 2000 entries of an average of 85 bytes, altogether 170 KBytes.

- The JPDA with suspend is factor 100.000 slower than an ordinary Java program. Without suspend it is factor 4.000 slower.

Similar values are also obtained by other JPDA applications. The Omniscient debugger is a lot faster because it uses load-time instrumentation and also Jinsight is faster because it uses an instrumented JVM.

### 7.3.3 Tracer Architecture

We conclude this chapter by presenting an overview of the design of the tracer (see Fig. 7.5). The design is multithreaded itself.

- The class Trace contains the main functionality of the application. It can open connections to one or more JVMs.

- Class Connect is responsible for setting up the connection to one JVM and for closing down the connection. It is also responsible to handle any unforeseen connection problems and to take actions if the connection breaks unforeseen. This can for instance be closing open files. Therefore, Connect has a link to the Protocoller class. It is useful to have a separate thread for these tasks.

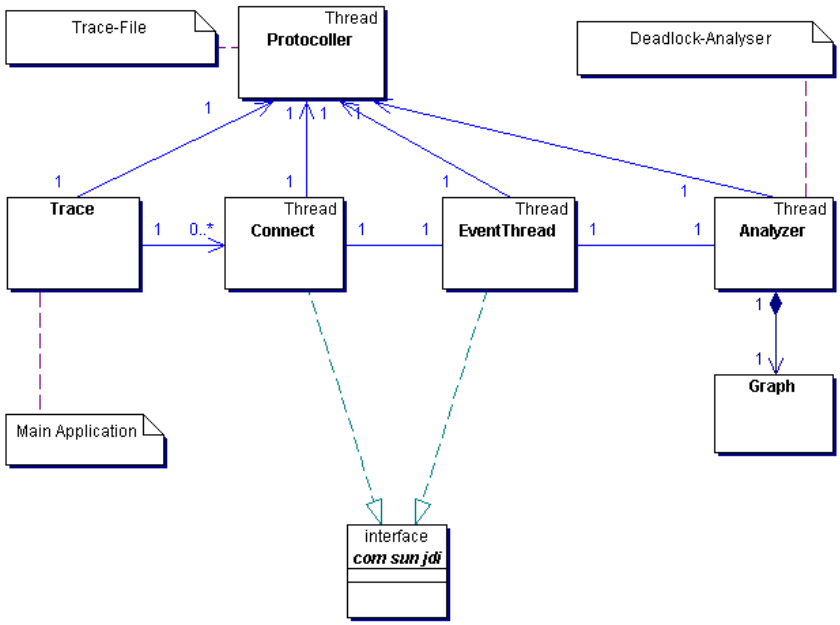


Figure 7.5: Class Diagram of the Tracer

- Class EventThread provides the callbacks. The callbacks are triggered when an event occurs for which the tracer is registered. It is better not to use the main()-thread for the callback computations and therefore a separate thread makes sense. The callbacks send trace data to the Protocoller.

  If online failure detection is enabled, an instance of EventThread creates a separate Analyzer thread for failure detection. The results it receives from the Analyzer are also sent to the Protocoller and hence are inserted into the trace file.

- Class Protocoller receives entries from one or more EventThreads. It maintains a buffer with trace file entries which are flushed from time to time to the trace file. Again, it is useful to have a separate thread.

- The Analyzer class can be used for online detection of failures. It uses a Graph class which provides a graph data structure with thread dependencies which is used for detecting failures. The Analyzer sends its results to the EventThread.

  The classes Analyzer and Graph will be described in detail in the next

chapter when we describe concepts for algorithms for failure and potential detection.

The main behaviour scenario of these classes is as follows. The main class creates the user interface. For each program whose execution shall be observed it creates a Connect thread. It also creates the Protocoller thread which is the same for all observed programs. The Connect thread sets up the connection. Then it creates the EventThread for this connection. If the user has enabled the online failure detection, the EventThread creates an Analyzer thread when a blocking method call event is traced.

## 7.4 Summary

In this chapter we have presented an in-depth discussion of tracing facilities. We have provided a solution using standard technology of the Java platform. The concepts for tracing have been implemented in the JAVIS prototype. The tracing facilities lay the ground for the following chapters.

We have presented the JPDA-based solution first in [Wey01] and [Meh02]. We also carried out an experiment to measure the performance of the solution, first presented in [Meh03].

# Chapter 8

# Trace-based Failure and Potential Analysis

In this chapter we draw on the precise failure and potential specifications in order to derive algorithms to detect them in traces collected from a program execution. We present existing algorithms and describe the choice for our prototype implementation JAVIS.

## 8.1 Analysis Requirements

Analysis depends on the preceding data collection, which provides the input to the analysis algorithms. The results of the analysis are in turn the input to the subsequent failure visualisation. Therefore, analysis is depicted in Fig. 8.1 as a component labelled *data analysis* layered between *data collection* and *data visualisation*. The goals of the analysis algorithms, i.e. the failures and potentials which have to be detected, are depicted by the arc labelled *failure domain* in Fig. 8.1. The detailed requirements for analysis are given in the following.

### 8.1.1 Functional Requirements

The functional requirements are that each occurrence of a failure and a potential in a given trace should be detected. Therefore, our approach should not be confused with source code based approaches intending to detect concurrency failures by analysing the source code.

The input to the algorithms is a trace in the format which was described in the last chapter. The trace format is designed such that the information is sufficient for the desirable detection algorithms. Therefore, the commu-

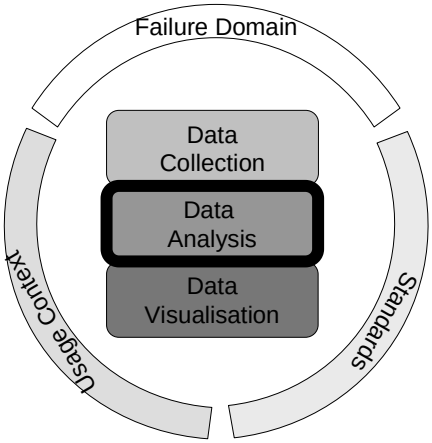


Figure 8.1: Requirements for Data Analysis

nication between data collection and data analysis is straightforward. First the trace data is collected, then the algorithms can be executed. It is not required that the analysis algorithms control what kind of data is collected, albeit this is a conceivable concept. For instance, the analysis component could change filters during tracing.

In the previous chapter, we have already pointed out that we include the information that a thread is blocking on a lock in our traces. This is necessary for detection of failures involving blocking, namely for deadlock, lockout, and join-induced deadlock. For detection of the corresponding potentials only successful locking is required. That means, that potential detection could work also from a trace format containing less information than ours. These details are important to know if the algorithms are implemented in a different context.

Although a trace is already easier to analyse than source code, algorithms for potential detection can incur intolerable overhead. As a consequence, it might be desirable to use a faster but less accurate algorithm. Where algorithms are not able to produce accurate results it should be indicated what kind of misses or false positives have to be expected.

For the output of analysis results we do not require a format. We propose to include them in the trace files. This will prove convenient for the visualisation component described in the next chapter as we will describe our own solution for reading trace file input.

### 8.1.2 Time and Space Complexity

As already pointed out, detection should compute results in a tolerable time. Also, space consumption could become an issue. Therefore, it is important to consider the time and space complexity of the specified algorithms.

Concerning time-efficient computations, reducing trace size already during tracing can help. To this end, we have already introduced filters in the previous chapter.

## 8.2 Related Work

Here we describe algorithms which have been proposed in literature for the failures and potentials we are interested in. We present solutions for Java but also for other widely used object-oriented programming languages.

### 8.2.1 Deadlock Detection

Detection of cyclic dependencies in a trace of a concurrent program is not a hard problem. This explains why there is not a lot of literature about languages like Java while at the same time many commercial tools implement it.

#### Operating Systems Support

Deadlock detection algorithms have a long tradition in operating systems in order to detect deadlocks of *processes*. Most often, the Banker Algorithm and reducibility of resource dependency graphs are cited [Tan97].

These two algorithms can be used for deadlock detection but also for computing deadlock free resource allocation orders. They work from the assumptions (i) that it is completely known, which resources will be needed by a process, and (ii) that a resource allocation system can decide to which process a resource is granted first.

This assumption does not hold for Java locks granted to threads. A lock is granted when the thread acquires it and when it is free. If different threads are claiming a lock, one is arbitrarily chosen. Therefore, we cannot simply implement a banker algorithm or the resource graph reduction algorithm to detect Java deadlocks or to detect potentials.

**Commercial Tools for Java and Similar Languages**

Deadlock detection for Java can be found in many commercial tools, for instance JBuilder [JBd01], JProbe Threadalyzer [JPr00], and Assure Threadanalyzer [Ass] for Java. Note that these tools do not cover failures similar to the deadlock like lockout or join-induced deadlock. The above mentioned commercial tools do not publish the concrete implementation of their deadlock detection algorithms. In these tools, deadlock detection is often combined with detection of deadlock potentials which will be discussed in the next section.

We have already mentioned in the introduction that COMPAQ observed that users of the pthreads library did not have appropriate support for concurrency problems. Therefore, COMPAQ built Visual Threads [Har00] to provide runtime analysis of deadlocks, deadlock potentials and race problems. The definition of a deadlock is specific for pthreads. It covers deadlocks involving mutex locks, joining, and read-write locks as found in the pthread library. For the deadlock detection, Visual Threads requires that at runtime blocking can be traced. This is the same observation we have made.

The deadlock detection algorithm identifies cycles in a thread dependency graph. This graph contains thread nodes and directed edges, meaning that one thread depends on another for its progress. The algorithm starts only if a thread changes from running from blocking. This blocked thread is uniquely marked and then the algorithm recursively visits all threads on which the thread depends. For each new thread visited it is checked if the thread has already been marked with the same value. If so, a cycle is detected. Otherwise, each thread visited is marked with the same value. Assuming 32bit integers as marks, the marks are guaranteed to be unique and do not have to be cleared. With this algorithms not only a cycle involving the start thread can be detected but also a cycle which can be reached from the start thread. The complexity of this algorithm is not given in [Har00]. Assuming that a thread has at most one dependency edge to another thread and that a thread can be at most once in the dependency graph, the complexity of detecting a cycle is linear in the number t of threads, i.e. O(t). The complexity for maintaining the dependency graph cannot be determined without knowing more about its implementation.

### 8.2.2 Deadlock Potential Detection

Here, we describe the solution found in Visual Threads [Har00] and in Java Pathexplorer JPAX [Hav00, HR01].

JPAX [Hav00, HR01] has a facility which checks traces against temporal

logic formulas, and more interesting for our context, which checks for the potentials of deadlock and data race. The deadlock potential detection algorithm called *GoodLock* [Hav00] constructs a lock hierarchy for each thread from information about a running programming.

Then the lock hierarchies of different threads are compared for pairs held at the same time but which were acquired in opposite order, and, most importantly, which are not controlled by a preceding commonly used lock. This presents an improvement over other algorithms which only detect pairs and hence produce false positives. The presented implementation only deals with deadlock potential between two threads.

Therefore, a subsequent publication described an algorithm which can detect deadlock potential between many threads [HR01]. Two data structures are maintained by this algorithm: a thread map keeps track of which locks are owned by any thread at any point in time. The second data structure, a lock graph, maintains an accumulating graph of all the locks taken by threads during an execution, recording locking orders as edges. That is, an edge is introduced from a lock 1 to a lock 2 in case a thread owns 1 while taking 2. If this graph ever becomes cyclic the program has a deadlock potential. The complexity is not given. As the algorithm is not exactly given, we cannot give a precise assessment.

It has been noted by [Hav00] that deadlock potential detection in traces using a model checker suffers from the state explosion problem. Local deadlocks, i.e. deadlocks which are only possible due to a part of the lock hierarchy, cannot always be found. Therefore, the detection of potentials using a dedicated solution is more feasible and to this end the GoodLock algorithm was proposed.

Visual Threads also provides detection of deadlock potential [Har00]. The algorithm is only coarsely sketched. It detects inconsistent lock acquisition order by maintaining a set of *must-not-be-locked-before* relationship pairs. When a new lock is acquired a search is performed to find any existing lock order pairs involving the new lock and each of the other locks already held by the thread. The must-not-be-locked-before relationship is transitive and the search recursively follows chains of relationships. If the search fails to find any inconsistencies with previous execution behaviour, then new must-not-be-locked-before relationship pairs are created using the new lock and each lock currently held. This algorithm does not avoid false positives as a consequence of gate locks. The complexity is not given with the algorithm.

### 8.2.3 Failures and Potentials Involving wait() or join()

For other failures such as the ones described by [Lea00] and the ones we have determined we could not find any algorithms in the literature. The reason might be that they are very specific for the Java monitor concept and that some of them are very similar to the deadlock because they involve cycles. For these failures, other algorithms can be easily adapted. Also, for the corresponding potentials no literature could be found.

## 8.3 JAVIS-Algorithms for Liveness Failures and Potentials

In this section, we describe our choice for a cycle detection algorithm with the example of a deadlock and we propose an algorithm for detecting missed notifications.

### 8.3.1 Cycle Detection

All failures involving cyclic dependency based on blocking, joining and waiting can be detected in the same way. A dependency graph is constructed and then the algorithm detects cycles in the graph. Here we describe the idea exemplarily for deadlock detection.

#### Graph Structure

Dependencies exist between threads and monitor objects or between threads. Threads and objects become nodes of the graph, locking and blocking is represented as edges.

The exact definition of the graph is as follows. Nodes are of type *thread* or *object* and edges of type *lock* or *acquire* which can be inserted between a thread and an object only. A weight for re-entrance of threads is added to the data structure for each locking edge. Otherwise one would not know when to delete a locking edge. Note that the graph is not necessarily fully connected.

A cycle is a sequence of alternating locking and acquire edges with one node being the start and the same node being the end. It may start from a thread which is connected by a locking edge to an object which is connected by an acquire edge to a (different) thread which is connected by a locking to a (different) object which is connected by an acquire edge to the first thread.

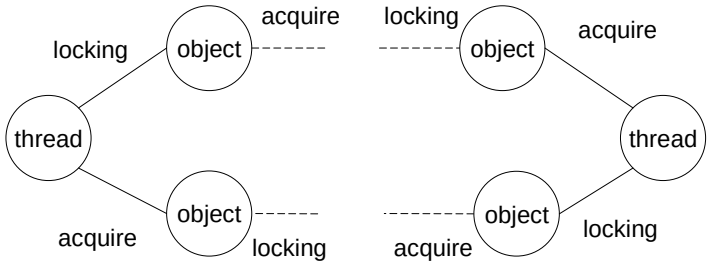


Figure 8.2: Cycle in Graph Structure

Also more object-thread pairs may be involved. This situation is depicted in Fig. 8.2.

### Algorithm

The algorithm has two main tasks.

- It creates and deletes thread and object nodes, and locking and blocking arcs as the program proceeds (or the the trace is read).
- It checks for cycles starting from a thread node whenever a new blocking edge is inserted.

For *maintaining* the graph, the algorithm inserts an acquire edge for a blocked call. For a synchronised method entry it inserts a locking edge if no exists and sets the weight to one. If one exists it increments the weight by one. If the thread or the object node do not yet exist they are created as well. When a locking edge is created it is checked if there has been an acquire edge which is then deleted. For a synchronised method exit the weight of the locking edge is decremented. If the locking weight reaches zero, the edge is deleted. If the object has no other edges it is deleted. A thread is only deleted after method exit for `run()`.

The cycle *detection* is triggered when a new acquire edge is inserted. For each thread it is checked whether it has an acquire edge. For the online algorithm the monitor status can be checked with the JVM which requires that all thread reference provided by the JVM are kept until a thread exits. For the offline algorithm the existence of a reference to the acquire edge has to be checked.

1. Determine all threads with an acquire edge.
2. Take one and follow its acquire edge to the object.

3. Repeat the following until there is no such edge:

   If the object has a locking edge, follow it to the thread. Then if the thread has an acquire edge, follow it to the object.

4. Check if the last thread reached is the same as the starting thread. Then a deadlock is detected.

5. Delete the thread from the determined threads.

6. If a deadlock was detected also delete all other threads involved from the determined threads.

The detected cycles have to be stored, e.g. in the trace file.

### Complexity

Here we determine the time complexity, The complexity depends on the number of threads t and the number of objects o. Space complexity also depends on t and o and is not considered an issue here.

The complexity of a *maintenance* operation is as follows. Finding the thread in a list can be accomplished in linear time O(t). The worst case is that the thread is not yet part of the list. In the case of a successful locking the object may not yet be part of the graph yet. Then the insertion of the object and the lock-edge has complexity O(1). If the object exists, the thread has a locking edge which is also computed in O(1). In the case of acquire an edge has to be inserted. The object already exists. If the objects are kept additionally in a list it takes O(o). If the object can only be found via threads, the complexity is O(t) plus O(n) because an object can be reached through only one thread. Deletions are analogous. The overall complexity reached by best practice is therefore O(t).

The complexity of the *detection* part is as follows. Each thread has to be checked for the existence of the acquire edge which takes O(t). The fan-out for leaving a thread by an acquire edge and for leaving an object by a locking edge is at most one. Therefore, following one edge is O(1). In the worst case all threads are involved in the deadlock and therefore following the cycle back to the starting thread takes O(t). Therefore, the overall complexity is O(t).

If used online, the algorithm is triggered each time a new acquire edge is inserted. As a result, many edges are visited again and again. This could be improved in an offline variant, marking already visited paths. For a proof-of-concept prototype this is considered efficient enough.

#### Implementation

We have implemented the online variant in the Analyzer class of the tracing system (see Fig. 7.5). When a deadlock is found, the cycle found is added to the trace file, and the tracing is stopped. The implementation is described in more detail in [Wey01].

#### Comparison

The cycle detection does not have the same complexity as the known algorithms for detecting arbitrary cycles. Our algorithm only checks whether a given node is involved in cycle and there is at most one outgoing arc from the node which can lead into a cycle. Therefore we do not need marking such as in [Har00]. If the algorithm should detect cycles which do not involve the starting thread but also a cycle which can be reached from the starting thread we need to keep track of each thread visited or we need to employ a constant marker during one search.

### 8.3.2 Missed Notification

Remember that a missed notification is a thread which cannot return from wait() because there is no subsequent notification but there was a notification before this wait(). That is, when we detect a thread waiting for a long time, we can start to search for a missed notification. To this end, the analysis component should have a configurable timer.

A missed notification is a notification which did not have an effect, because it happened too early. Instead of looking only for one such notification we propose an algorithm with which we can find also other notifications which did not have an effect. These can be seen also as potentials for missed notifications.

For detecting an ineffective notification we propose to tackle the problem by counting wait()- and notify()-calls for each object. A counter maintains the current balance between these calls. There is a counter for each Java object.

- The counter is initialised with zero.
- A wait()-call increments the counter by one.
- A notify()-call decrements the counter by one. If the counter is set to a number smaller than zero, this means that a notify() did not wake up any waiting thread. In this case a warning should be issued pointing to the notify()-call causing the warning. After that the counter is set to

zero, because notifications cannot be stored and hence one has to start counting the balance again.

- A notifyAll() sets the counter to zero again if if was positive because all threads are woken up. If the counter was already zero, a warning is issued because no thread could be notified. The counter remains zero as in the previous case.

This algorithm could be used online and offline. The algorithm involves for each call to wait, notify and notifyAll an integer operation and comparison. A counter is only created when the Java object is first used as a lock.

### 8.3.3 Potentials for Cyclic Dependencies

For the deadlock potential and other potentials of cyclic failures we do not have to provide a new algorithm as we think that the solution by [Hav00, HR01] is feasible also for our trace-based approach. The graph used for deadlock detection can be extended to store lock hierarchies.

## 8.4 Summary

In this chapter we have given an overview of algorithms for detecting concurrent liveness failures and potentials in traces. We have adapted a deadlock detection algorithm which works with our trace format and whose results serves as input for the visualisation described in the next chapter. We have also described a new algorithm for detecting missed notification and potential for missed notification.

The deadlock detection has been implemented in the JAVIS prototype. A first version of the analysis was described in [Wey01] and [Meh02].

# Chapter 9

# Trace-based Failure and Potential Visualisation

In this chapter we describe how the traces and the analysis results from failure detection are visualised in the JAVIS environment. Based on the trace format from Chapter 7 and based on the detection algorithms from Chapter 8 we can now derive requirements for the visual language and also for the tool environment in which the visualisation will be generated.

After refining the requirements we will discuss related work. We will discuss visual languages which can be found in tools for visualising the execution of object-oriented and concurrent programs and we will discuss the visual modelling language UML (Unified Modeling Language) [UMLa] which has been introduced to support modelling at the different phases of the software lifecycle. We will also discuss visualisation environments and visual language editors.

We will then present a languages for trace and failure visualisation which extends the UML. We will present a visualisation environment which can translate traces in our trace format and results from failure analysis into this language and display the visualisation.

## 9.1 Visualisation Requirements

The visualisation depends primarily on the analysis results, which it has to visualise, see also Fig. 9.1. If the failures are visualised in isolation from the surrounding execution they may become less understandable. Also, there is not really a reason why one should not aim at providing also a visualisation of the trace, not only of detected failures.

Therefore, we require that the failures can be visualised embedded into

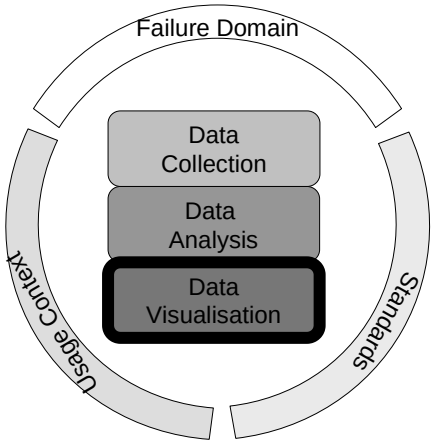


Figure 9.1: Requirements for Data Visualisation

the trace visualisation and separately in a dedicated view. Also, the trace view and the separate failure view should use a very similar language in order not to create a barrier for understanding when switching between the two.

We detail the requirements for the trace visualisation language in the next section. In the two sections to follow the next, we detail the requirements for the failure visualisation language and discuss requirements for the environment which is supposed to generate visualisation in these languages.

### 9.1.1 Trace Visualisation

The starting point for the trace visualisation is a textual trace such as the one given in Fig. 7.4. We require **accuracy**, i.e. the visualisation should be an equivalent presentation of the textual trace without loosing any of the information from the trace.

The textual presentation itself has drawbacks which can be easily overcome by **graphical** presentations. Text is linear and cannot display other dependency dimensions than the sequence of its elements. Other dependencies are only implicit, e.g. in the syntax or in the values. A graphical visualisation can present easily two dimensions explicitly, e.g. according to an x and y axis. An axis can also be organised in a spiral shape.

Additional dimensions can be encoded for instance by colour or shape. Colour and shape are graphical patterns which can be easily discriminated. Adequately designed, a graphical visualisation can convey selected dimension better than text. Therefore, we require a graphical visualisation. It can be

an optional feature to view the textual trace along the side.

Here, we aim at a **2-dimensional** graphical visualisation. While viewing 2D is familiar, e.g. using horizontal and vertical scroll bars, 3D may have acceptance problems, because it is typically not present in IDEs and requires specific familiarity and experience with 3D viewing such as in VRML (Virtual Reality Markup Language) environments.

A trace contains the dimensions time, objects, links and dependencies, and method calls. It is not possible to squeeze them all into one visualisation without the danger of overloading it. Therefore, our requirements for the visualisation have to prioritise these dimensions.

Traces are collected over time and therefore, time plays an important role for trace visualisation. It should have an explicit dimension in the visualisation which supports the notion of happened before or happened after. This is only possible if **time** is mapped to an explicit axis, e.g. x or y **axis**. We prefer an explicit time axis over animation. During animation, displayed information may be erased.

The other **axis** should be taken by the **objects** and the **method calls** between them. By objects we refer to thread objects and passive objects found in the trace. It is important to show passive objects in addition to threads, because of their key role in concurrent behaviour when they are shared by different threads. Either they are accessed without synchronisation, potentially leading to safety problems, or they are access mutually exclusive, which may lead to blocking, or they are used by threads for sending each other signals, e.g. by notification. Method calls are more important than the links and dependencies between the threads and objects for the general understanding of an execution. While we neglect the dimension of object links, we also have to consider the dimension of **concurrency**. Concurrency is related to time and expresses itself in the method calls. While it is already implicit it can be supported by colouring the different threads of control.

It is desirable to have also another **optional** trace visualisation which can show the other dimension of **links and dependencies** although this is difficult because they are changing over time.

The visualisation must also respect that the trace file may also contain a **call stack** at its beginning and must provide an adequate visualisation seamlessly integrated into the visualisation of the real trace.

Also, it is desirable to use an existing visual language and to use a language which is already used in the software development lifecycle such as the **Unified Modeling Language (UML)** [UMLa]. We have already argued that this language is known to many programmers which have to read UML design documents and that by using UML in debugging we extend the seamlessness in the use of notations. The UML contains **Interaction Diagrams**

which seem to be a candidate matching our requirements.

Using UML may conflict with the requirement that a general purpose trace visualisation should be **scalable**. It should be possible to visualise traces with many method calls or/and many different objects. UML diagrams can become very large and then the question is how well they still convey information.

### 9.1.2 Failure and Potential Visualisation

Generally, the failures involve the same dimensions as the traces. Depending on the kind of failure or potential, either the time dimension or the dependency dimension is more important. The dimension of objects is always needed.

Firstly, we look at the failures for which **dependencies** are important. The comparison of the different kinds of failures showed that there is a group of failures, namely, deadlock, nested monitor lockout, circular join, self join, and join-induced deadlock, which have in common that the involved threads are blocked. The comparison has identified three different kinds of such blockings, *blocking* on a lock, blocking while *joining*, blocking while *waiting* for a notification. In all these states, the blocking threads are dependent on a resource or on another thread. These dependencies have already been documented with the well-known wait-for-graphs.

Now it is straightforward to require, that these kind dependencies are visualised in the failure visualisation. That is, we do not require a visualisation for each kind of failure, but we exploit the commonalities to define a language with which different kind of failures can be described. The cycles resulting from the dependencies involved in the failures should be obvious in the visualisation.

However, we remember that it was one of our motivations to go beyond the usual visualisation with a wait-for-graph because it does not inform about the execution history nor about the method calls involved. Therefore, we require that both, the dependencies and the **execution order** is depicted.

In addition, we have required that, in order to improve understanding, the trace visualisation and the failure visualisation should use the same language concepts, and that the failure can also be viewed in the original trace. That means, the visualisation of the detection should be **embedded** in the visualisation of the overall behaviour.

Secondly, we look at failures and potentials for which **time** is more important. This applies to the missed notification and to circular join. For them, we do not have to derive completely new concepts but we require that they can be visualised as traces with the involved calls highlighted or with

the other calls excluded. Also for the deadlock potential it is adequate to show the lock accesses and releases over time.

### 9.1.3 Visualisation Environment

Here we list the functional and the non-functional requirements for the visualisation environment.

#### Functional Requirements

The main functionality is to read a trace from file and visualise it, and to visualise detected failures.

Either failure detection has been carried out before the visualisation, online or in a separate step. Then results are part of the trace or are in a separate file. Or the analysis can be triggered from within the visualisation, for example while reading the trace file.

It must be possible to view the failures as part of the trace and it must be possible to extract them in their own views.

The visualisation should be fully automatic, e.g. including the layout. We require that a trace or a failure can be displayed as a whole or stepwise, where a step corresponds to a method blocking, a method entry, or a method exit.

Different focusing techniques are also desirable, especially regarding the failure, but also a general zooming facility.

Storing the generated visualisations, exporting them and printing are also desirable features.

#### Nonfunctional Requirements

Such an environment can be built in different ways. One can use a framework, e.g. for a graphical editor, or one can use an extensible graphical editor. Also, the solution can be integrated in a standard IDE environment. Ultimately, it depends on what the developers typically use, which may vary from standard IDEs to UML round-trip tools. There is no tool which will find a high acceptance with all developers. Therefore, a special approach is not required.

## 9.2 Related Work

As we are interested in the behavioural aspects of the traces not in the structural information contained in it, here we discuss only approaches to

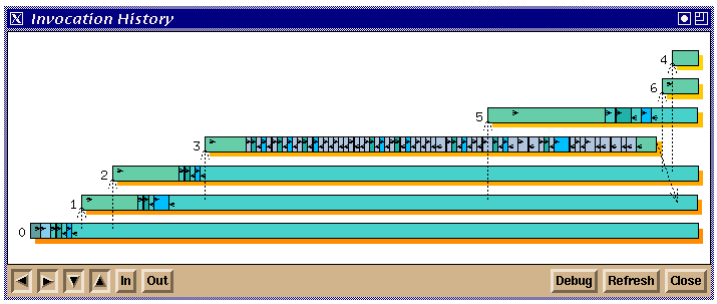


Figure 9.2: GThread History View

dynamic visualisation, i.e. visualisation of the runtime behaviour. We discuss approaches for concurrent and object-oriented programs.

### 9.2.1 Visualisation of Concurrent Programs

**GThreads** was one of the first tools for visualising the execution of concurrent programs [ZS95]. It instruments source code using the pthread library to trace calls to the library. In order to trace arbitrary function calls, the programmer manually has to insert tracing methods in the code. Traces are output as files.

The post-mortem visualisation is an animation. It is implemented with the POLKA [SK93] animation toolkit, primarily intended for algorithm animation. Animation is speed-controlled and can be executed stepwise. Here, we primarily discuss the used visual languages. We do not consider animation to be an central feature for our goals. It has disadvantages for large traces, also the information displayed can be erased and therefore we prefer an explicit presentation of time.

GThreads contains an animated call graph, a history view, a mutex view, and a barrier view. For us the history view and the mutex view are of interest. The function view is neither object-oriented nor does it incorporate synchronisation-specific information. This kind of information can be found in the mutex and barrier view. The barrier is a synchronisation not supported by Java and therefore we do not consider it here.

The history view provides an overview of the trace (see Fig. 9.2). It shows exactly all function calls. A bar describes the execution of a thread, it contains segments which denotes the different, nested function calls. The shadows of the bars have all different colours and denote different threads. This view is feasible for imperative programs but not for object-oriented programs.

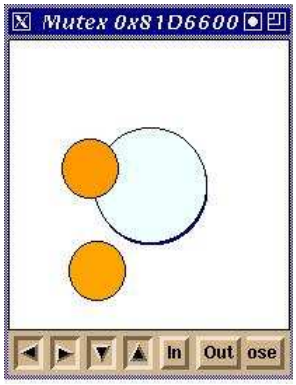


Figure 9.3: GThread Mutex View

The mutex view depicts a mutex lock with a bigger dice and threads using it with smaller dices (see Fig. 9.3). A thread holding the lock is depicted inside the big dice. A thread waiting for a lock is depicted outside the big dice. For each mutex there is such a view. Deadlock can only be detected by comparing different views.

### 9.2.2 Object-Oriented Visualisation

#### Jinsight

IBM **Jinsight** is an industrial research prototype for visually exploring the runtime behaviour of a Java program. It is the most prominent tool for visualising Java execution. Here we refer to the last published version 2.1 [Jin, PKV94, PKV98, PJM+02]. Thereafter, the concepts of Jinsight have been introduced into commercial tools like IBM Websphere [WbS] or the IBM Hyades plugin for Eclipse [Hya].

Jinsights helps in understanding a program's time performance, in understanding a program's memory usage and anomalies such as memory leaks, and in understanding a program's data structures and its bottle necks by understanding how objects are created and interrelated. Its scope is concurrent Java programs. Jinsight traces method calls, object creation and deletion. The tracing approach has been described in chapter 7 in depth. Jinsight provides post-mortem visualisation.

Jinsight provides three kind of views:

- views depicting the execution at different level of details,
- views depicting statistical summaries of the execution, and

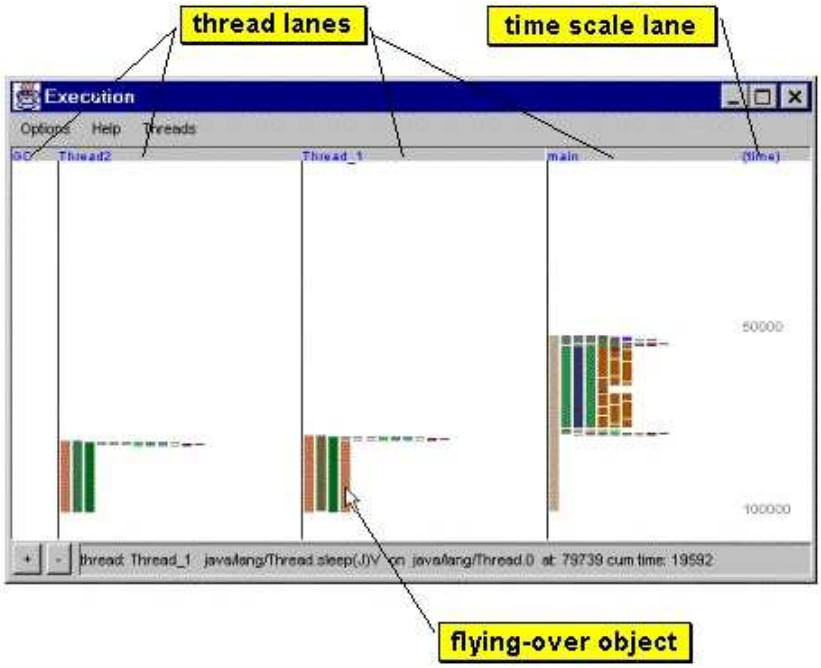


Figure 9.4: Jinsight Execution View

- views depicting patterns which emerge during the execution.

For us, the **execution view** is the most interesting. It allows the exploration of the detailed program execution sequence per thread. It starts by an overview (see Fig. 9.4) depicting threads in columns and depicting nested method calls of each threads by vertical coloured stripes stacking to the right. The colour indicates the object. Time progresses from top to bottom. The view provides pop-up information about selected calls or threads. It allows the zooming into the picture until the level of individual method calls is reached (see Fig. 9.5).

At its most detailed level of presentation, the visual language used in the execution view has similarities with UML sequence diagrams. It is however possible to compress this view by omitting details and therefore the language used is scalable. Jinsight allows the user to customise views by slicing, grouping, and filtering information.

Although threads are covered, this tool is not thread-specific. Liveness failures can only be detected using the heuristic that a thread from the execution view fails to make progress.

Jinsight provides also views showing dependencies between objects. The **reference pattern view** (see Fig. 9.6) supports the exploration of patterns

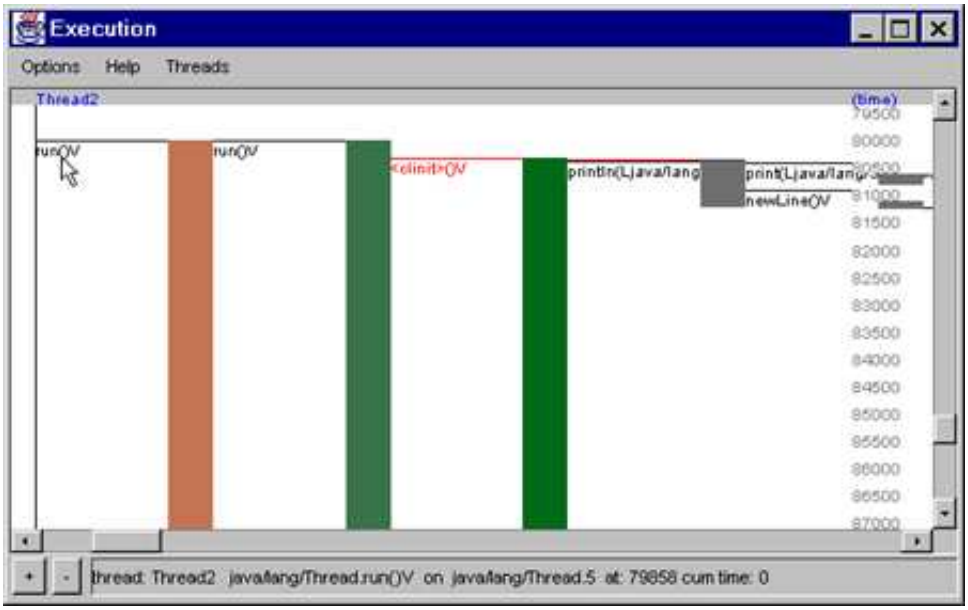


Figure 9.5: Jinsight Execution View - Zoom

of references to or from a set of objects, in varying detail. It is intended for studying data structures and finding memory leaks. The view shows references to an object at different points in time.

We do not show the views providing summary information and the views for identifying patterns of interactions emerging as the program executes, as they cannot help with our purpose of highlighting erroneous behaviour as detailed as possible.

Jinsight as a visualisation environment is characterised by the fact that it is an implementation for exactly the trace rsp. dump format. It is not open source and it is also not intended to be a framework. The user control is characterised by the fact that the user can choose between different views and in addition can choose projections called slices to work with in separate projects called workspaces. We think that it is important that the views are from the two orthogonal dimensions behaviour and structure, i.e. diagrams with time axis and diagrams for object structures such as the reference pattern view. We also think that techniques for focusing such as the zooming facility are very important for a user to work with such a tool.

### 3-Dimensional Visualisation

For instance, [Rei98] uses 3D diagrams for displaying object lifetime, call graphs, and object interaction. The Software Tomograph [LS02, SOT] uses

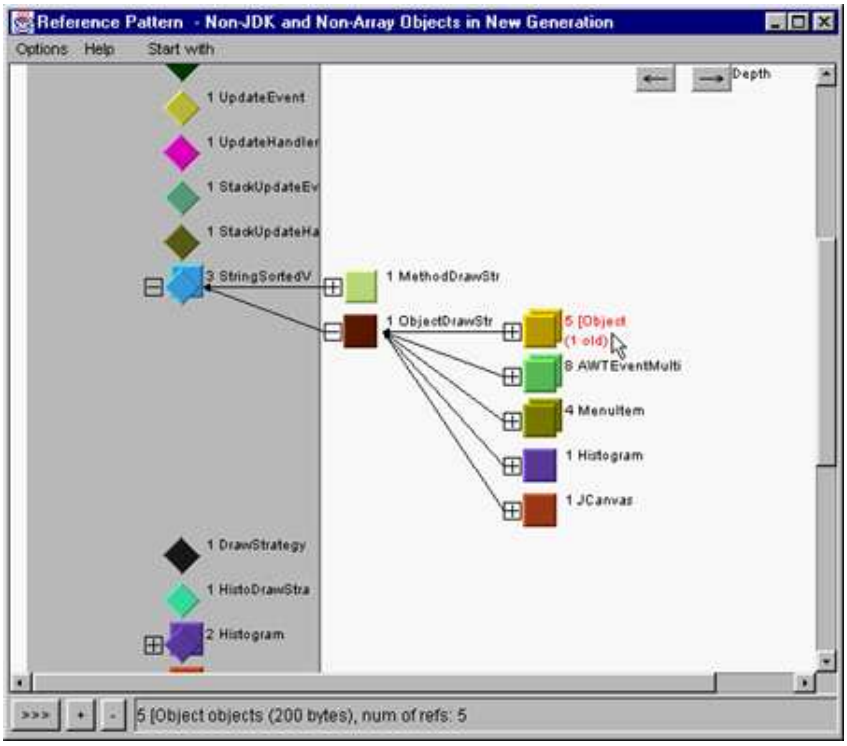


Figure 9.6: Jinsight Reference Pattern

the third dimension to map metrics to distances between classes in a 3D space.

Note that the 3D consideration do not mean that UML must be excluded from consideration. The UML specification does not prevent showing several UML diagrams in a 3D space. There is work about 3-dimensional class diagrams [MLMD01] but not about 3-dimensional interaction diagrams.

We have not yet considered 3D when visualising single traces. The third dimension could for instances be used to show how links and dependencies change overtime.

Changing dependencies and links can only be observed in the stepwise replay of the collaboration diagram. An alternative could be interesting but with the focus on failures we reckon that the user is only interested in the final result of the cyclic dependencies. For reconstructing the behaviour over time the user can view execution over time in the sequence diagram, which is more compact than a sequence of structural snapshots. Also, this sequence would need to depict the method call sequence, and lastly it seems that the call sequence is the key into understanding behaviour over time. Besides this, the third dimension could be used for additional information such as for

comparing several traces or for visualising the dichotomy between structure and behaviour.

### 9.2.3 UML-based Visualisation

For visualisation of structure and behaviour of software it is also possible to consider visual modelling languages from software engineering. Today, the most widely used modelling language is the Unified Modeling Language (UML) [UMLa].

Typically, UML is supported by UML CASE (Computer Aided Software Engineering) tools which provide editors and also code generations facilities. Tools to visualise structure and behaviour of existing software with UML come from at least two different areas, reverse engineering and program comprehension.

Before we describe the use of UML in these areas we will give a short overview of the UML.

#### UML

The **Unified Modeling Language (UML)** [UMLa] has become a de facto standard for visually describing artifacts of the software lifecycle, covering software requirements, analysis and design documents, program documentation, test specification, and also activities and processes of the software development itself. The UML provides a set of diagrammatic languages and a constraint language (OCL) which helps in specifying information which cannot be specified visually.

In the context of this thesis, the diagrams for describing behaviour on instance level are of interest, namely **interaction diagrams**, which have two possible representations, as **sequence diagrams** and as **collaboration diagrams**. While a sequence diagram depicts objects on one axis and method calls along an explicit time axis, a collaboration diagram depicts objects and its links in a two-dimensional space and depicts messages on links with a number indicating their order. What is interesting for concurrency is that interaction diagrams support the notion of an **active object** which owns a thread of control.

In the context of this thesis, not only the visual languages for describing software behaviour are of interest but especially the possibilities to extend them for special domains. The UML is designed to be adapted to domain-specific requirements by defining **profiles**. A profile is a set of refinements of existing modelling elements. The UML already provides a set of standard

profiles, though none for Java execution. We will provide details on the profiling mechanism when we propose our UML-based visualisation.

### UML CASE Tools

The UML Language is supported today by UML CASE tools such as Borland **Together** [Tog]. These tools provide UML editors, code editors, debuggers, and support round-trip engineering or at least code generation. In round-trip engineering, the model is used to generate corresponding code fragments in an object-oriented programming language. Typically, only classes, fields, and method headers are generated but not method bodies. Also, the tools can generate UML class diagrams from the source code. When the code or the model is changed, it is reflected in the model and in the code. Existing round-trip concepts are fairly limited covering mainly structure but not behaviour. For instance, Together can generate an interaction diagram from a method body but subsequent changes in the code or in the sequence diagram are not reflected. The sequence diagram can not be used for code generation.

Together can import UML diagrams via XMI, however with restrictions, and also via a proprietary API by which a UML diagram can be generated and also be extracted again. Together supports UML extension mechanism like stereotypes and tags.

There are many similar tools, e.g. Rational Rose or Argo UML. These tools demonstrate, that UML-based modelling and programming are moving closer. Programmers often have to know the UML.

### UML-based Code Generation

Except for the STATEMATE [HP98] CASE tool and similar approaches which generate executable code from statecharts, code generation in the previously mentioned commercial CASE tools is limited to generation of class fragments containing no behaviour.

An extension to include code generation from behavioural UML diagrams is for instance proposed in [EHSW99]. In order to generate executable Java code fragments from collaboration diagrams, the use of standard elements has been restricted, e.g. destructors are not allowed because they have no counterpart in Java. To this end, the UML meta-model is refined. The approach does not cover thread synchronisation.

### UML-based Reverse Engineering

Reverse engineering aims at supporting the maintenance of a software system. It aims at deriving the design of a system from code. In the past, the focus

was mainly on structure but now there exist also dynamic reverse engineering approaches.

UML has been used since its existence in reverse engineering of object-oriented systems, mainly visualising structure with class diagrams, replacing the predecessors of UML. Today, most UML editors support the generation of static structure UML diagrams from source code such as Java, e.g. Together [Tog]. Here, we are focusing on dynamic reverse engineering of object-oriented systems. The academic prototype **Shimba** [Sys99] generates sequence diagrams using a proprietary tracing facility. They are however only an intermediate step for deriving statecharts from several different traces. This tool does not support concurrency and therefore the sequence diagrams are not suitable for our purpose.

The approach by [KG01] describes a **meta-model of Java behaviour** in UML. This model is used to extract facts about behaviour such as a method invocation from the code. Then rules are used to generate UML-conform collaboration diagrams from the extracted information. The proposed Java meta-model does not support concurrency.

### Program Comprehension using UML

Program comprehension aims at supporting the understanding of an existing program. Tools for program comprehension aim at supporting the process of a human to build its own mental image of the structure and behaviour of a piece of code, or of aspects thereof, either starting from code or from its execution. Because of many different conceivable purposes, tools differ a lot. Tools like debuggers or profilers are typically not considered program comprehension tools although the borders are not clear. Often tools are also simply called software visualisation tools, a term which hides even more the purpose of a tool. It is not our aim to give a precise definition of these terms but merely to indicate areas of related work.

The academic prototype **JavaVis** [Oec02] aims at improving the understanding of small-scale Java programs for educational purposes. It supports stepwise execution of a program and visualises the execution in a sequence diagram. The tool supports threads but does not support blocking or locking. It does not produce a trace but only incrementally extends the displayed sequence diagram.

The academic prototype **Jacot** [LRRM03] also generates UML sequence diagrams from a running Java program and shows different thread states in a simple statechart. The visualisation is online and does not generate a trace. The tool aims at supporting the identification of liveness failures through the visualiation but does not provide automated detection. It supports multi-

threading including locking and unlocking. Blocking is not visualised.

### 9.2.4 Comparison

#### Visual Trace and Failure Language

There are many different possibilities to visualise traces which allow to depict the order of events explicitly. Object-oriented traces, in particular, have the problem that they need to abstract from some of the dimensions involved in the trace in order to achieve a compact visualisation. Only the UML depicts object interaction explicitly, which is space-consuming. Therefore UML interaction diagrams are less compact than other trace visualisations. Other trace visualisations therefore are more scalable than the UML.

Scalability is not an issue because we primarily aim at visualising failures, i.e. only parts of a trace. In a concurrency failure, only a certain number of threads and other objects and especially method calls are involved.

We have seen very few failure-specific visualisations. Their drawback is that they use a very special visual language and are not integrated with the trace language.

For our purpose, single interactions are important, also in the trace, because they are the source of the failures we want to depict. We also want to use the same language for traces and failures. With UML interaction diagrams it can be especially useful, that they have two different representations. The sequence diagram is more suitable for a trace with many calls while the collaboration diagram is more suitable for depicting a few calls which provoke a failure, especially because it has a space for depicting dependencies between objects, which is currently used for links.

Therefore, for a unified approach to trace and failure views the UML seems to be apt. It can also be extended to depict Java-specific behaviour. Another argument is that developers often are familiar with the UML. Programmers need to be able to read UML design documents. Using UML also during debugging makes a step further towards seamless use of one notation and we do not have to introduce a new visual language.

Because UML is not primarily designed for visualising program traces it has to be adapted, which is made possible through the UML profiling mechanism. We are of the view that the UML can be extended to cover traces and failures sufficiently. Existing efforts to adapt UML to describe Java execution, either for the purpose of forward or reverse engineering, do not go far enough as they do not yet cover concurrency on the level of detail we require here.

#### Tool Support

Most visualisation environments are prototypes. There is no best way to integrate a new tool in the development process. Of course, it is advantageous to extend an existing tool, but processes differ a lot, and a tool used by some programmers may not be used by others. Also there is no clear decision how tight a new tool has to interoperate with existing ones.

We favour to extend a UML CASE tool. For implementing a prototype as a proof of concept, this kind of tool provides the highest level of reuse as we can use the UML editor, zooming, printing, exporting and so on.

Note that JavaVis and Jacot were built in parallel with the JAVIS prototype. They have very much in common with our approach from the UML point of view and from the technical point of view. Our purpose is more concise, supporting automated debugging, and our tracing and hence our viewing can contain more information.

## 9.3 JAVIS-Visualisation

In this section, we build on UML sequence and collaboration diagrams for trace and failure visualisation. Because they need to be adapted we explain the profiling mechanism in depth before we explain how we use it here. Then we describe how we extend the UML CASE tool Together [Tog] for visualising traces and failures.

The idea to visualise concurrent execution and concurrency failures with the Unified Modeling Language was the beginning of the research described in this thesis. We published our first version of a UML profile in 2000 [MW00]. Thereafter, we began with the design and the implementation of the tracing, the analysis, and the visualisation.

### 9.3.1 UML Profile for Java Traces, Failures, and Potentials

The standard way to extend the UML is by defining a profile which refines existing UML model elements. In this section we define a UML profile for concurrent Java traces and for liveness failures and potentials, which can occur in these traces.

#### UML Profiles, Stereotypes, and Tags

A UML **profile** [UMLa] is a stereotyped package that contains model elements that have been customised for a specific domain or purpose by extend-

ing the meta-model using stereotypes, tagged definitions, and constraints.

The principal extension mechanism is the concept of **stereotype**. It provides a way of defining virtual subclasses of UML meta-classes with new meta-attributes and additional semantics. A stereotype is a model element that defines additional values (based on tag definitions), additional constraints, and optionally a new graphical representation. A stereotyped model element receives these values and constraints in addition to the attributes, associations, and super classes that the element has in the standard UML. Extensions must be strictly additive to the standard UML semantics, i.e. they must not conflict with or contradict the standard semantics.

**Tag** definitions specify new kinds of properties that may be attached to model elements. The actual properties of individual model elements are specified using **tagged values**, either simple data type values or references to other model elements. A tagged value is a keyword-value pair that may be attached to any kind of model element. The keyword is called a tag. Both the tag and the value are encoded as strings. One or more pairs are enclosed in {}, separated by commata, e.g. {persistent = true}. In the case of boolean values, the value true may be omitted, e.g.{persistent}.

**Constraints** can be attached to any model element to refine its semantics linguistically. Constraints can be specified formally, for example by using OCL (Object Constraint Language) rules.

### Sequence and Collaboration Diagrams

In the comparison, we have already argued that sequence diagrams shall be used to depict traces and that collaboration diagrams shall be used for depicting failures involving dependencies because they are designed to capture dependencies like links. Depicting failures has been inspired by the well-known resource allocation graphs and wait-for-graphs, which were shown in chapter 3 and 5.

Note that we do not expect the sequence diagram to be extended with dependencies because it does not yet show any kind of dependencies, also no links. We are of the view, that the diagram which shows already specific dependencies should be extended for this purpose [MW00].

For visualising traces we rely on the following subset of standard elements of interaction diagrams:

- synchronous method calls (messages), method call parameters, method return, active objects, (passive) objects, lifelines, activation bars, shading of activation bars, multiple activation bars, links, and method call sequence numbers.

For depicting the complete trace format in interaction diagrams we miss:

- synchronized-method calls
- method calls found on the stack

For depicting failures in collaboration diagrams we miss:

- dependencies between threads and objects (locking and acquiring, i.e. blocking)
- dependencies between threads (joining and waiting)

### Tag Definitions for Method Calls

Here we provide a solution for distinguishing synchronized-method calls and method calls found on the stack from other Java method calls in a sequence or collaboration diagram.

Java methods which have a modifier synchronized are particularly interesting for our approach because they may result in locking or blocking, and when they return, in releasing a lock. Therefore, we want to be able to distinguish them from other methods. In the trace format, they are identified by the last entry which is true for a synchronized-method.

As we have already pointed out, the trace file may contain a call stack for each thread. The method entries and exits in this part have their method name labelled with <stack>.

For both cases, we will define new tags and tagged values to augment method calls with the additional semantics. Such a tag is not bound to a specific model element, e.g. the stack property could also be attached to objects, and the synchronized property could also apply to blocks of statements. Therefore, we do not define stereotypes for them. However, in our profile we limit the use of these tags to method calls.

We need to determine a suitable element to which this tag is added. A synchronized-method call in an interaction diagram maps to a model element **Message** from the **Collaborations**-Package. A message defines a particular communication between instances that is specified in an interaction. For the model element **Message** we define two tags, {synchronized} and {stack}, where the tagged values are of type boolean (see Table 9.1).

For an illustration, see the sequence diagram in Fig. 9.7. The first two calls were found in the stack, the following return was traced, and then a synchronized-call was traced, followed by two returns. So far, we have not yet shown a collaboration diagram. In Fig. 9.7, we also illustrate how the tags are integrated in a collaboration diagram, which is the equivalent of the sequence diagram.

| *Tag* | *Stereo-type* | *Type* | *Multipl.* | *Base Class* | *Description* |
|---|---|---|---|---|---|
| syn-chronized | none | boolean | 0..1 | Message | Attached to a call of a Java synchronized-method |
| stack | none | boolean | 0..1 | Message | Attached to a call of a Java method which has been found on the stack of the JVM when the trace was started |

Table 9.1: UML Profile for Concurrent Java Traces: Tags

### Stereotyping Dependencies

In collaboration diagrams, we want to show the dependencies between threads and objects which stem from synchronised interactions. Note that dependencies also occur if threads are not involved in a failure. Therefore, the extensions we describe here, are general extensions for describing concurrent Java traces. It is not necessary to provide only failure-specific extensions.

For each of the situations where a thread is waiting for an object or another thread we need an extension. In addition, a thread locking an object also has to be depicted as a dependency.

We need an additional model element for this because it is not a refinement of an existing element. Take for instance Fig. 9.7. The synchronized-call uses a link and creates a dependency between the main thread and the locked object. There is no link which we could tag for this purpose. Also semantically this would not be satisfying.

Because all dependencies need additional elements, we have defined them as stereotypes. The desired dependencies can occur between different kind of existing model elements, depending on its kind, between a thread object and a passive object, or between two thread objects.

In the context of an interaction diagram we consider the following three model elements as candidates from which we can derive suitable stereotypes. There is no perfect fit with our requirements, hence we try to determine the best trade-off.

- A **Link** is a tuple with a pair of object references. It is an instance of an association.

  We see the following arguments against stereotyping a link. There is not always a corresponding association and if there is one, it would have to be instantiated twice.

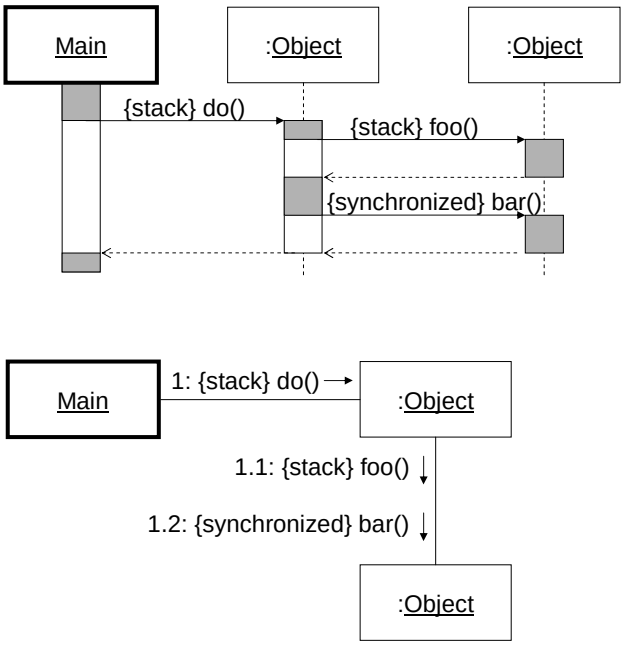


Figure 9.7: Tagged Values for Messages in Interaction Diagrams

- A **Dependency** captures a relationship other than association, generalisation, flow, or a meta-relationship. A Dependency is a directed relationship from a client (or clients) to a supplier (or suppliers), shown as a dashed arrow from client to supplier. It states that the client(s) depend(s) on the supplier(s), i.e. the implementation or functioning of the client(s) requires the presence or knowledge of the supplier(s). It indicates a situation in which a change to the target element may require a change to the source element in the dependency. A dependency does not require a set of instances for its meaning.

  As we are not taking a modelling perspective we do not really match the description of the dependency but we also have the intuitive notion that one thread depends on another thread or object.Also, it is useful that the dependency is directed.

- A **Derived Element** is one that can be computed from another one, but that is shown for clarity or that is included for design purposes even though it adds no semantic information. A derived element is shown by placing a slash (/) in front of the name of the derived element, such as an attribute or a role name. The details of computing a derived element can be specified by a dependency with the stereotype ≪derive≫ or by

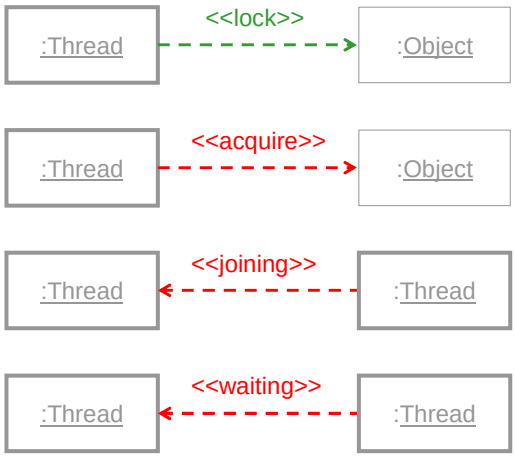


Figure 9.8: Graphical Notation for Stereotypes

simply placing a constraint near the derived element.

Locking, acquiring, joining and waiting dependencies are obviously derived from their corresponding method calls. This does not provide us with a model element for the derived element type. It can be an interesting option to show from which method call the dependency is derived but it also can be a visual overload. Also, these kind of relationships are obvious for the programmer.

Our conclusion is that we can use the model element **Dependency** to derive new stereotypes for locking, acquiring, joining, and waiting. We have used names slightly different from the corresponding method names.

Table 9.2 defines the corresponding stereotypes. Note that we have omitted the parent and tag column because none of our stereotypes is derived from another and none defines additionally tags.

Each stereotype has a graphical notation. They are given in Fig. 9.8. The first one has a green colour because it depicts a successful locking. The others are coloured in red to indicate waiting situations. Note that for locking and acquiring we draw the arrow from the thread to the lock. This resembles the direction of the method call. In the case of joining we draw the arrow from the joined thread to the joining thread because the joining thread depends on the other thread. In the case of waiting we draw the arrow from the thread which can send a notification to the waiting thread.

In Fig. 9.9 we illustrate, how the different failures involving cycles look using the stereotypes. Here, we have omitted the corresponding method calls.

| *Stereotype* | *Base Class* | *Description* | *Constraints* |
|---|---|---|---|
| Lock ≪lock≫ | Dependency | The lock dependency states that a thread is holding a lock after having called a synchronized Java method. | The dependency source must be an active object. From the active object there must be a at least one synchronized method call entry not paired with its exit. |
| Acquire ≪acquire≫ | Dependency | The acquire dependency states that a thread is waiting for a lock after having called a synchronized Java method. | The dependency source must be an active object. From the active object there must be a synchronized method call entry being blocked in the sequence diagram. |
| Joining ≪joining≫ | Dependency | The joining dependency states that a thread is waiting for another thread to exit. | The source and target must be active objects. The source must not yet have exited. |
| Waiting ≪waiting≫ | Dependency | The waiting dependency states that a thread is waiting for another thread to send a notification. | The source and target must be active objects. |

Table 9.2: UML Profile for Concurrent Java Traces: Stereotypes

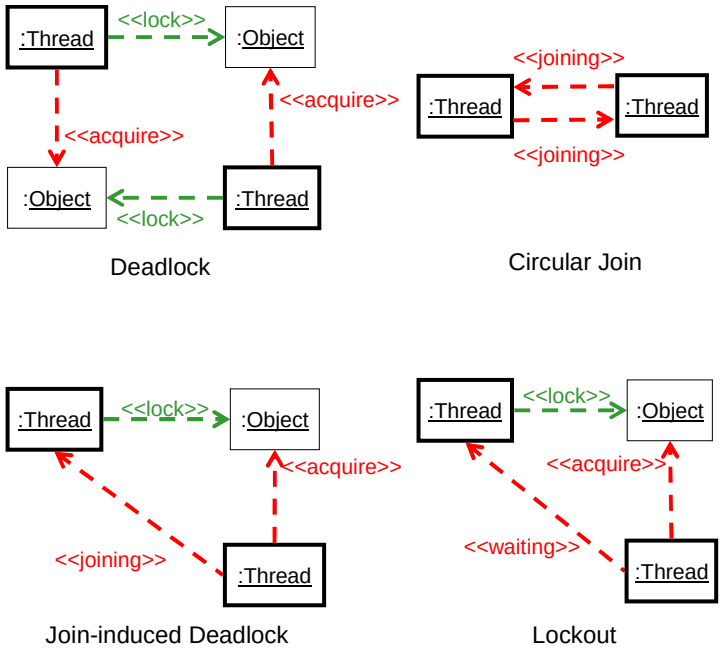


Figure 9.9: Cyclic Failures using Stereotypes

### Dealing with Blocking of Method Calls

As a last issue we discuss whether we really want to display blocked method calls as in the motivation.

The trace format distinguishes three kind of stages of method call, entry, exit, and blocking, which all have to be depicted consistently. Not being interested primarily into profiling issues such as how long a thread was blocked before it entered, we choose not to show this explicitly. Also, this kind of information would not really be exact as we do not have time stamps. We would only see how many other calls happened during the blocking. While not giving exact durations it could still be identified how many threads have been blocked at the same time at a lock, which helps in identifying bottle necks. We choose to replace blocking by entries when they exist, in both kind of diagrams. As we provide the possibility of stepwise visualisation, we do include an option to show the blockings until they are replaced by successful entries.

### 9.3.2 Visualisation Architecture

We implemented the visualisation of the JAVIS prototype as plug-in to the UML CASE tool Together [Tog]. A first version of this architecture was described in [Meh02].

Together provides two ways to import UML diagrams, either using the specialised XML format XMI or via an API. The API is easy to use and supports the construction of the same diagrams as the editor. As our traces are not encoded in an XML format we did see a need to transfer them into XMI.

In order to use the API we developed a set of Java classes (see Fig. 9.10). The PreParser presents summarised information to the user before displaying a trace. The Parser reads the trace and the Generator generates a corresponding interaction diagram using the API. Together interacts with our Java classes by calling a run()-method, which we provided. This method handles our user interface and controls our parser and diagram generator. The user interface contains several dialogs for configuring the reading and analysis of traces and for controlling their display.

Together provides the two different presentations of an interaction diagram automatically. In our prototype we have extended Together with the stereotypes for ≪lock≫ and ≪acquire≫ in the intended colours. Colouring individual threads of control could not be achieved. Together only supports colour for stereotypes.

## 9.4 Summary

This chapter discussed visualisation requirements, existing visualisations, and provided a UML profile for concurrent Java programs and concurrent Java liveness failures [MW00]. The profile was implemented with the UML CASE tool Together and is part of the JAVIS prototype [Meh02].

None of the existing tools using UML for visualisation provided extensions for concurrent Java and specifically the detailed extensions for describing thread synchronisation.

In the next chapter we will provide more insights into the Together plug-in by demonstrating its use with examples. Thereby, also the use of the UML profile will be demonstrated in more depth. The next chapter describes a complete usage scenarios starting with tracing, followed by analysis, and finishing with the visualisation.

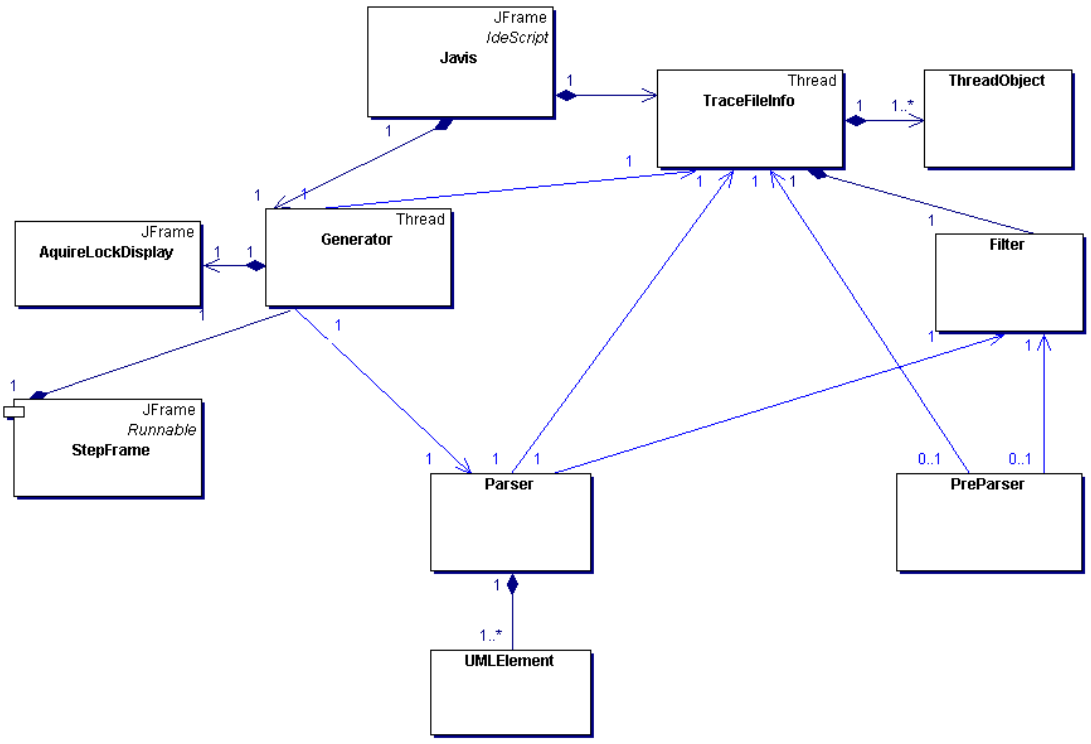


Figure 9.10: Class Diagram of the Together Visualisation

# Chapter 10

# Using the JAVIS Prototypes

In the previous three chapters, we have presented the technical details of the prototypical solutions for tracing, analysis, and visualisation.

In this chapter we present a visual tour of the prototypes at work. We demonstrate how they are intended to be used and how they work together by illustrating a typical use case with an example Java program. Thereby we will also introduce examplarily the user interface of the different prototypes.

After presenting the main example we will present a small experiment which demonstrates that the general purpose tracing and visualisation functionality also works for real-size software.

## 10.1 Example for Automated Failure Detection

Our main example use case is a non-deterministic test run. It is based on the banking application which was introduced in detail in chapter 4 and which had been used in parts already in the motivation.

We will illustrate the following steps.

- Running the application with the Java tracer (see Chapter 7).
- Automatic analysis of deadlocks of the running application with the option for deadlock detection (see Chapter 8).
- Generating different views of the trace and the deadlocks found with the Together extension (see Chapter 9).

In the following we will look again at the banking example and focus on a part of its behaviour.

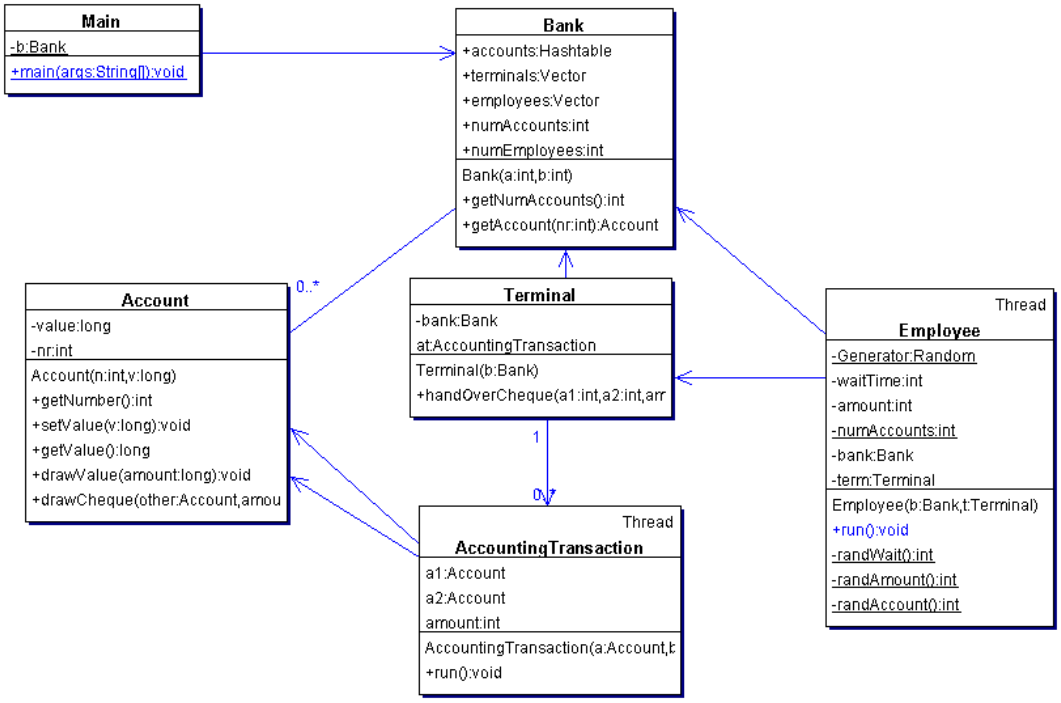


Figure 10.1: Banking Application Class Diagram

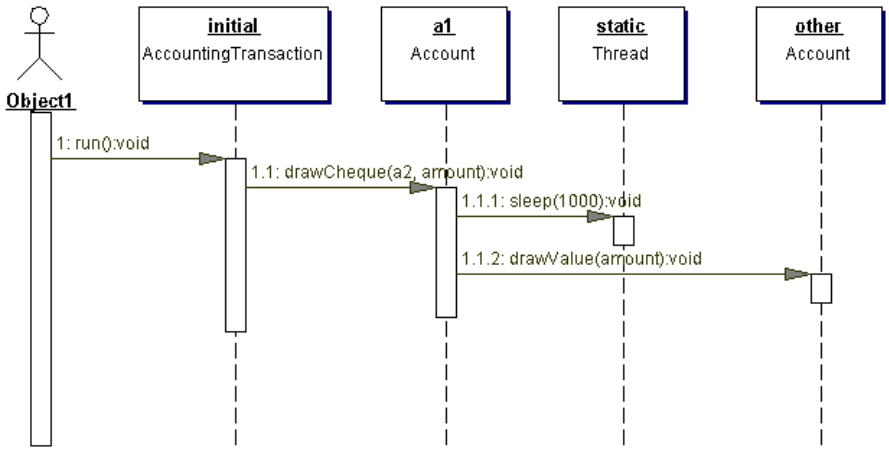


Figure 10.2: Sequence Diagram Generated by Together

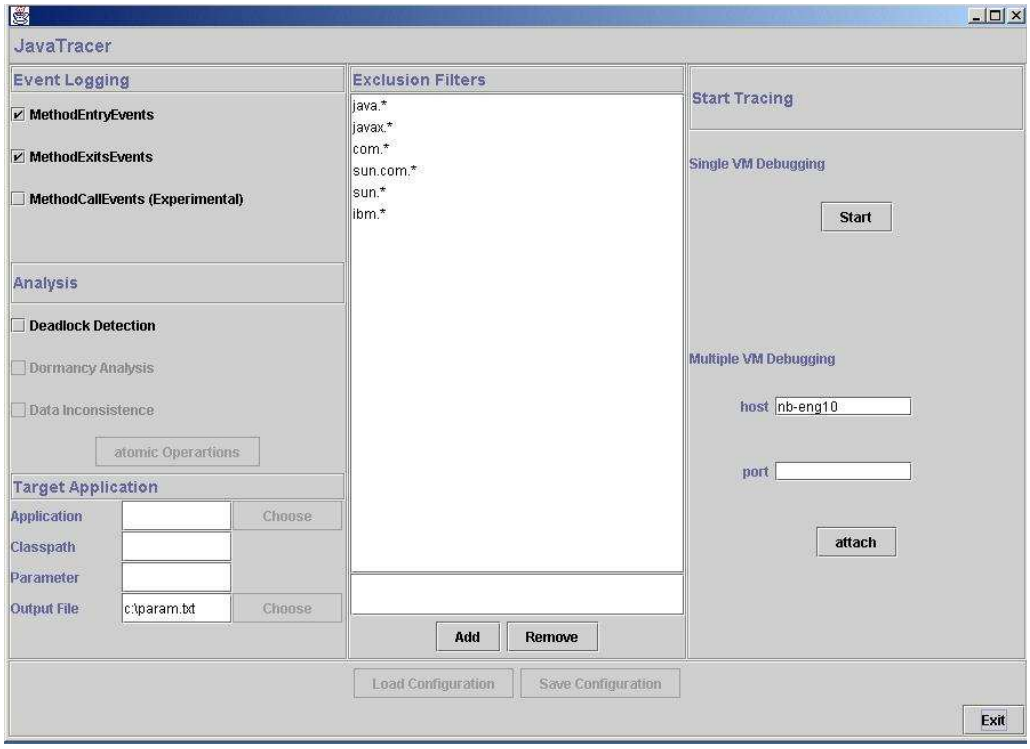


Figure 10.3: Trace Generation Dialogue

Then we will trace the behaviour and check for deadlocks. The result will be visualised in Together using our extension which can read traces and visualise them with our UML profile.

### 10.1.1 The Banking Example Revisited

The example is the banking application which has already been used in other chapters. Fig. 10.1 depicts the complete class diagram including methods.

For illustrating the behaviour of these classes we use the facility of Together to generate a sequence diagram. Note that this facility is not able to deal with concurrency. It can only generate single-threaded behaviour. The generated sequence in Fig. 10.2 depicts the behaviour of a thread transferring money between accounts. It accesses a first account, and from there it accesses the second account. Then it transfers the money from the second to the first. Note that Together does not show whether a method is synchronized or not. The two methods used to access the accounts are synchronized. In the program, several such threads work concurrently. As the code is a simulation, a delay has been inserted between accessing the first and the second

```
main:java.lang.Thread@1::Account@101:<init>:false:Enter
main:java.lang.Thread@1::Account@101:setValue:true:Enter
main:java.lang.Thread@1::Account@-1:setValue:true:Exit
main:java.lang.Thread@1::Account@-1:<init>:false:Exit
.
.
.
Thread-0:Employee@112::Bank@95:getAccount:true:Enter
Thread-0:Employee@112::Bank@-1:getAccount:true:Exit
Thread-0:Employee@112::Bank@95:getAccount:true:acquire
Start: Deadlock Detection
End: Deadlock Detection
Thread-1:Employee@114::Bank@95:getAccount:true:Enter
Thread-1:Employee@114::Bank@-1:getAccount:true:Exit
Thread-1:Employee@114::Bank@95:getAccount:true:acquire
Start: Deadlock Detection
End: Deadlock Detection
Thread-0:Employee@112::Bank@95:getAccount:true:Enter
Thread-0:Employee@112::Bank@-1:getAccount:true:Exit
Thread-1:Employee@114::Bank@95:getAccount:true:Enter
Thread-0:Employee@112::AccountingTransaction@157:<init>:false:Enter
Thread-1:Employee@114::Bank@-1:getAccount:true:Exit
Thread-6:AccountingTransaction@141::Account@101:drawValue:true:acquire
Start: Deadlock Detection
AccountingTransaction@133:Account@102
AccountingTransaction@141:Account@101
Description:AccountingTransaction@133 acquires Account@102
Description:Account@102 is locked by thread AccountingTransaction@141
Description:AccountingTransaction@141 acquires Account@101
Description:Account@101 is locked by thread AccountingTransaction@133
DEADLOCK
End: Deadlock Detection
```

Figure 10.4: Part of the Trace

account.

### 10.1.2 Tracing

In order to find out if the behaviour is as desired, we make a non-deterministic test run with the JAVIS tracer. The banking example has to be started with a port and in suspend mode. The last argument is the class which contains the method main().

```
java -Xrunjdwp:transport=dt_socket,server=y,suspend=y
-Xdebug -Xnoagent -Djava.compiler=none
Main
```

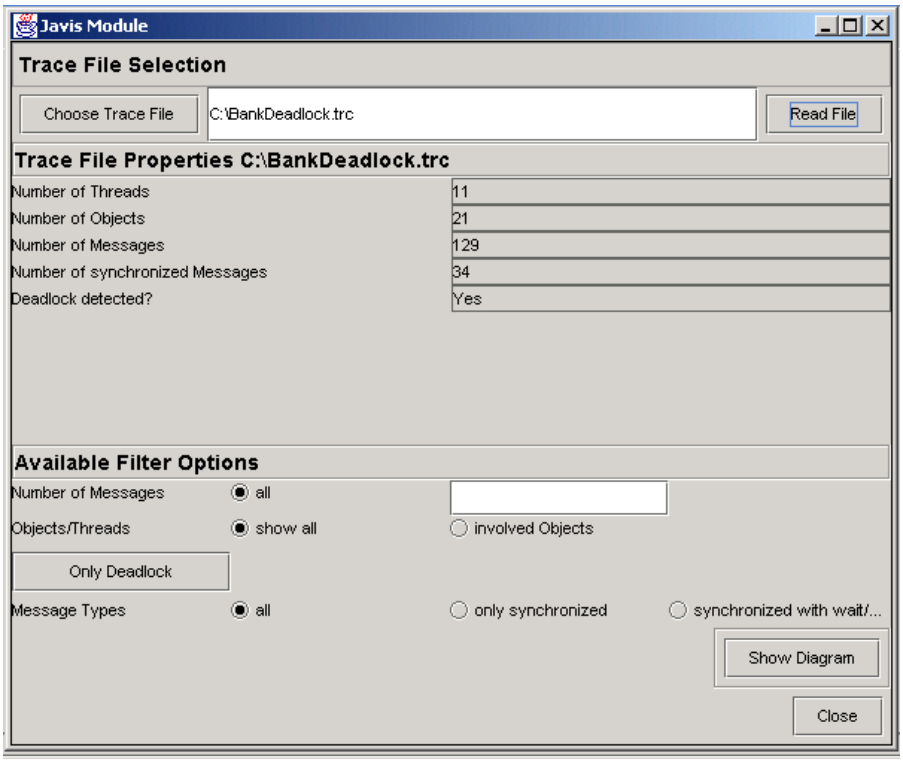


Figure 10.5: Importing a Trace in Together

When the program is started in this way, the command line returns a port number. The tracer can connect to a program started in the above way. It can also directly start a program itself. Fig. 10.3 depicts the main dialog window of the tracer. The port number of the started banking application has to be entered.

### 10.1.3 Automated Deadlock Detection

The tracing stops automatically because a deadlock has been detected. The resulting trace is depicted in Fig. 10.4. The information about the deadlock is part of the trace in our trace format.

### 10.1.4 Trace and Deadlock Visualisation

The generated trace can be imported into Together using the JAVIS plug-in. In the corresponding dialog in Fig. 10.5 different options can be chosen. Either the entire trace, the methods involved in the deadlock, or the deadlock only can be visualised. Also a stepwise visualisation is offered.

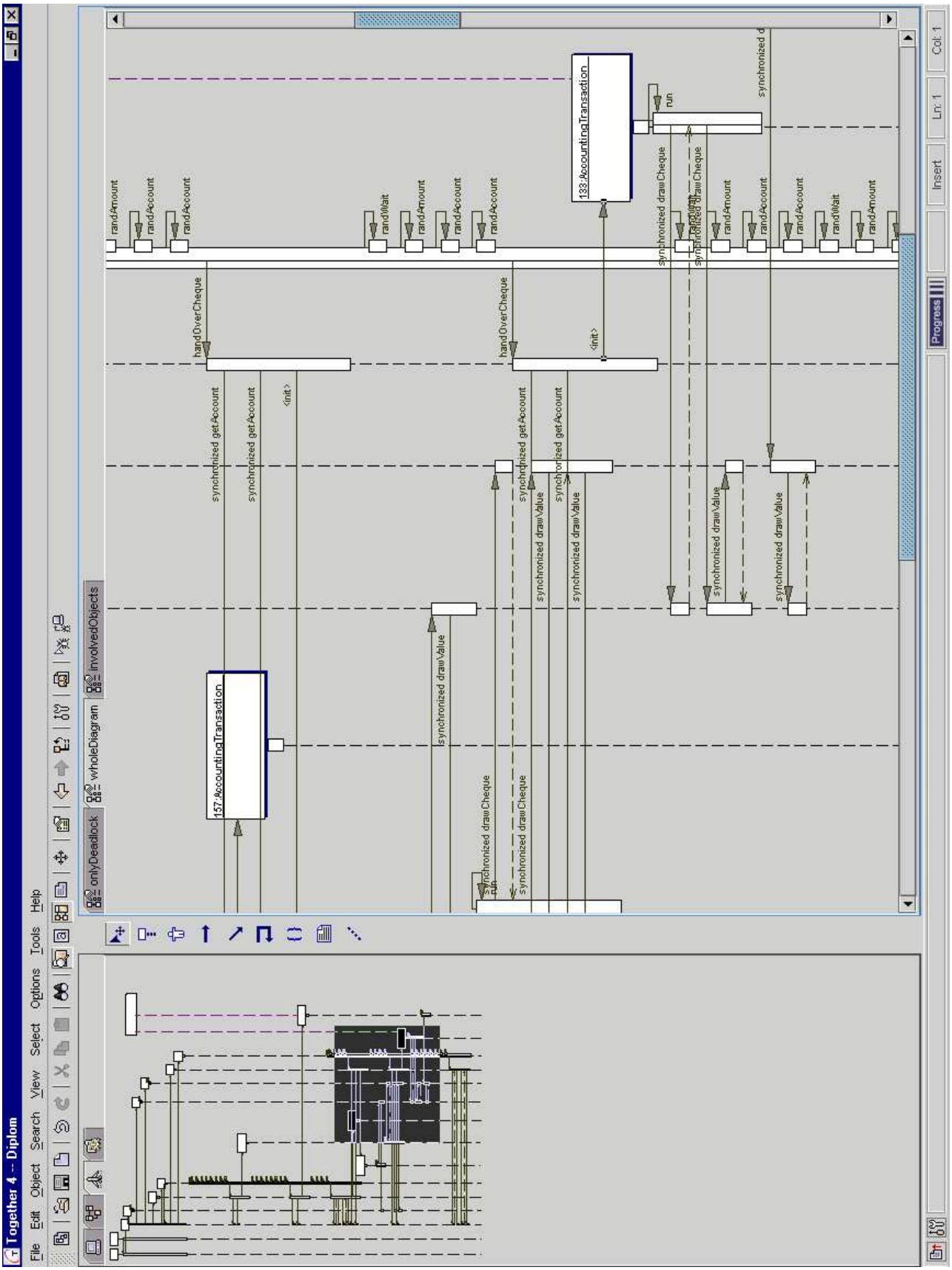


Figure 10.6: Sequence Diagram of a Trace

The complete trace can be visualised as sequence (see Fig. 10.6). In the sequence diagram our profile allows the identification of a method with a synchronized-keyword. The methods involved in the deadlock can be visualised as a sequence, too. Here we prefer to show the corresponding collaboration diagram (see Fig. 10.7). Together can generate the corresponding collaboration automatically. Here, the stereotypes for locking and acquiring are used. It is also possible to visualise the deadlock separately (see Fig. 10.8).

The sequence and collaboration diagrams generated by our extension depict the Java execution more accurate than the sequence diagram which was generated with Together from the Java source code because the profile distinguishes mutual exclusive calls based on synchronized.

It is also possible to visualise the complete trace in Together, see Fig. 10.9. Together is able to display traces with over hundred calls. It also has a zoom-in and a zoom-out functionality. However, zooming-out does not scale very well. Labels cannot be read any longer. It would be better to have a way to compact a view while keeping important information.

### Correcting the Java Code

To avoid the deadlock the code can be corrected such that the accounts are not used via nesting but in parallel.

```
class Account{
  private long value;
  public void synchronized setValue(long newValue){
    value = newValue
  }
  public long synchronized getValue(){
    return value
  }
  public void synchronized drawValue(long amount){
    setValue(getValue() - amount);
  }
  public void synchronized drawCheque(long amount){
    setValue(getValue()+ amount);
  }
}

class AccountingTransaction extends Thread {
  public void run(Account target, Account receiver, long amount) {
    target.drawValue(amount);
    receiver.drawCheque(amount);
  }
}
```

Other recommended strategies are *lock ordering*, i.e. enforcing that locks are always accessed in the same order (see for instance [Lea00]), or a *gate lock*, i.e. using locks which guard a set of other locks.

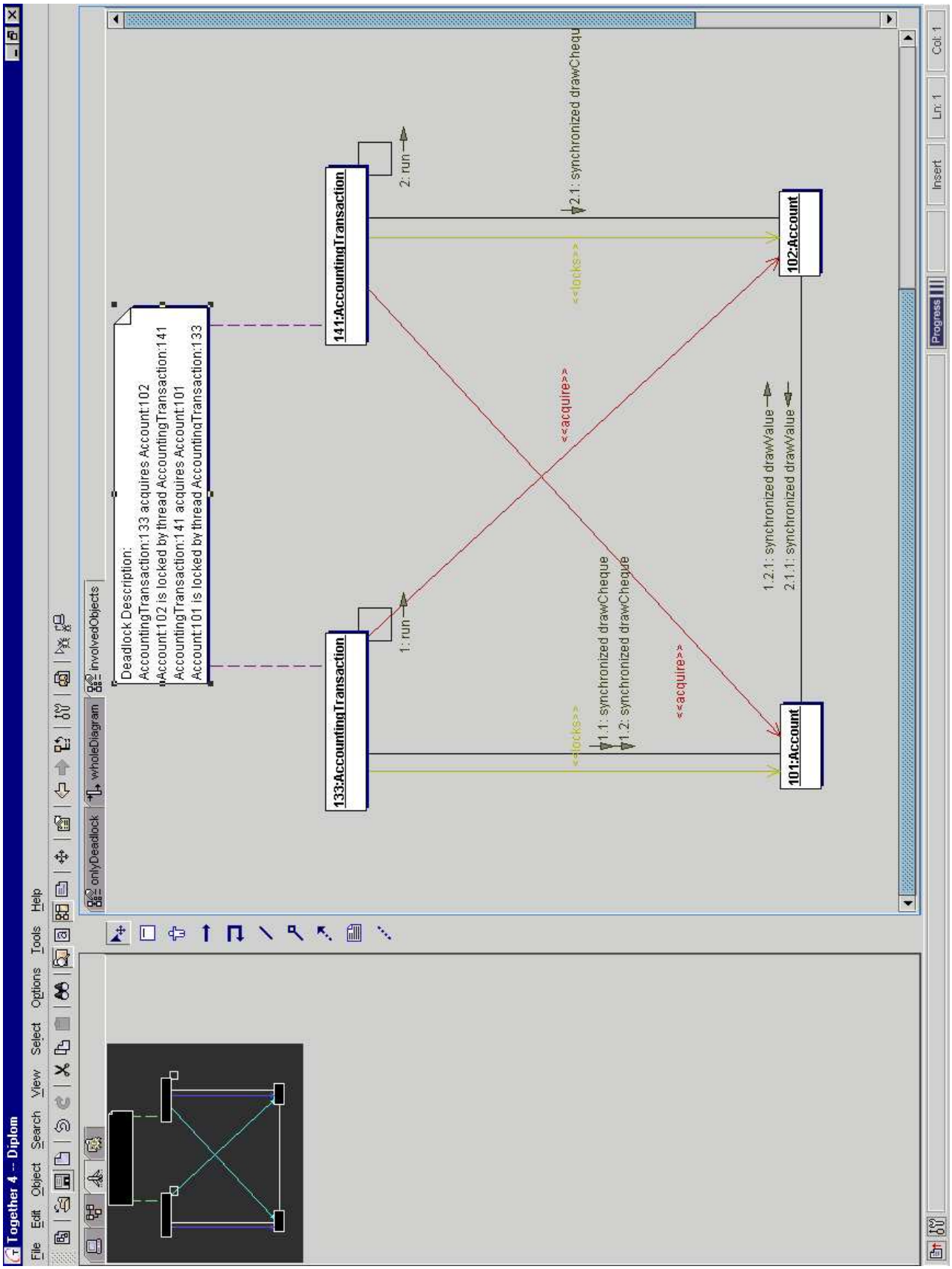


Figure 10.7: Collaboration Diagram with Involved Methods

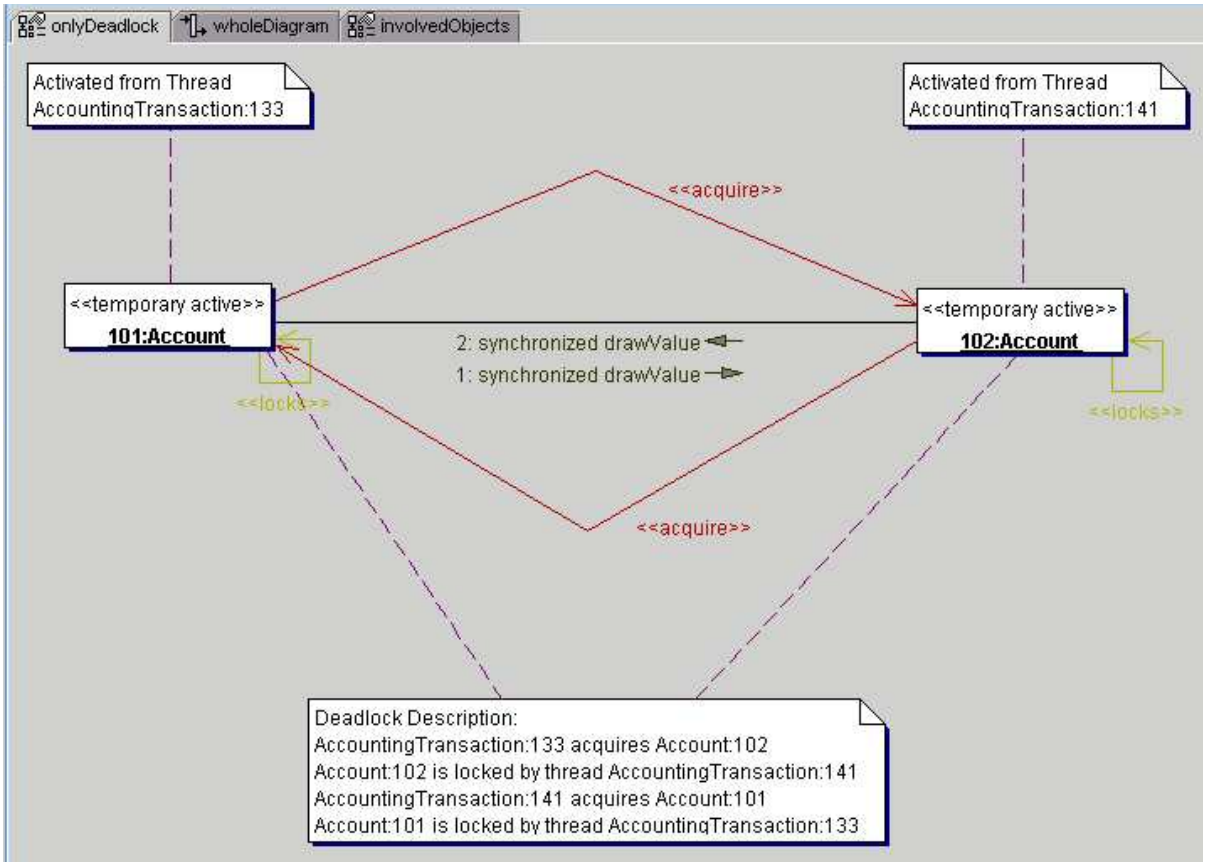


Figure 10.8: Collaboration Diagram with Deadlock

The banking example should be seen as an illustrating example. We do not claim that, in a real software, the code to transfer money between account looks exactly like here.

## 10.2 Example for General Purpose Tracing

Here we report on an experiment with remote tracing and visualisation of a real-size concurrent Java application.

### 10.2.1 A Simulation Software Example

The JEVOX application [JEV] is a simulation of a factory environment. In this factory, shuttles are running on tracks, stopping from time to time at robots, from which they obtain goods or which remove goods. The tracks and the behaviour of shuttles is visualised on the screen.

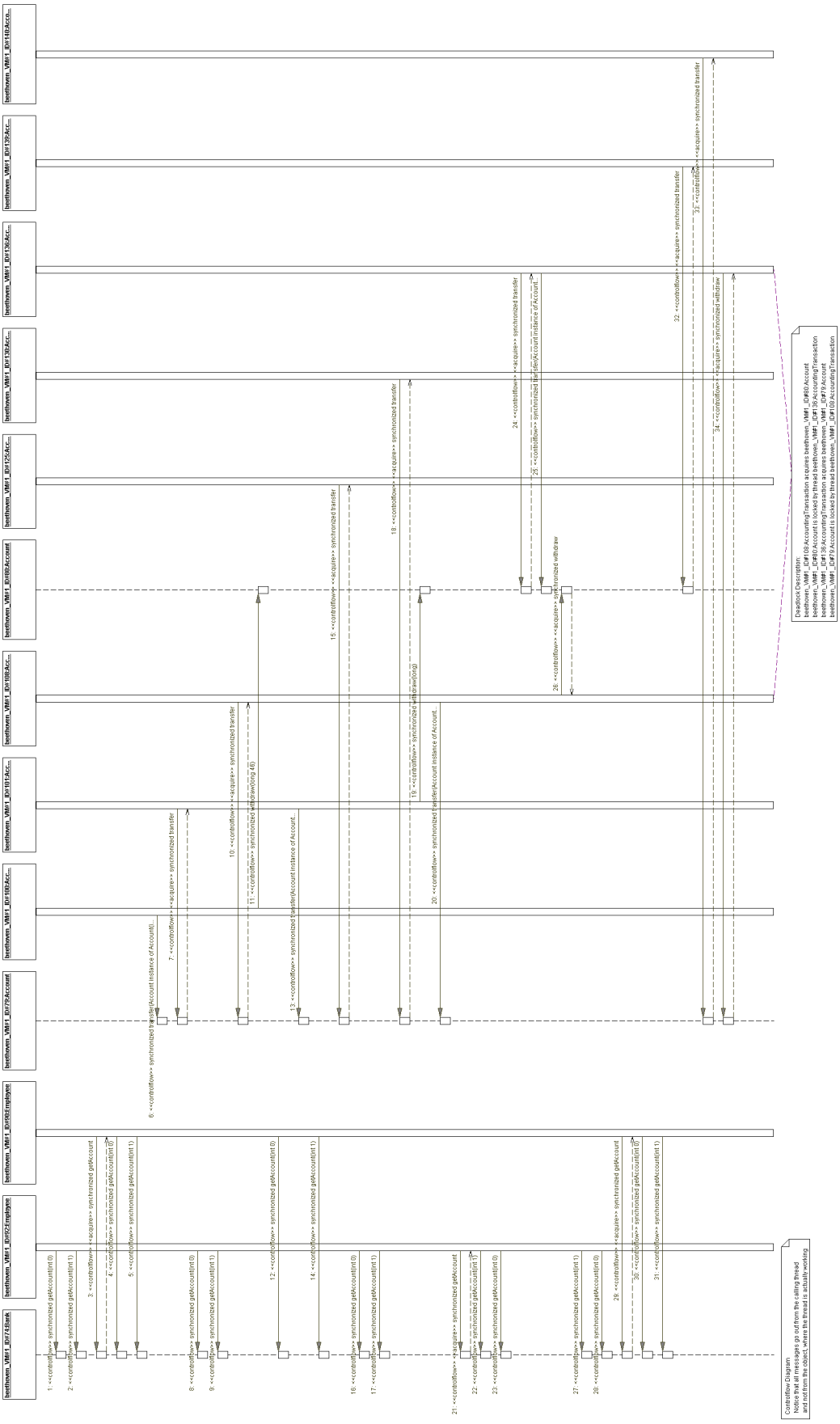

Figure 10.9: Complete Trace Visualisation in Together

The JEVOX simulation has been generated from a high-level executable specification. During the testing of the simulation, a failure occurred. The simulation crashed when two shuttles were closely following each other.

### 10.2.2 Tracing and Visualising JEVOX

We traced this situation in order to capture the exact behaviour before the crash. The remote tracing facility was very convenient. We did not have to install the software we wanted to trace on our own machine nor did we have to install the tracer on the application machine. Instead, the application had to be restarted with opening a port. The tracing facility running on a different machine connected to the port. The only preliminary for this was that both machines were connected via a network.

Note that the code we traced was generated code. Figure 10.10 shows the first part of the trace. Although it is generated code, the main principle of the code is obvious. The object on the left is the execution engine of the simulation. It triggers a robot to interact with a shuttle. The robot makes several calls on the shuttle and so on.

The trace contained over 500 calls, the trace file had 173 KByte. Tracing took a few minutes. With our Together extension for trace visualisation we could visualise a bit more than hundred calls. To this end we had manually truncated the trace before visualisation. The trace of the calls of generated methods was easily understandable for the developers of the application. From the trace they understood that methods were called in an unexpected order, which provoked the crash.

Of course, this failure could not be detected automatically because it was a failure in the logic of the application. However, catching a trace automatically, and subsequently viewing the trace was more convenient than the stepwise debugging of the application.

## 10.3 Summary

In this chapter we first described how to work with the JAVIS prototypes. This was illustrated with a small-scale example, in which a deadlock could be successfully detected using automated detection.

Then we described how we traced a real-size software project in order to help the developers understand a failure in the concurrent application logic. Here, it was the general purpose tracing and visualisation facility that helped the developers in understanding the failure, which was caused by an unexpected execution order.

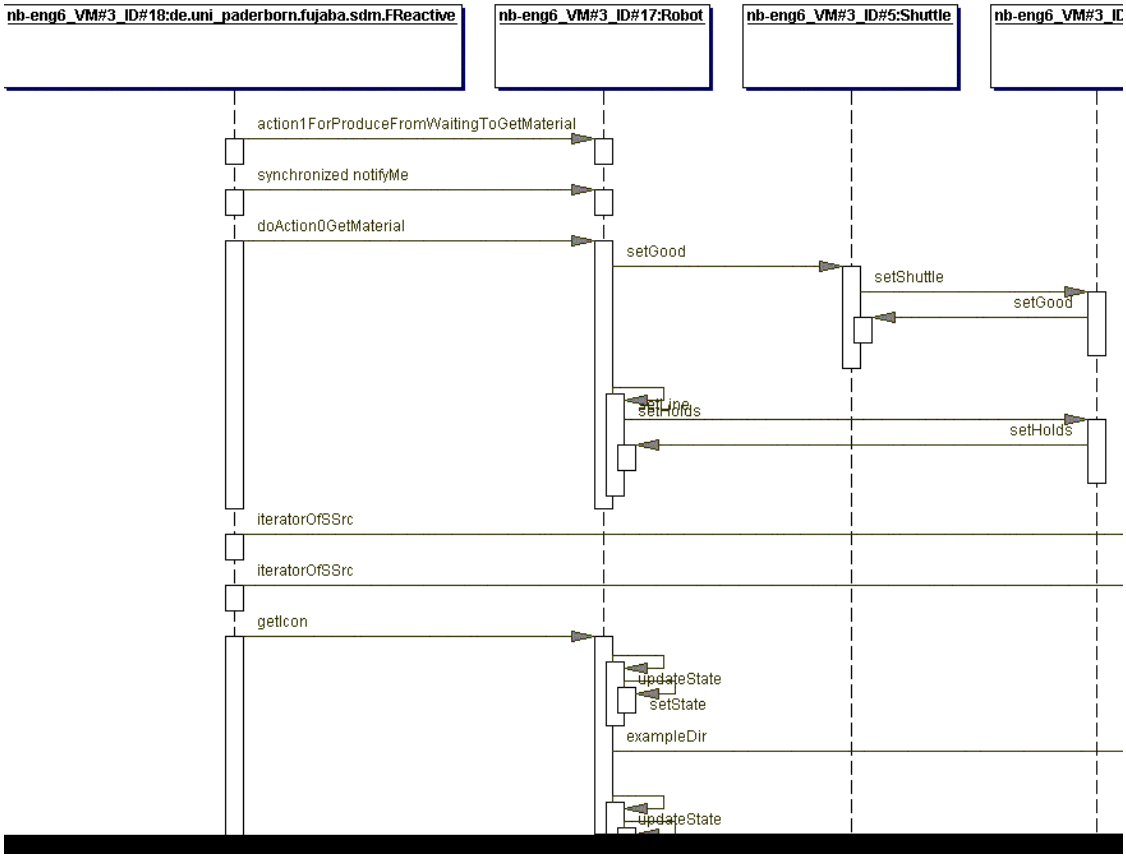


Figure 10.10: Part of the Remote Trace

# Chapter 11

# Conclusion

This thesis describes an approach for automated detection of concurrent liveness failures and their potentials in the execution of Java programs. It contributes new concepts to or adapts existing ones in four different areas: the domain of *concurrent Java liveness failures*, the area of *runtime data collection*, the area of *runtime analysis*, and the area of *software visualisation*.

The focus of this thesis is on presenting a coherent treatment of these areas by showing how concepts from the domain can be put to work in the other areas in order to achieve the overall goal. This results in concepts for tool support and in a set of prototypes which seamlessly work together.

In the following, the contributions are described in more detail and evaluated.

## 11.1 Contributions

1. **Liveness Failure and Potential Classification**

   For each conceivable failure and potential a detailed description was provided. It was exemplarily illustrated with a UML sequence diagram.

   The failures and potentials were analysed for the Java synchronisation mechanisms involved. A set of thread dependencies was identified as a source of failures and potentials.

2. **Formalisation of Liveness Failures and Potentials**

   The synchronised behaviour of a single thread was modelled using a UML statechart. Based on the statechart, the different threads in an executing Java program and the dependencies between them were modelled.

Based on these models, the failures and potentials were specified using logic formulas.

3. **Detection of Liveness Failures and Potentials**

   Algorithms were given for detecting failures involving cycles and for detecting their potentials.

4. **Tracing of Concurrent Java Programs**

   A trace format and a tracing method for concurrent Java programs were defined which are suitable for the implementation of the specified algorithms.

5. **Visualisation of Liveness Failures and Potentials**

   A UML profile for visualising the execution of concurrent Java programs was developed.

   It allows failures and potentials to be visualised in execution scenarios.

6. **Tools and Prototypical Evaluation of Concepts**

   Prototypes for General Purpose Tracing, Analysis, and UML-based Visualisation were implemented.

   Their intended use was illustrated with examples.

## 11.2 Evaluation

The above contributions are assessed for their usefulness, i.e. how well they solved the goals.

### Domain of Liveness Failures and Potentials

The analysis of the domain of liveness failures was carried out in depth. Failures were described at different levels of detail and abstraction including intuitive visualisations and formalisations.

This filled in a gap in the literature. It helped us a lot in understanding the domain. This kind of material would have been able to save the students involved in the projects related to this thesis a lot of time when learning which pitfalls to avoid.

We also made the experience that the explicit analysis of how and why failures happen gives programmers, which have to deal with Java threads from time to time, valuable feedback.

### Statechart

In the same vein, the statechart was a major part to better understand the domain and to capture the analysis made.

In fact, the formalisation of thread dependencies helped in two respects. Firstly, it was more accurate than the informal description, which was only accompanied by the visualisation with UML sequence diagrams. This means that it allowed to specify dependencies explicitly such as which locks are held by a thread. This accurateness was needed to specify algorithms.

Secondly, the statechart allowed us to find additional failures about which we had not read in the literature. It was straightforward to use the statechart to check different combinations of execution histories of threads.

The use of the statechart also raised further issues which could not be treated here such as formal semantics. Nevertheless, we think that a visual formalism has advantages e.g. over textual formalisms albeit other formalism might be more mature.

### Algorithms

For a prototype, our choice of algorithm is adequate. We did not need to be as fast as possible. If algorithms for all kind of failures and potentials run in parallel, performance becomes an issue. For a commercial tool, one should try to make the algorithms as efficient as possible.

### Tracing

The trace format we have specified is a general purpose format for concurrent Java programs. We think that we will be able to experiment with it also for other kind of goals.

We have discussed tracing methods in depth, including technologies which we consider state-of-the-art such as the Java Platform Debugger Architecture (JPDA). We have contributed to the dissemination of these technologies by using them and by reporting our experience with them such as with performance, which we measured with an experiment in the absence of performance data published by the provider of the Java Virtual Machine.

Note that we proposed to use the JPDA for a purpose for which it was not designed in first place. This is also the reason for the lack of performance, which we measured. It is an interesting observation that there is no dedicated support for tracing although dynamic analysis is considered more and more an important source during analysis of software qualities.

### Visualisation

From time to time, general purpose software visualisation prototypes are proposed without supporting a dedicated goal. This typically has the result that they are not very useful. Therefore, we have set out to support a dedicated goal to increase the chance of providing a useful software visualisation tool.

We also have gone further than others and have adapted the UML for the purpose of dynamic visualisation of existing software. Here, our conclusion is that the UML does not provide ideal extensible elements for using it to exactly describe the semantics of a programming language. One can reply that this is not the intention of the UML.

The UML is certainly a debatable candidate for visualisations in the area of debugging. Nevertheless, the UML is a good fit for traces of object-oriented languages like Java. Our conclusion is that it is possible to visualise selected parts of traces with UML. Failures and methods causing potentials are such selected parts.

There is however no evidence that UML is better than others visual languages in this respect. We consider it very difficult to empirically assess these issues, especially if languages are not very different from each other. In the first place, the functionality of a software tool is important.

### Prototypes

For the prototypes we were using state-of-the-art technology which is scalable and turned out to be very stable. The choice for the state-of-the-art technology was made not only from the academic point of view but also from a practical perspective, so that the solution can be adapted for commercial tools. Our experiences and results are of interest for the community of program developers and developers of tools for program developers.

For the tracing component, we have tried to provide essential features such as tracing an already running program remotely. Also, we have provided concepts how tracing can be used within many environments. These features were easily accomplished using the Java Platform Debugger Architecture. The user control we provided with the tracer is not very flexible compared to a debugger. It mainly lacks interactive control.

The UML CASE tool Together is widely used in industrial settings. Together is still having an impact also if new tools like Eclipse [Ecl] have appeared. Its UML modelling facilities are more mature than UML plug-ins for Eclipse.

It was straightforward to use the Together Open API to write a plug-in for generating diagrams from the trace file. Once this was done, we benefited

from the standard diagram editor facilities. We acknowledge that this saved a lot of development time.

For our UML profile it was not a problem that the Together version we used did not support graphical stereotypes or constraints on stereotypes.

# Chapter 12

# Outlook

The outlook is concerned with three issues: addressing parts of the thesis which can be treated in more depth, the evolution of our ideas in the evolving domain of Java and UML, and the transfer of our ideas to related domains.

## 12.1 Remaining Issues

### Concurrency Libraries

Our approach focuses on tracing concurrency implemented directly in Java. Because of the deficiencies of the concurrency constructs, there is a movement to develop suitable Java libraries for concurrency [Lea03]. When such libraries are used, our approach can be of limited usefulness. Firstly, the user does not want to debug the library. Secondly, liveness failures based on software with high-level protocols cannot be detected in the same way as the basic Java failures.

Such protocols use synchronized, wait(), and notify/All() to implement their high-level synchronisation strategies. Therefore, when a thread cannot make progress this is also caused by the typical method calls and statements. But this kind of waiting has a special meaning in terms of a given protocol. This meaning cannot be detected by our approach. Our approach would have to be adapted to each library used.

### Formalisation

We have already pointed out that the formalisation of thread synchronisation is a field with open issues. Existing formalisations do not yet cover it completely. Our approach using statecharts still lacks a formal semantics. This

is a prerequisite for the required validation of properties of the statechart and for the required validation against the Java Language Specification.

Also, it could be considered to use and extend one of the other formalisms presented in this thesis.

### Tracing

Tracing of Java still has deficits. Especially, performance has to be improved.

To foster interoperability of our approach with tools from testing, we think it is worthwile to consider other standardised formats such as for testing.

The control over the tracing process could be more flexible. Tracing could be combined more tightly with debuggers. For instance, debuggers could automatically generate small traces to improve their visualisation of history. Tracing on the other hand could generate breakpoints at interesting lines of code and start the debugger when these breakpoints are hit. The debugger is important because it allows better browsing of object structures. We think it would be a good decision to integrate tracing into Eclipse.

### Visualisation

Typically, traces are visualised using 2-dimensional languages such as the UML. So far, there is not yet a 3-dimensional trace language but it has already been considered in research.

## 12.2 Evolution

The area of software development is constantly changing. Therefore, practical solutions need to be constantly adjusted, too. Software development tools must continuously be adapted to improvements of programming languages, middleware, and operating systems.

Besides the many benefits and inventions of Java, many of its languages concepts are not new, they appeared already in other languages. The situation is even worse, as the design of some initial features was not even state-of-the-art. Therefore, Java undergoes bigger changes than other programming languages.

### Java 1.5

Java 1.5.0, also called 5.0, was released only recently [Jav04]. For concurrent programming, it provides new classes such as a lock class for mutual exclu-

sion. This will surely have an impact on the failures we have to deal with. In analogy to the above discussion on third party concurrency libraries, the tracing, the analysis, and the visualisation will have to cover the additional classes.

The lock class for mutual exlusion is very similar to synchronized. Its advantage is that the thread waiting for the mutex can be woken up. This can be used to resolve deadlocks.

Nevertheless, the provided new classes are at a similar level of abstraction and hence our ideas can be adapted.

#### Java Virtual Machine Tooling Interface (JVMTI)

Also, Java 1.5 has deprecated the use of the native profiling and debugging interfaces JVMPI and JVMDI. They are replaced by the Java Virtual Machine Tooling Interface (JVMTI). In subsequent releases, the deprecated interfaces will be removed.

Therefore, all existing tools based on the JPDA have to be ported to the new interface. The JVMTI is an answer to the many requests for feature enhancements and hopefully, the JMVTI will have erased some of the problems of the JPDA, and hopefully, it will provide implementations of methods which have already been proposed in the JPDA but which have never been implemented. The JVMTI will support the deadlock detection we had still to program ourselves.

#### UML 2.0

Not only programming languages are evolving, also modelling languages. Our approach is still feasible with the UML 2.0 [UMLb]. Without going into details we conclude that the amendments to interaction diagrams do not conflict with our purposes.

## 12.3 Related Domains

In this thesis, the focus was on Java. There are other concurrent object-oriented languages of industrial strength such as Ada [Ada95] or C♯ [Cs] or even Modula-3 [Mod]. They also face concurrent liveness failures. Our ideas could be adapted and also a UML-based visualisation could be used. For instance, C♯ has a threading concept very similar to Java. The ideas could also be adapted for languages using thread libraries such as POSIX threads.

This thesis only looked at concurrency. In distributed systems, support for failure detection is also an important issue. For instance, the tracing could

be extended to include remote method invocation (RMI). For a distributed setting, time stamps are required. With these preliminaries it is possible to identify deadlocks in RMIs [Sch02].

Tracing has a high potential to gain more impact in the future because it can be used to guide model-checking such as proposed in [Hav00].

# Bibliography


[ACM97] The Debugging Scandal and What to Do About It. *Special Issue Communications of the ACM*, 40(4), April 1997.

[ACM03] Developer Tools - Proceed with Caution. *ACM Queue - Tomorrow's computing today*, 1(6), September 2003.

[Ada95] Ada95. Ada Home, 1995. `http://www.adahome.com`.

[AJ98] AspectJ Specification. Technical report, XEROX, Palo Alto Research Center, `http://www.parc.xerox.com/aop/aspectj`, 1998.

[Ass] Assure Thread Analyzer. Intel. `http://www.intel.com`.

[AWY93] G. Agha, P. Wegner, and A. Yonezawa. *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, 1993.

[Ban04] The Bandera Project. `http://bandera.projects.cis.ksu.edu/`, 2004.

[BCH+00] K. Berg, S. Canditt, A. Hennig, A. Hentschel, E. Reyzl, and J. Schmitz-Foster. Monitoring with TMT - Insight Into Distributed Systems. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2000*, 2000.

[BDE+02] R. Brown, K. Driesen, D. Eng, L. Hendren, J. Jorgensen, C. Verbrugge, and Q. Wang. STEP: A Framework for the Efficient Encoding of General Trace Data. In *PASTE 02*, 2002.

[BS99] E. Boerger and W. Schulte. A Programmer Friendly Modular Definition of the Semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523, pages 353–404. LNCS, 1999.

[BT98] A. Bechini and K. Tai. Design of a Toolset for Dynamic Analysis of Concurrent Java Programs. In *Proc. International Workshop on Program Comprehension IWPC98*, 1998.

[But97] D. Butenhof. *Programming with POSIX Threads.* Addison-Wesley, 1997.

[CDH+00] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting Finite-State Models from Java Source Code. In *22nd International Conference on Software Engineering*, June 2000.

[CES71] E. G. Coffmann, M. J. Elphick, and A. Shoshani. System Deadlock. *ACM Computing Surveys*, 3(2):67–78, June 1971.

[CGR01] S. Canditt, K. Grabenweger, and E. Reyzl. Trace-basiertes Testen von Middleware. *Java Spektrum*, (3):17–32, May 2001. (in German).

[CKRW99] P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An Event-Based Structural Operational Semantics of Multi-threaded Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523, pages 157–200. LNCS, 1999.

[CLL+02] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. In *ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, June 2002.

[Cs] C♯. Microsoft. `http://www.microsoft.com`.

[CT91] R. Carver and K. Tai. Replay and Testing for Concurrent Programs. *IEEE Software*, 8(2):66–74, 1991.

[CZ02] J.-D. Choi and A. Zeller. Isolating Failure-Inducing Thread Schedules. In *International Symposium on Software Testing and Analysis ISSTA*, 2002.

[DD00] E.-E. Doberkat and S. Dissmann. *Einführung in die objektorientierte Programmierung.* Oldenbourg, 2000. (in German).

[Ecl] Eclipse. IBM. `http://www.eclipse.org`.

[EHSW99] G. Engels, R. Hücking, S. Sauer, and A. Wagner. UML Collaboration Diagrams and their Transformation to Java. In *Proceedings of UML 1999*. LNCS, 1999.

[Eng99] J. Engel. *Programming for the Java Virtual Machine*. Addison-Wesley, June 1999.

[FR01] M. A. Francel and S. Rugaber. The Value of Slicing While Debugging. *Science of Computer Programming - Special Issue on Program Comprehension (Pittsburg, PA, 1999)*, 40(2-3):151–169, 2001.

[GH03] A. Goldberg and K. Havelund. Instrumentation of Java Bytecode for Runtime Analysis. In *Fifth ECOOP Workshop on Formal Techniques for Java-like Programs*, volume 1885, July 2003.

[GJM91] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, 1991.

[GJS97] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1997.

[GJSB00] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, 2nd edition*. Addison-Wesley, 2000.

[Har00] J. Harrow. Debugging Multithreaded Applications on Compaq Tru64 UNIX Operating Systems. Technical report, Compaq Computer, 2000.

[Hav00] K. Havelund. Using Runtime Analysis to Guide Model Checking of Java Programs. In *Proc. 7th Intl. SPIN Workshop on Model Checking of Software*, volume 1885. Lecture Notes in Computer Science, August 2000.

[HD01] J. Hatcliff and M. Dwyer. Using the Bandera Tool Set to Model-Check Properties of Concurrent Java Software. In *Proceedings of CONCUR 2001*, volume 2154. Lecture Notes in Computer Science, June 2001. Invited Tutorial Paper.

[HG97] D. Harel and E. Gery. Executable Object Modeling with Statecharts. *IEEE Computer*, 30(7):31–42, 1997.

[HK04] D. Harel and H. Kugler. The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML). 2004.

[HLL03] A. Hamou-Lhadj and T. Lethbridge. Compact Trace Format (CTF). In *Workshop ATEM2003 co-located with WCRE 2003*, 2003.

[Hol71] R. Holt. Some Deadlock Properties of Computer Systems. *ACM Computing Surveys*, 3(2), June 1971.

[Hol98] A. Holub. Programming Java Threads in the Real World - The Perils of Race Conditions, Deadlock, and Other Threading Problems. Java World, Java Toolbox Column, `http://www.javaworld.com`, October 1998.

[Hol00] A. Holub. If I were king: A proposal for fixing the Java programming language's threading problems. IBM Developer Works, Java Technology, `http://www.holub.com/publications`, October 2000.

[HP98] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*. McGraw-Hill, 1998.

[HR01] K. Havelund and G. Rosu. Java PathExplorer - A Runtime Verification Tool. In *Proc. The 6th International Symposium on AI, Robotics and Automation in Space*, May 2001.

[Hya] Hyades for Eclipse. IBM. `http://www.ibm.com`.

[I3E90] Standard Glosary of Software Engineering Terminology Std 610.12-1990. IEEE Computer Society, 1990.

[Jav01] Java2Platform, Standard Edition, V1.4.2. `http://java.sun.com/j2se/1.4.2/docs/api/index.html`, 2001.

[Jav04] Java2Platform, Standard Edition, V1.5.0. `http://java.sun.com/j2se/1.5.0/docs/api/index.html`, 2004.

[JBd01] JBuilder 5. Borland Software Corporation, 2001. `http://www.borland.com/jbuilder/`.

[JDv01] JDeveloper. Oracle, 2001. `http://www.oracle.com/technology/products/jdev/`.

[JEV] JEVOX - Shuttle Simulation Environment. `http://wwwcs.upb.de/cs/jevox/`.

[Jin] IBM Jinsight - A Visual Tool for Optimizing and Understanding Java Programs. W. De Pauw and others. `http://www.research.ibm.com/jinsight`.

[JMT] JMThreadUnit. `http://www.sourceforge.net`.

[Jon97] M. Jones. What Really Happened on Mars. `http://research.microsoft.com/m̃bj/Mars_Pathfinder/Authoritative_Account.html`, 1997.

[JPD] JDK 1.4 Java Platform Debugger Architecture. JavaSoft. `http://www.javasoft.com/products/jpda/`.

[JPF03] Java Pathfinder - A Formal Methods Tool for Java. NASA, 2003. `http://ase.arc.nasa.gov/visser/jpf/`.

[JPr00] Threadalyzer. JProbe, 2000. `http://www.jprobe.com`.

[JTe] JTest. `http://www.sourceforge.net`.

[JTr] JTrek. COMPAQ/SRC. `http://research.compaq.com/SRC/`.

[JUn] JUnit. JProbe. `http://www.jprobe.com`.

[KG01] R. Kollmann and M. Gogolla. Capturing Dynamic Program Behaviour with UML Collaboration Diagrams. In *5th European Conference on Software Maintenance and Reengineering. IEEE*, 2001.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of ECOOP '97*, number 1241 in LNCS, pages 220–243, 1997.

[Kra98] E. Kraemer. Visualizing Concurrent Programs. In *Software Visualization: Programming as a Multimedia Experience*, pages 237–256. MIT Press, Cambridge, MA, 1998.

[KRR98] D. Kimelman, B. Rosenburg, and T. Roth. Visualization of Dynamics in Real World Systems. In *Software Visualization: Programming as a Multimedia Experience*. MIT Press, Cambridge, MA, 1998.

[L4J] Log4J. `http://logging.apache.org/log4j`.

[Lam77] L. Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, March 1977.

[Lea97] D. Lea. *Concurrent Programming in Java, Design Principles and Patterns*. Addison-Wesley, 1997.

[Lea00] D. Lea. *Concurrent Programming in Java, Design Principles and Patterns (2nd Ed.)*. Addison-Wesley, 2000.

[Lea03] D. Lea. Java Concurrency Utility Library (developed under JSR-166). `http://gee.cs.oswego.edu/dl/`, 2003.

[Lew03a] B. Lewis. Debugging Backwards in Time. In *AADEBUG 2003*, 2003.

[Lew03b] B. Lewis. Omniscient Debugging. In *ASARTI Workshop ECOOP*, 2003.

[LF98] H. Lieberman and C. Fry. ZStep95: A Reversible, Animated Source Code Stepper. In *Software Visualization: Programming as a Multimedia Experience*. MIT Press, Cambridge, MA, 1998.

[LMM99] D. Latella, I. Majzik, and M. Massink. Towards a formal operational semantics of UML statechart diagrams. In *Proceedings of Formal Method for Open Object-based Distributed Systems FMOODS 1999*. Kluwer Academic Publishers, 1999.

[LRRM03] H. Leroux, A. Requile-Romanczuk, and C. Mingins. JACOT: A Tool to Dynamically Visualise the Execution of Concurrent Java Programs. In *Proc. PPPJ 2003*, 2003.

[LS02] C. Lewerentz and F. Simon. Metrics-based 3D Visualization of Large Object-Oriented Programs. In *Proceedings of the First IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2002)*, pages 70 – 77. IEEE Computer Society Press, 2002.

[Meh02] K. Mehner. JaVis: A UML-Based Visualization and Debugging Environment for Concurrent Java Programs. In S. Diehl, editor, *Software Visualization, International Seminar, Dagstuhl Castle, Germany, May 20-25, 2001. Revised Papers*, volume 2269, pages 163 – 175. LNCS, 2002.

[Meh03] K. Mehner. Performante Überwachung von Methodenaufrufen mit JPDA. *Java Spektrum*, (6):42–46, November 2003. (in German).

[Mit00] W. D. Mitchell. *Debugging Java - Troubleshooting for Programmers*. McGraw-Hill Computing, 2000.

[MK99] J. Magee and J. Kramer. *Concurrency: State Models & Java Programs*. John Wiley & Sons, 1999.

[MLMD01] J.I. Maletic, J. Leigh, A. Marcus, and G. Dunlap. Visualizing Object-Oriented Software in Virtual Reality. In *Proceedings of the 9th IEEE International Workshop on Program Comprehension (IWPC'01)*. IEEE Computer Society Press, 2001.

[Mod] Modula-3. Compaq Research. `http://research.compaq.com/SRC/modula-3/html/home.html`.

[MW99] K. Mehner and A. Wagner. On the Role of Method Families in Aspect-Oriented Programming. In *Workshop on Aspect-Oriented Programming, ECOOP 1999*, 1999.

[MW00] K. Mehner and A. Wagner. Visualizing the Synchronization of Java-threads with UML. In *Proc. IEEE Symposium on Visual Languages, Seattle*, pages 199–206. IEEE Computer Society Press, 2000.

[MW03] A. Marburger and B. Westfechtel. Tools for Understanding the Behavior of Telecommunication Systems. In *Proceedings of the 25t International Conference on Software Engineering ICSE03*, 2003.

[Nag03] M. Nagl. *Softwaretechnik mit Ada 95-Entwicklung großer Systeme (2., durchgesehene Auflage)*. Vieweg Verlag, 2003. (in German).

[Oec02] R. Oechsle. JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI). In S. Diehl, editor, *Software Visualization, International Seminar, Dagstuhl Castle, Germany, May 20-25, 2001. Revised Papers*, volume 2269. LNCS, 2002.

[OL82] S. Owicki and L. Lamport. Proving Liveness Properties of Concurrent Programs. *ACM Transactions on Program Languages and Systems*, 4(3):455–495, 1982.

[OW99] S. Oaks and H. Wong. *Java Threads (2nd Ed.)*. O'Reilly, 1999.

[PJM+02] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang. Visualizing the Execution of Java Programs. In S. Diehl, editor, *Software Visualization, International Seminar, Dagstuhl Castle, Germany, May 20-25, 2001. Revised Papers*, volume 2269. LNCS, 2002.

[PKV94] W. De Pauw, D. Kimelman, and J. Vlissides. Modeling Object-Oriented Program Execution. In *ECOOP 94*, 1994.

[PKV98] W. De Pauw, D. Kimelman, and J. Vlissides. Visualizing Object-Oriented Software Execution. In *Software Visualization: Programming as a Multimedia Experience*. MIT Press, Cambridge, MA, 1998.

[PMR+01] W. De Pauw, N. Mitchell, M. Robillard, G. Sevitsky, and H. Srinivasan. Drive-by Analysis of Running Programs. In *Proceedings for Workshop on Software Visualization, International Conference on Software Engineering 2001*, 2001.

[Ree98] G. Reeves. Re: What Really Happened on Mars? Risks-Forum Digest, January 1998.

[Rei98] S. Reiss. Software Visualization in the Desert Environment. In *Proc. PASTE*, 1998.

[RM02] A. Roychoudhury and T. Mitra. Specifying Multithreaded Java Semantics for Program Verification. In *International Conference on Software Engineering*, 2002.

[Sch02] T. Schattkowsky. Ansätze zur objektorientierten Modellierung nebenläufiger Systeme. Diploma Thesis, University of Paderborn, 2002. (in German).

[SHS03] D. Seifert, S. Helke, and T. Santen. Test Case Generation for UML Statecharts (Short Paper). In *Proc. Perspectives of System Informatics (PSI 2003)*. Springer, 2003.

[SK93] J. Stasko and E. Kraemer. A Methodology for Building Application-specific Visualisations of Parallel Programs. *Journal of Parallel and Distributed Computing*, 18(2):258–264, June 1993.

[SOT] The Software Tomograph (Sotograph). `http://www.sotograph.com/`.

[SRL90] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, page 1175, September 1990.

[Sys99] T. Systä. On the relationships between static and dynamic models in reverse engineering Java software. In *Proceedings of the 6th Working Conference on Reverse Engineering*. IEEE Computer Society Press, 1999.

[Tan97] Andrew Tanenbaum. *Operating Systems: Design And Implementation (2nd Ed.)*. Prentice Hall, 1997.

[Tog] Together Control Center. Borland Software Corporation. `http://www.togethersoft.com/`.

[UMLa] UML Specification Version 1.5 (formal/03-03-01). Object Management Group. `http://www.omg.org`.

[UMLb] UML Specification Version 2.0 (ptc/03-08-02). Object Management Group. `http://www.omg.org`.

[Vis00] Visual Age. IBM, 2000. `http://www.ibm.com`.

[vPG01] C. von Praun and T. R. Gross. Object Race Detection. In *Proc. OOPSLA 2001*, 2001.

[WbS] WebSphere. IBM. `http://www.ibm.com`.

[Wey01] B. Weymann. Visualisierung der Synchronisation in Java-Programmen mit der UML. Diploma Thesis, University of Paderborn, 2001. (in German).

[WM00] P.H. Welch and J.M.R. Martin. A CSP Model for Java Multithreading. In P. Nixon and I. Ritchie, editors, *International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 114–122. IEEE, IEEE Computer Society Press, June 2000.

[XML00] Extensible Markup Language XML Version 1.0 W3C Recommendation. World Wide Web Consortium, 2000. `http://www.w3c.org`.

[ZH02] A. Zeller and R. Hildebrandt. Simplifying and Isolating Failure-inducing Input. *Transactions on Software Engineering*, 38(2):183–200, February 2002.

[ZK00] A. Zeller and J. Krinke. *Programmierwerkzeuge: Versionskontrolle - Konstruktion - Testen - Fehlersuche unter Linux/Unix*. dpunkt-Verlag, 2000. (in German).

[ZS95] Q. Zhao and J. Stasko. Visualizing the Execution of Threads-based Parallel Programs. Technical Report GIT-GVU-95-01, Georgia Institute of Technology, Atlanta, GA, January 1995.

# Index


Abstract State Machines (ASM), 97
aspect-oriented programming (AOP), 60, 140

banking example, 8, 193
bottle neck, 82

classification
  Java concurrency concepts, 58
  Java liveness failure potentials, 84
  Java liveness failures, 84
  Java thread dependencies, 85
  thread lifecycle methods, 102
code instrumentation, 136, 139
code-debug-fix cycle, 13
Communicating Sequential Processes (CSP), 98
concurrency, 5
  error, 7, **62**
  failure, 7, **62**, 68
  failure potential, 10, **64**, 82
concurrent programming, 6, **35**
conditional synchronisation, 53, **55**
control flow state, 93
  intermediate state, 94
cycle detection, 162

data analysis, 31, **157**
data collection, 29, **129**
data race, 70
data visualisation, 32, **167**
deadlock detection, 159
deadlock potential detection, 160
debugger, 13, 143
debugging, 13, 141, 143
domain model, 89

error, 62
execution history, 90
execution state, 90

failure, 7, **62**, 68
failure potential, 10, **64**, 82
false positive, 75
fault, 62
formal semantics, 97

GThreads, 172
Guarded Commands, 98

instrumentation, 136, 139

Java, 6, **35**
  deprecated, 35
  exception, 48
  interrupt, 51
  Java 1.4, 35, 152
  Java 1.5, 210
  join, 52
  Language Specification (JLS), 35, **96**
  Memory Model (JMM), 41, **97**
  notify, 56
  notifyAll, 56
  priority, 45
  Runnable, 42

run, 42
start, 43
scheduling, 39
sleep, 45
static, 102
synchronisation, 38
synchronized, 53
Thread, 36, **42**
Virtual Machine (JVM), 37
wait, 56
Java Libraries, 140
Concurrency, 60, 133, 209
JDI, 141
JPDA, 141
JVMDI, 141
JVMPI, 140
JVMTI, 211
Log4J, 142
Runtime, 142
JAVIS prototype, 129, **191**
-Analysis, 162
-Tracer, 148
-Visualisation, 181
overview, 129
usage examples, 191
JEVOX example, 199
Jinsight, 173

liveness, 7, **68**
failure, 68
blocking on I/O, 79
circular join, 77, 124
deadlock, 71, 122
dormancy, 81
formalisation, 122
indefinite blocking, 73
join-induced deadlock, 78, 124
livelock, 78
missed notification, 73, 122
missing notification, 75
nested monitor lockout, 75, 124
self join, 77, 124
starvation, 81
termination by exception, 79
wait-induced deadlock, 72, 122
failure potential, 82
deadlock potential, 83, 125
formalisation, 124
join-induced deadlock potential, 84, 126
unused interrupt, 84
unused notification, 83, 125
lost update, 40

mistake, 62
model for thread synchronisation, 98
monitor, 53
mutual exclusion, **52**, 53

nondeterminism, 5, **39**

potential (see failure potential), 10, **64**, 82
preemption, 46
priority-inversion, 1, **82**
program comprehension, 179

race condition, 69
requirements, 25
resource dependency graph, 10
reverse engineering, 178

safety, 7, **68**
semantics, 97
software visualisation, 171, 181
stall, 81
statechart for thread synchronisation, 98
Structured Operational Semantics (SOS), 97
synchronisation, 38

testing, 12
thread, 5

thread dependencies, 85
thread synchronisation, 90
    model, 98
    statechart, 98
Together, 178, 189
trace, 18, 31, 131, **148**
trace visualisation, 171, **181**
tracing, 30, **129**, 145

Unified Modeling Language (UML), 32, **177**
    CASE tool, 178
      Together, 178, 189
    collaboration diagram, 177, 182
    interaction diagram, 170, 177
    Object Constraint Language (OCL), 182
    profile, 177, 181
    sequence diagram, 8, 71, 177, 182
    statechart, 99
    stereotype, 184
    tagged value, 183
    UML 2.0, 211

visualisation, 32

wait-for graph, 10
wait-queue, 56